Robustness Testing and Hardening of CORBA ORB Implementations

M.S. Thesis

Jiantao Pan

jpan@ece.cmu.edu

Electrical and Computer Engineering Department

Carnegie Mellon University

Pittsburgh, Pennsylvania 15213

USA

December, 2000

Advisor:	Philip Koopman
Co-advisor:	Daniel Siewiorek

Robustness Testing and Hardening of CORBA ORB Implementations

Jiantao Pan
Department of Electrical and Computer Engineering,
Institute for Complex Engineered Systems
Carnegie Mellon University
Pittsburgh, PA, USA
jpan@ece.cmu.edu

Abstract

Before using CORBA (Common Object Request Broker Architecture) applications in mission-critical scenarios, it is important to understand the robustness of the Object Request Broker (ORB) being used, which forms the platform for CORBA applications. We have extended the Ballista software testing technique to test the exception-handling robustness of C++ ORB client-side application interfaces, and have tested two major versions of three ORB implementations on two operating systems, yielding robustness failure rates ranging from 26% to 42%. To improve ORB robustness, we also introduce a probing method to harden object and pseudo-object related data types against exceptional inputs. A simple probing method for omniORB 2.8 has proven to be effective in eliminating simple cases of robustness failures found during testing. These results suggest that CORBA implementations currently have significant robustness vulnerabilities, but that the problems largely can be overcome with better exception handling approaches.

Acknowledgments

This research was funded primarily by AT&T Labs – Research. The Ballista project is supported in part by DARPA (contract DABT63-96-C-0064).

1. Introduction

The development of CORBA (Common Object Request Broker Architecture) has advanced the concept of component software: diverse software modules implemented in different programming languages that can be integrated as a distributed system using the CORBA interface, and interact in a plug-and-play manner. Using this new component software model, an application can be built by assembling legacy software modules, third-party software modules, and custom-made software modules on a common CORBA platform connected by ORBs (Object Request Brokers), instead of developing a totally custom-made monolithic application, saving both development cost and time to market.

Even mission-critical systems, such as aerospace/defense, banking/finance, healthcare/insurance, e-commerce and telecommunication applications [15], have selected a distributed architecture based upon CORBA. Enterprises and government agencies from all over the world, including NASA, Boeing, Chase Manhattan bank, Motorola, Ericsson, and Independence Blue-Cross, are using CORBA in various applications, from Web-based online banking to cellular phone management, patient care and even applications for the Hubble Space Telescope. Recently, the Object Management Group (OMG) initiated a Space Domain Task Force to encourage the Space and Satellite industry to foster the emergence of cost effective, timely, commercially available and interoperable space, satellite, and ground system domain software components through CORBA technology [17]. While cost and development time is a common consideration for general purpose systems, the robustness of the software – the degree to which a software component functions correctly in the presence of invalid inputs or stressful environmental conditions [5] – is almost always a major concern for mission-critical applications as the examples listed above. It is important that these applications are resistant to failures caused by abnormal inputs.

CORBA applications used in critical scenarios must be robust. But, the heterogeneous environment; the use and reuse of commercial off-the-shelf, third-party and legacy software modules; and their complex interactions will all be likely to trigger exceptions. Thus, the graceful handling of expected and unexpected exceptions is critical for the robustness of CORBA-based systems.

CORBA applications are built upon an Object Request Broker (ORB) interface. The ORB accepts requests from CORBA applications, processes the requests, and manages the communication among different objects, applications and ORBs. The robustness of the ORB is very important since the ORB is the operating platform of CORBA applications and the venue for a CORBA software component to communicate and interact with the rest of the system and the outside world. Developers of critical applications will often need to know the robustness of candidate ORB implementations prior to deciding which one to use. However, methods to evaluate CORBA ORB robustness are rare. We have an urgent need for a method to evaluate the robustness of ORB implementations.

This paper makes three contributions. First, this paper quantitatively measures and compares the exception-handling robustness of CORBA ORB implementations using the Ballista robustness testing methodology. Second, some common exception-handling robustness problems of the ORB implementations under test are identified. Third, suggestions about how to improve the robustness of ORB exception-handling robustness are discussed. The testing tool has been implemented for testing C++ ORBs and used to study two major versions of three ORB products on two operating systems. The operations tested are selected from CORBA 2.1 standard [18]. In the text that follows, Section 2 details the methodology, Section 3 discusses the experimental setup, Section 4 gives results and analysis, and Section 5 lists related work. Conclusions can be found in Section 6.

2. Methodology

Many factors can contribute to the robustness of a software component. Although stressful environmental conditions are important, we focus on measuring how gracefully a software module under test behaves under exceptional inputs. As stated in the definition of software robustness in IEEE standard in terminology [5], it is the software component's responsibility to identify exceptional inputs and handle them gracefully. Typically, two methods are used to handle exceptional input situations: returning error-return codes and raising exceptions. Error-return codes are used extensively in software implemented using the C language, such as POSIX standard [6] function calls in most operating systems. Raising exceptions is used as the standard reporting and handling mechanism for exceptional inputs in the CORBA standard [18] for C++ and Java mappings. In this study, we test and measure exception-handling robustness of C++ ORB implementations.

2.1 Metric

Previous Ballista work [9] proposed a way to measure the robustness of software modules. In this work we extend the same approach to measure CORBA ORB robustness by testing API calls.

The CORBA standard defines a common API for ORB vendors to implement. This API defines a collection of operations that a client or server object can request the ORB to perform on behalf of a user program. The CORBA standard has no restrictions on how the vendors should implement the operations specified in the standard. However, it does have requirements on how an ORB should perform under abnormal input situations. For example,

> "The ORB manages the control transfer and data transfer to the object implementation and back to the client. In the event that the ORB cannot complete the invocation, an exception response is provided." [18]

> "If an abnormal condition occurs during the performance of a request, an exception is returned." [18]

The above specification clearly states that an ORB operation is robust under exceptional inputs if the operation can identify the exceptional inputs and raise exceptions. However, the exceptions should be defined and reflect the actual exceptional situations. We consider raising unknown exceptions non-robust, since no useful informa-

Table 1. Classification of robust and non-robust behaviors.

Robust behaviors	**Successful return (no exceptions)** **Raise CORBA exception**
Non-robust behaviors (Robustness failures)	**Computer crash (Catastrophic failure)** **Thread hang (Restart failure)** **Thread abort (Abort failure)** **Raise unknown exception** False success (Silent failure) Bogus error information (Hindering failure)

tion is given for error recovery and there is no guarantee that the ORB is in a consistent state when an unknown exception is thrown.

Table 1 lists the possible robust behaviors and non-robust behaviors that may happen in testing ORB implementations and maps them to the existing Ballista CRASH scale metric [9]. Among the listed robustness failures, computer crashes, thread hangs, thread aborts and unknown exceptions can be automatically detected by Ballista. False successes and bogus error returns cannot be discovered in an automated manner using the Ballista harness and are not measured in this study. More discussion about false successes (*i.e.* silent failures) and how to estimate them can be found in [9]. In summary, the items highlighted in boldface in Table 1 are the responses that we expect to find in testing the robustness of CORBA ORB implementations.

2.2 Using the Ballista robustness testing methodology

We adopt the Ballista software robustness testing methodology to evaluate ORB implementation robustness. The Ballista testing framework is designed to test COTS (Commercial Off-The-Shelf) software modules for exception-handling robustness problems triggered by invalid inputs.

2.3 CORBA ORB robustness testing architecture

Conceptually, Figure 1 shows the CORBA ORB testing architecture using Ballista. The *Ballista server* performs client code generation and test case generation. The *test manager* of the Ballista client iterates through test cases in the test case database, and manages test case set-up, response monitoring and test case tear-down. The

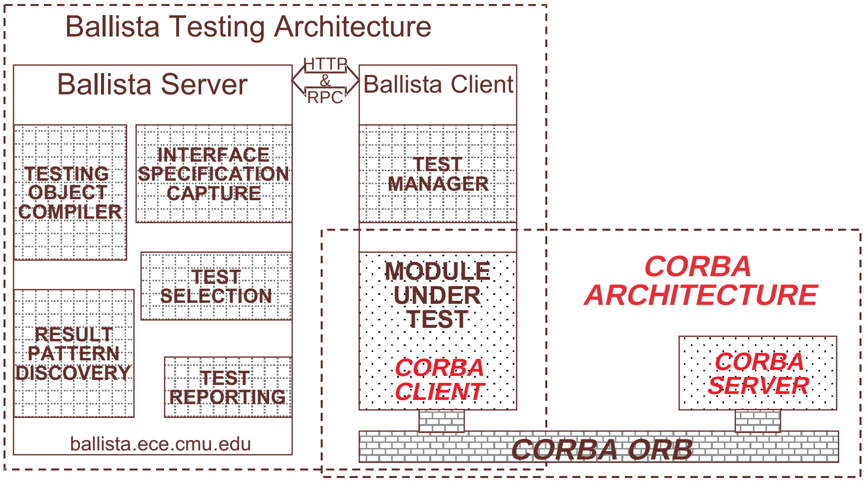


Figure 1. CORBA testing architecture using Ballista.

module-under-test in the Ballista architecture is in this case a CORBA client. In testing, the *module-under-test* communicates and interacts with the *CORBA server* object via the *CORBA ORB* interface, if necessary.

For each test case, the *test manager* spawns a corresponding *module-under-test* thread, and monitors the status of this child thread. Figure 2 shows a generic *module-under-test* in pseudo-code form. The *initialization()* part initiates the ORB and creates necessary variables to be used during the testing process. The *parameter_instantiation(parameter_list)* procedure creates an instance of each parameter from the values specified in the test case database. The actual call to the operation under test appears in *ORB_operation_invocation(parameter_list)*. Exceptions thrown by the ORB operation during testing are caught and analyzed by the *exception_handling()* section.

```
module-under-test{

  initialization();
  parameter_instantiation(parameter_list);
  ORB_operation_invocation(parameter_list);
  ...
  exception_handling();

}
```

Figure 2. *module-under-test* pseudo-code.

```
//userCatches
catch (const CORBA::SystemException& se)
catch (const CORBA::Exception& e)

//low-level exception catches
catch (...)
```

Figure 3. Exception catching levels.

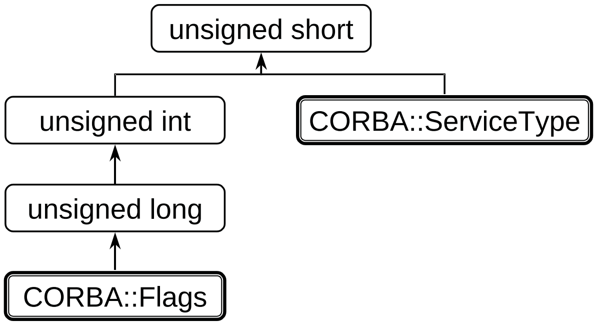


Figure 4. Sample test case inheritance.

The different levels of exception handling are shown in Figure 3. Based on the CORBA standard, CORBA-defined exceptions are caught and broadly categorized into `CORBA::SystemException` and `CORBA::Exception`. If the ORB operation under test only raises these CORBA-defined exceptions, it is considered robust. The *low-level exception catching* makes sure all other unknown exceptions which are not defined in CORBA standard are caught, which generally cannot be recovered and are classified as robustness failures.

2.4 Test case inheritance

A test case inheritance scheme is used to maximize the reuse of test cases. Most ORB operations use CORBA specific data types as parameters. For C++ mappings, the CORBA specific data types are eventually mapped to C++ language data structures. `CORBA::Flags`, for example, is mapped to `unsigned long` in the C++ language. We have designed an inheritance hierarchy to structure CORBA data types. A child data type inherits test cases defined in its parent data type and expands the parent data type by providing test cases specific to the child. As a general rule, a child data type usually expands its parent data type in value range or semantics. In the example inheritance tree shown in Figure 4, data type `CORBA::Flags` inherits all test cases (e.g. `MAX_UNSIGNED_LONG`) defined in the parent data type `unsigned long` (which also inherits test cases from its parent data type `unsigned int`), and adds `ARG_IN`, `ARG_OUT`, `ARG_INOUT`, *etc.*, as its specific test cases. These test cases for CORBA related data types have been selected based on the CORBA specification.

3. Experimental Setup

The Ballista CORBA client has been implemented for two major versions of three ORB implementations for the C++ language mapping on Solaris and Linux platforms.

3.1 ORB platforms under test

There are many ORB implementations available on the market, forming a potential rich set of candidates to conducting our study. We chose Orbix, omniORB and VisiBroker as the candidate platforms based on popularity and availability. Specifically, the following ORBs were tested:

- Orbix 3.0.1 and Orbix 2000
- omniORB 2.8 and omniORB 3.0
- VisiBroker 3.3 and VisiBroker 4.0

Among the ORBs, omniORB is freely available under GNU public license. Orbix 2000 and VisiBroker 4.0 were tested using evaluation downloads from the vendor web sites. All ORBs were tested on a Sparc workstation running Solaris 5.6 to facilitate fair comparisons. Orbix 2000, omniORB 3.0 and VisiBroker 4.0 were also tested on a Pentium machine running RedHat Linux 6.2 (kernel version 2.2.14-5.0smp). The earlier versions of the ORB products are not tested on the Linux platform because VisiBroker 3.3 and Orbix 3.0.1 do not have publicly available Linux releases.

3.2 Test set

A subset of basic ORB operations defined in CORBA standard 2.1 [18] was chosen as the test set. The test set includes operations from basic core interfaces that an ORB should support, such as operations in interfaces `CORBA::Request`, `CORBA::NVList`, `CORBA::Context`, `CORBA::ORB` and `CORBA::Object`. Most of these operations are client-ORB interactions, not client-server operations or inter-ORB operations. No GIOP/IIOP operations are currently under test. However, the test set could be expanded in the future to include inter-ORB client-server calls and other new operations defined in later standards, such as POA operations in CORBA standard 2.3 supported by Orbix 2000, omniORB 3.0 and VisiBroker 4.0.

3.3 Implementation issues

Although Orbix 3.0.1, omniORB 2.8 and VisiBroker 3.3 all claim to support or fully comply with CORBA standard 2.1, and the advanced versions claim to be compatible with CORBA standard 2.3, not all standard operations are supported by every ORB. This is partly because of the rapid updates and ambiguity in the CORBA 2.1 standard. For example, we have observed that the same operations may appear under different names on different ORBs: `CORBA::Object` operation `get_policy()` is defined as `CORBA::Object::_get_policy()` in Orbix [8], appears as `CORBA::Object::get_policy()` in VisiBroker [7], and is undefined in omniORB [11].

Some operations, such as `get_default_context()` and `get_service_information()` for Orbix 2000, have prototypes defined but are not implemented. They always raise CORBA system exception `CORBA::NO_IMPLEMENT` during testing. Although this response is valid per the CORBA standard, it is unfair to compare these operations (which technically would be 100% robust) with implemented versions from other ORBs (which will likely have failures). Therefore, the operations without implementations are deleted from Orbix 2000 test sets.

Due to the above issues, the test operations actually launched for each ORB are not fully identical. But a fair comparison can still be made by taking averages of all the operations tested for each ORB, mitigating this effect.

4. Experimental Results and Analysis

4.1 Overview

Table 2 summarizes the results for the ORBs under test. The total number of test cases and the total number of operations tested are given. For example, for the omniORB 2.8 Solaris build, there are 6999 test cases launched for 22 operations, within which one operation (`create_list()`) exhibits thread hangs, 17 trigger thread aborts, and one operation (`CORBA::string_alloc()`) raises unknown exceptions.

We have observed another failure mode while testing the ORBs, other than the common robustness failures listed in Table 1. While testing operation `create_list()` for omniORB 2.8 and omniORB 3.0 on Sun Solaris platform, we have found a *libthread panic* failure. The failure is denoted by a * in Table 2. This failure cannot be isolated to one test case because it is related to resource problems; however, it does happen each time `create_list()` is tested. One possible explanation is that when the testing thread times out and is killed when a

Table 2. Overview of ORB robustness testing results.

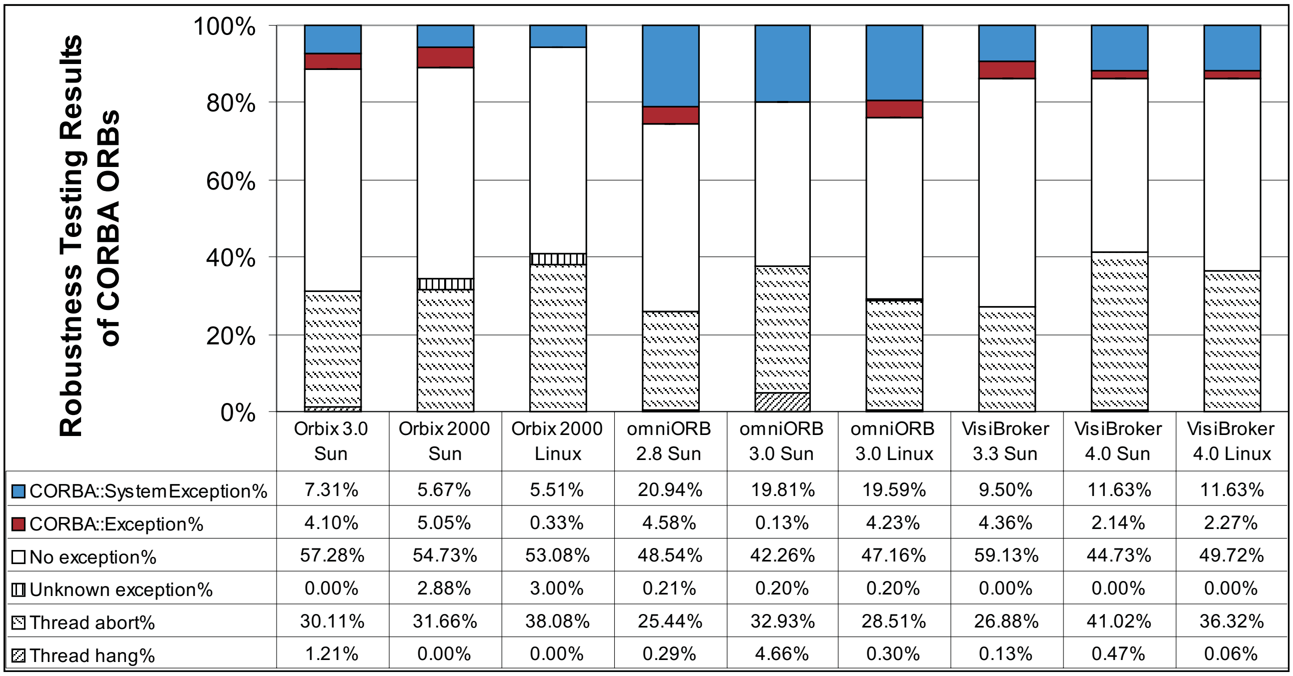
* libthread panic observed

Product	Version	Total test cases	Total functions tested	Number(#) of functions exhibiting each type of failure					
				Solaris 5.6			RedHat Linux kernel version 2.2.14-5.0smp		
				# of functions with thread hangs	# of functions with thread aborts	# of functions with unknown exceptions	# of functions with thread hangs	# of functions with thread aborts	# of functions with unknown exceptions
Orbix	3.0.1	5361	22	6	16	0	N/A	N/A	N/A
omniORB	2.8 *	6999	22	1	17	1	N/A	N/A	N/A
VisiBroker	3.3	6581	21	1	17	0	N/A	N/A	N/A
Orbix	2000	3219	17	0	14	1	0	14	1
omniORB	3.0 *	6581	21	2	17	1	1	17	1
VisiBroker	4.0	7023	23	7	20	0	1	21	0

thread-hang failure happens, *libthread* has a resource leakage that eventually leads to panic. This failure may be unrelated to the ORB under test and is not counted in the final results.

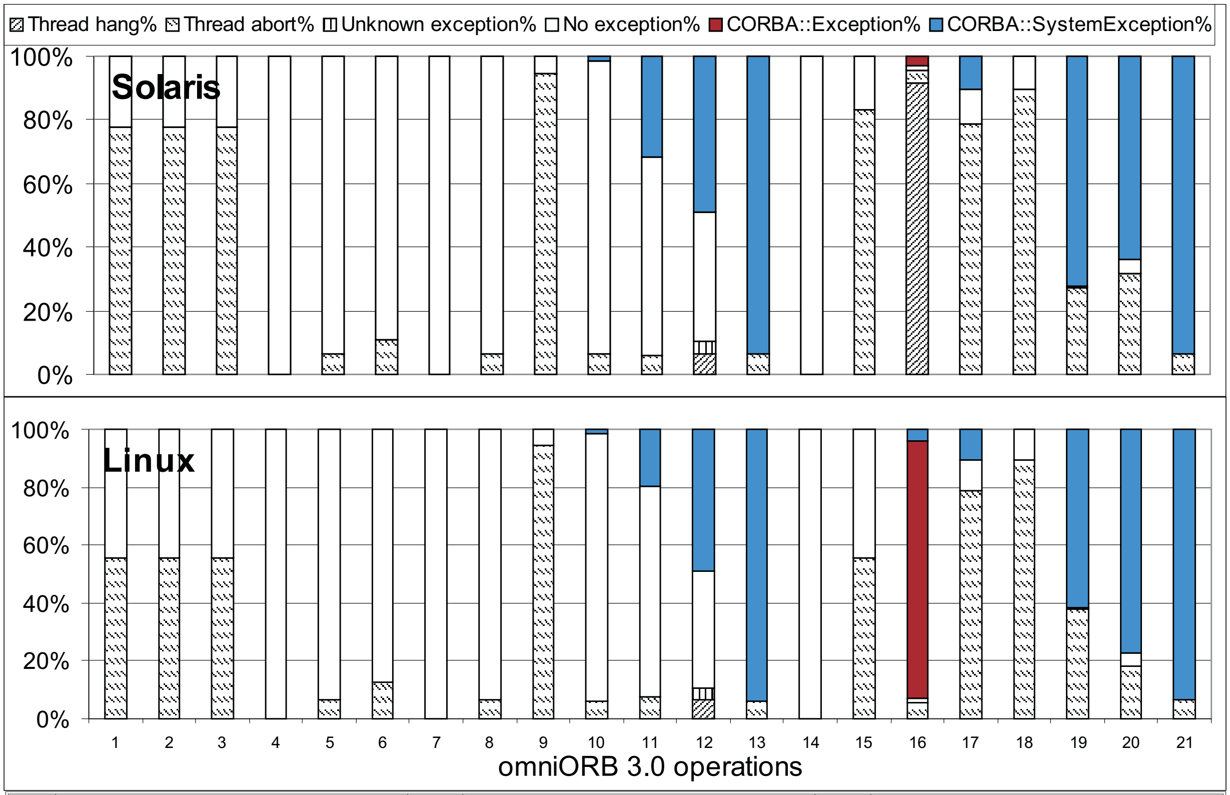
4.2 Average percentage of failures

Results are analyzed using a straight average across all the operations as a comparison metric. Figure 5 shows the robustness testing results for the ORBs we have tested. Each bar represents an ORB implementation, with the average percentage of robustness failures (including thread aborts, thread hangs and unknown exceptions) shown at the bottom, and the average percentage of robust behaviors (including `CORBA::SystemException`, `CORBA::Exception` and no-exception responses) shown at the top. Figure 5 shows that all ORB implementations studied have a high thread-abort percentage under the current test set, ranging from 25.44% for omniORB 2.8 Solaris build to 41.02% for VisiBroker 4.0 Solaris build. Thread-hang failures are less common and usually concentrate in only a few operations such as `create_list()`. Orbix 2000 has more unknown exceptions than



	Orbix 3.0 Sun	Orbix 2000 Sun	Orbix 2000 Linux	omniORB 2.8 Sun	omniORB 3.0 Sun	omniORB 3.0 Linux	VisiBroker 3.3 Sun	VisiBroker 4.0 Sun	VisiBroker 4.0 Linux
🟦 CORBA::SystemException%	7.31%	5.67%	5.51%	20.94%	19.81%	19.59%	9.50%	11.63%	11.63%
🟥 CORBA::Exception%	4.10%	5.05%	0.33%	4.58%	0.13%	4.23%	4.36%	2.14%	2.27%
□ No exception%	57.28%	54.73%	53.08%	48.54%	42.26%	47.16%	59.13%	44.73%	49.72%
▥ Unknown exception%	0.00%	2.88%	3.00%	0.21%	0.20%	0.20%	0.00%	0.00%	0.00%
▧ Thread abort%	30.11%	31.66%	38.08%	25.44%	32.93%	28.51%	26.88%	41.02%	36.32%
▨ Thread hang%	1.21%	0.00%	0.00%	0.29%	4.66%	0.30%	0.13%	0.47%	0.06%

Figure 5. Robustness testing results for 2 versions of 3 ORB products on 2 operating system platforms for a collection of CORBA 2.1 standard operations. Each bar represents the average percentage of different response categories of approximately 22 functions tested using 3000 to 7000 test cases.



1	CORBA::Object::_duplicate	8	_is_a	15	object_to_string
2	CORBA::is_nil	9	_is_equivalent	16	resolve_initial_references
3	CORBA::release	10	add_item	17	send_multiple_requests_deferred
4	CORBA::string_alloc	11	create_child	18	send_multiple_requests_oneway
5	CORBA::string_dup	12	create_list	19	set_one_value
6	CORBA::string_free	13	delete_values	20	set_values
7	_hash	14	get_default_context	21	string_to_object

Figure 6. OmniORB 3.0 robustness testing results for selected CORBA functions on Solaris and Linux platforms. Each bar represents one CORBA operation whose name is listed in the table.

other ORB implementations but is free of thread hangs. No unknown exceptions are found in the VisiBroker group.

The omniORB group demonstrates a higher percentage of CORBA exceptions, indicating a better exception handling scheme. OmniORB 2.8 has the highest exception percentage of over 25%.

Figure 6 gives a snapshot of the testing results for omniORB 3.0 on Solaris and Linux platforms, where we can see that most operations have the same failure profile, with only a few exceptions. Other than the *libthread panic* failure observed on Solaris, the most different testing result was found for operation `resolve_ini-`

tial_references()- a significant percentage of thread hangs on Solaris but only a small percentage of thread-abort failures on Linux.

The operations that have a failure rate higher than 50% are follows: _duplicate(),CORBA::is_nil(), CORBA::release(), _is_equivalent(), object_to_string(), send_multiple_requests_deferred(), and send_multiple_requrests_oneway(). Among these 7 operations, the first five take CORBA::Object as one of the parameters. The last two takes RequestSeq as the parameter value, which is a sequence of object references. This indicates that CORBA object and pseudo-object data type implementations have more severe robustness vulnerabilities than other data types for omniORB implementations.

The results are not marked enough to tell whether and how much operating systems factor into ORB robustness, although there are differences between the same ORB implementation build on the Solaris platform and the Linux platform. For Orbix 2000, the Solaris build is more robust than the Linux version, while for omniORB 3.0 and VisiBroker 4.0, the results are the opposite.

The results show a noticeable, sometimes significant, increase in average percentage of robustness failures from the older version of the product to the new version. Similar phenomena have also been observed in previous Ballista testing results on the HLA RTI simulation backplane [3]. In POSIX testing [10], two operating systems have an increase in robustness failure rate going from lower versions to higher versions, and three operating systems behave just the opposite. One possible explanation is that if a version change of a software product is a result of adding new features and capability, it is likely that the newer version will have more robustness failures since the added new functionality is not yet stable enough and may introduce more robustness failures. If a version change is a result of quality improvement, it is likely that the newer version is more robust. In the case of CORBA ORBs, a version increase is most likely a result of adding new features, because the fast-evolvement and rapid updates of the CORBA standard.

4.4 Robustness failure protection for object reference data types

The testing results in previous sections have not shown us a promising picture of ORB implementation robustness. This section provides some simple but effective measures for ORB robustness improvement and reports on initial experiments on hardening some ORB operations to improve exception-handling robustness.

We propose a probe technique that is flexible and can be easily implemented by ORB vendors to protect a large class of robustness failures, especially those failures caused by invalid and un-initialized object and pseudo-object references. A probe function is a function that can be used to determine the validity of a parameter value. We argue that the probe function must be sensitive, non-intrusive and robust. The probe function should be designed to be as sensitive as possible so that it can discern an invalid object or pseudo-object reference value from a valid one. It must be non-intrusive so that the parameter value as well as the state of the program remains unchanged after the check. It must be robust so that no extra robustness failures are introduced by the probe function itself. The probe function should also be as lightweight as possible to minimize performance overhead; however, there may be a trade-off between sensitivity and performance.

We have found that omniORB 2.8 and omniORB 3.0 actually provides the necessary basis for a primitive probe technique. In omniORB 2.8 and 3.0, each object and pseudo-object data type is assigned a sequence number named `PR_magic`. This “magic” number is unique for each object and pseudo-object data type and serves as an identity mark. When an instance of an object or pseudo-object data type has been correctly set up, the constructor initializes a member variable `pd_magic` to contain the correct `PR_magic` value specific to this data type, its “identity mark”. This variable is set to “invalid” by the destructor when the reference is freed. Therefore, a valid reference which is properly initialized will contain a correct magic number set up in the variable `pd_magic` in its lifetime. If `pd_magic` does not contain the correct value, the reference must be invalid (this ensures that no false positives will happen). An invalid object reference or a pseudo-object reference cannot accidentally have a correct `pd_magic` value set up, unless deliberately sabotaged (this prevents true negatives from happening, which are mainly Silent failures). Therefore, we can detect invalid and un-initialized references to objects and pseudo-objects by checking whether `pd_magic` contains the right `PR_magic` value at run time. A static member function `PR_is_valid()` is defined in omniORB for each object and pseudo-object data type to do this checking.

However, the above functionality provided by omniORB is not robust enough to serve as a probe function for our purposes. For many invalid object references that are in our test set, the `PR_is_valid()` check will trigger a robustness failure instead of returning false. Also, NULL object references cannot be protected by `PR_is_valid()` checking.

We have taken several steps to make a sensitive, robust and non-intrusive probe functionality by refining the `PR_is_valid()` checking method. The current sensitivity of `PR_is_valid()` is largely acceptable, except that we have added `CORBA::is_nil()`, the specialized NULL object reference checking methods defined in the CORBA standard. We have made the checking procedure more robust by adding necessary signal-handling code. A signal triggered while either `PR_is_valid()` or `CORBA::is_nil()` is accessing the parameter value also indicates that an invalid parameter value is detected. A multi-threaded checking scheme can also be used instead of the signal-handling method, but would probably have higher performance cost without rigorous optimization efforts.

We have conducted some initial experiments to study the effectiveness of this method. From our CORBA 2.1 operation test set, we selected a subgroup of eight operations that take CORBA object references or CORBA pseudo-object references as parameters. A simple protection-code generator was implemented to generate protection code suitable for different parameter types. First, a NULL-checking experiment was conducted. Second, the `PR_is_valid()` checking code was generated and added to the target module. Third, a signal handler was installed. The CORBA operations were tested on omniORB 2.8, Linux platform.

The results in Table 3 show that the protection scheme is effective. All thread aborts, formerly 37.77% of the test cases, were successfully eliminated. Note that without necessary signal-handling mechanisms, `CORBA::is_nil()` and `PR_is_valid()` checking actually introduced additional robustness failures because none of the checking methods had a zero robustness failure rate.

Performance overhead was measured by running the target operation 5,000,000 times, with the probe functionality turned on or off, and calculating the difference of the average execution times. Valid parameter values are

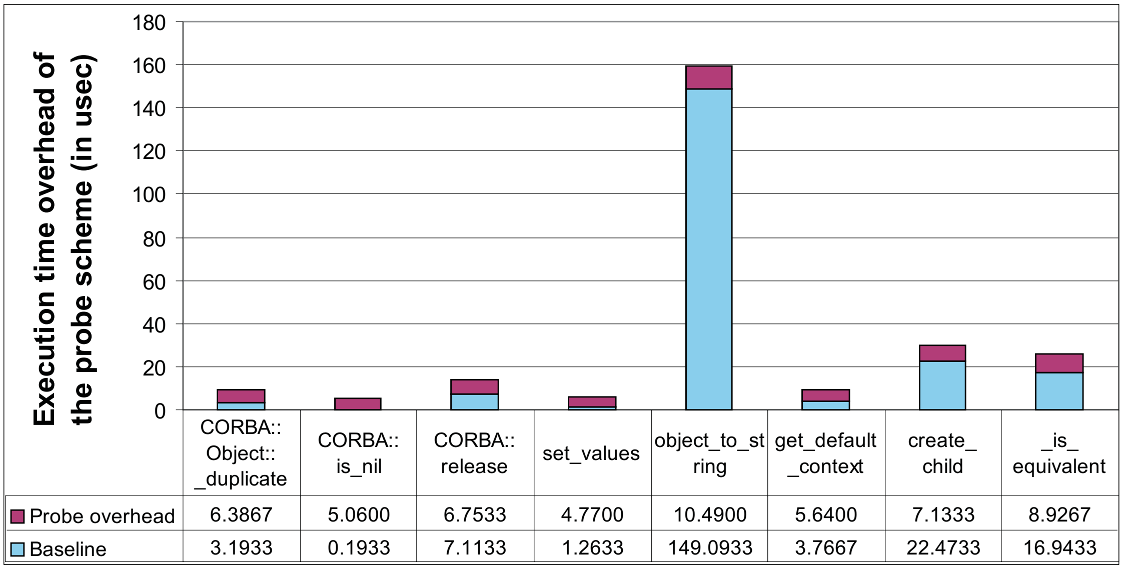
Table 3. Effectiveness of NULL hardening and PR_magic hardening in omniORB 2.8 Linux build.

	Thread-abort percentage
Original failure percentage	37.77%
After `CORBA::is_nil()` checking (alone)	42.45%
After `CORBA::is_nil()+PR_is_valid` checking	41.99%
After adding signal handling	0%

used in this measurement because it is more important to know the performance cost under normal execution situations. Since the protection code is compact and the test program is also very short, all instructions are expected to be resident in cache and this assessment method is optimistic.

The measurement results are shown in Figure 7. We can see that the probe function takes from 4.77 *us* to 10.49 *us* to execute, which is as high as 26 times of the execution time for a simple operation `CORBA::is_nil()` (which contains only one `if` statement) and as low as 7% for a complex operation `object_to_string()`. This can be explained by the fact that the probe functions for all the object data types and pseudo-object data types are very similar.

From this experiment we see that a simple probing technique can protect references to object data types and pseudo-object data types against a class of exceptional values. Similar methods can be standardized and generated



	CORBA:: Object:: _duplicate	CORBA:: is_nil	CORBA:: release	set_values	object_to_st ring	get_default _context	create_ child	_is_ equivalent
🟥 Probe overhead	6.3867	5.0600	6.7533	4.7700	10.4900	5.6400	7.1333	8.9267
🟦 Baseline	3.1933	0.1933	7.1133	1.2633	149.0933	3.7667	22.4733	16.9433

Figure 7. Performance overhead of the probe scheme.

as part of stub code by the `CORBA idl` compiler, so that users can use it to protect their custom data types selectively.

5. Related work

Most previous efforts comparing ORB implementations have focused on measuring and optimizing ORB performance. [4] measures latency and throughput of Orbix and VisiBroker over high-speed ATM networks and identifies major overhead. The study in [14] provides a performance pattern language and a performance measurement object that can be used to extensively test ORB performance. The CORBA comparison project [2] compares omniORB, ORBacus, and Orbix using a rich set of benchmarks, mainly focusing on latency, throughput and scalability. Robustness of these ORBs is also briefly compared in terms of their maximum message size and the number of objects they can handle.

Fault injection on Orbix and DCOM applications [1] studies distributed object behavior under real and simulated failure scenarios. Failures at thread-level, process-level, and machine-level are simulated and injected into the server, and the response of the client is monitored and categorized, which marks the difference from our work. Our approach tries to manifest robustness failures in the CORBA ORB native code using exceptional inputs.

Various efforts have been made to build fault-tolerant CORBA applications, CORBA services and middleware, such as [12][13][20]. The Fault-Tolerant CORBA Standard [16] extends the present CORBA standard for applications requiring high dependability, targeting zero single-point failures. This standard mainly aims at tolerating crash failures using replication and does not address issues of exceptional parameter input handling.

Previous Ballista testing of the High Level Architecture Run-Time Infrastructure (HLA RTI) [3] provides another example of applying the Ballista testing methodology to testing distributed applications. The RTI is a standard distributed simulation system intended to provide completely robust exception handling. This effort extended the Ballista architecture for testing exception-based error reporting models and object-oriented software structures, which paves the path for the work presented here.

The probe technique used in this paper can be classified as a type of signature monitoring technique found as early as in [22] and [21]. Control flow signatures exploit the fact that control flow is mostly sequential by encoding signatures at the end of basic blocks at compile time [22][21][19][23]. Faults can be found at execution time using a simple hardware monitor. [25] uses idle pipeline cycles to reduce the performance overhead of signature checking. [24] combines signature monitoring and encryption to protect hardware faults, software and hardware design faults, and computer viruses. The data structure signature technique in [26] uses a modified compiler to embed a signature in front of data structures to detect data access faults, and performance overhead can be largely minimized by using a special signature monitor that can be added to a standard pipeline processor.

Most previous data-structure signature techniques require either special hardware or compiler modification, which makes it hard for ORB vendors to adopt the techniques, because almost every commercial ORB product relies on commercially available processor/compiler pairs, *e.g.* Intel X86/gcc, Sun Sparc/SunCC, aiming at the broadest customer base. In contrast, the probe technique does not need architecture or compiler support. An ORB can use the probe technique to protect native ORB functions under a robust operating mode and can turn the checking off in normal operation mode where performance is more important. The ORB vendors could also provide the user with the flexibility to selectively use the probe to check against important object and pseudo-object references where robust operation is required.

6. Conclusions

In this paper we have introduced a methodology to test and measure the exception-handling robustness of CORBA ORB implementations using Ballista. We have ported Ballista testing clients to work with the ORBs, and tested two major versions of three ORB implementations on two operating systems for several CORBA 2.1 standard operations. This approach enables us to evaluate the robustness of a specific ORB implementation, to compare different ORB implementations provided by various vendors, and to enhance the robustness of a specific ORB implementation.

We have presented results on the average percentage of failures for the ORBs we have tested. Results show that the ORB implementations generally lack exception-handling robustness. This result suggests that the users must pay close attention and use necessary measures to enhance ORB robustness when building critical applications on CORBA-based systems.

In the CORBA standard, complex data types, such as object references and pseudo-object references, are frequently used as parameters in standard operations. We have observed that complex data types are more likely to have robustness failures. A simple probe technique is used for checking against invalid object references and pseudo-object references. Experimental results done on omniORB 2.8 show that this method is very effective and completely eliminates all robustness failures in our test data set for the eight CORBA operations we tested. This suggests that probe methods can be used to protect object or pseudo-object data types against a large class of exceptional input values.

7. References

[1] P. Emerald Chung, Woei-Jyh Lee, Joanne Shih, Shalini Yajnik and Yennun Huang, "Fault-Injection Experiments for Distributed Objects", Proceedings of the International Symposium on Distributed Objects and Applications, Edinburgh, United Kingdom, September 5 - 7, 1999.

[2] Distributed Systems Research Group, "CORBA Comparison Project", http://nenya.ms.mff.cuni.cz/thegroup/COMP. Accessed June 10, 2000.

[3] Kimberly Fernsler and Philip Koopman, "Robustness Testing of A Distributed Simulation Backplane", Proceedings of ISSRE 99, Boca Raton, Florida, November 1-4, 1999.

[4] Aniruddha Gokhalé and Douglas Schmidt, "Evaluating CORBA Latency and Scalability Over High-Speed ATM Networks", Proceedings of ICDCS 97, Baltimore, MD, May 27-30, 1997.

[5] IEEE Computer Society, "IEEE Standard Glossary of Software Engineering Terminology (IEEE Std610.12-1990)", Dec. 10, 1990.

[6] IEEE Computer Society, "IEEE Standard for Information Technology – Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) – Amendment 1: Real-time Extension [C Language] (IEEE Std 1003.1b-1993)", 1994.

[7] Inprise Corporation, "VisiBroker for C++ Reference, Version 4.0", 2000.

[8] IONA technologies PLC, "Orbix 2000 Programmer's Reference", March 2000.

[9] Philip Koopman & John DeVale, "The Exception Handling Effectiveness of POSIX Operating Systems", IEEE Transactions on Software Engineering, , September 2000.

[10] Philip Koopman and John DeVale, "Comparing the Robustness of POSIX Operating Systems", Proceedings of FTCS 29, Madison, Wisconsin, June 15-18, 1999.

[11] Sai-Lai Lo and David Riddoch, "The omniORB2 version 2.8 User's Guide", AT&T Laboratories Cambridge, July 1, 1999.

[12] Silvano Maffeis, “A Fault-Tolerant CORBA Name Server”, Proceedings of the 1996 IEEE Symposium on Reliable Distributed Systems. Niagara-on-the-Lake, Canada: IEEE, October 1996.

[13] L. Moser, P. Melliar-Smith and P. Narasimhan, “A Fault Tolerance Framework for CORBA”. Proceedings of FTCS 29, Madison, Wisconsin, June 15-18, 1999.

[14] S. Nimmagadda, C. Liyanaarachchi, A. Gopinath, D. Niehaus and A.Kaushal, “Performance Patterns: Automated Scenario Based ORB Performance Evaluation”, Proceedings of COOTS 99, San Diego, California, May 3-7, 1999.

[15] Object Management Group, “The OMG’s site for CORBA and UML success stories”, http://www.corba.org/success.htm. Accessed June 22, 2000.

[16] Object Management Group, “Fault Tolerant CORBA standard”, http://www.omg.org/cgi-bin/doc?orbos/2000-01-19. October, 1999. Accessed June 19, 2000.

[17] Object Management Group, “Space Domain Task Force”, http://www.omg.org/homepages/space/index.htm. Accessed Dec 5, 2000.

[18] Object Management Group, “The Common Object Request Broker: Architecture and Specification”, August, 1997.

[19] N. Saxena and E.McCluskey, “Control-Flow Checking Using Watchdog Assists and Extended-Precision Checksums”, *IEEE Trans. Computers*, vol 39, no. 4, pp. 554-558, Apr. 1990.

[20] J. Schonwalder, S. Garg, Y. Huang, A. P. A. van Moorsel and S. Yajnik, “A Management Interface for Distributed Fault Tolerance CORBA Services”, Proceedings of the IEEE 3rd International Workshop on System Management, Newport, RI, April 1998.

[21] M. Schuette and J. Shen, “Processor Control Flow Monitoring Using Signature Instruction Streams”, *IEEE Trans. Computers*, vol.36, no. 3, pp 264-276, Mar. 1987.

[22] T. Sridhar and S. Thatte, “Concurrent Checking of Program Flow in VLSI Processors”, Proc. *Int’l Test Conf.*, pp. 191-199, 1982.

[23] N. Warter and W. Hwu, “A Software Based Approach to Achieving Optimal Performance for Signature Control Flow Checking,” *Proc. Int’l Symp. Fault-Tolerant Computin*g, pp. 442-449, 1990.

[24] K. Wilken and J. Shen, “Concurrent Error Detection Using Signature Monitoring and encryption”, *Proc. Int’l Working Conf. Dependable Computing for Critical Application*s, pp. 365-384, 1991.

[25] K. Wilken, “An Optimal Graph-Construction approach to Placing Program Signatures for Signature Monitoring,” *IEEE Trans. Computer*s, vol. 42, no. 11, pp. 1,372-1,381, Nov. 1993.

[26] Kent Wilken & Timothy Kong, “Concurrent Detection of Software and Hardware Data-Access Faults”, *IEEE Transactions on Computers*, VOL. 46, No. 4, April 1997.