Automatic Generation of Fault-Tolerant CORBA-Services

Andreas Polze, Janek Schwarz, and Miroslaw Malek

Department of Computer Science
Humboldt University of Berlin
10099 Berlin, Germany
{apolze|schwarz|malek}@informatik.hu-berlin.de

Abstract

The Common Object Request Broker Architecture (CORBA) is the most successful representative of an object-based distributed computing architecture. Although CORBA simplifies the implementation of complex, distributed systems significantly, the support of techniques for reliable, fault-tolerant software, such as group communication protocols or replication is very limited in the state-of-the-art CORBA or even fault-tolerant CORBA.

Any fault tolerance extension for CORBA components needs to trade off data abstraction and encapsulation against implementation specific knowledge about a component's internal timing behavior, resource usage and interaction patterns. These non-functional aspects of a component are crucial for the predictable behavior of fault-tolerance mechanisms. However, in contrast to CORBA's interface definition language (IDL), which describes a component's functional interface, there is no general means to describe a component's non-functional properties.

We present here a generic framework, which makes existing CORBA components fault-tolerant. In adherence with a given, programmer-specified fault model, our framework uses design-time and configuration-time information for automatic distributed, replicated instantiation of components. Furthermore, we propose usage of aspect-oriented programming (AOP) techniques to describe fault-tolerance as a non-functional component property.

We describe the automatic generation of replicated CORBA services based on aspect information and demonstrate service configuration through a graphical user-interface.

Key Words: CORBA, Fault-Tolerance, Replication, Aspect-Oriented Programming

1. Introduction

Distributed computing using commercial-off-the-shelf (COTS) components is appealing because of its cost-effectiveness. A number of standards have evolved at the end of the last century, which ensure interoperability among heterogeneous hardware platforms from different vendors. Object technology has been used to describe interfaces and interaction patterns for such distributed applications. The Common Object Request Broker Architecture (CORBA) is the most successful representative for an object-based distributed computing architecture.

Although CORBA simplifies the implementation of complex, distributed systems significantly, support of techniques for reliable, fault-tolerant software, such as group communication protocols or replication is very limited in available CORBA implementations. We have developed a framework for fault-tolerant CORBA services, which employs consensus protocols and a technique called analytic redundancy to tolerate timing faults such as overrun, programming system faults such as illegal addressing, and semantic faults due to modeling, algorithm design or implementation errors.

Within this paper we discuss the necessary description and configuration steps at design-time, configuration-time and runtime of a fault-tolerant service. We demonstrate how a graphical demonstration application (Mandelbrot set computation) can be extended automatically to tolerate incorrect computation faults and crash faults of processes and processors.

The remainder of the paper is organized as follows: Section 2 presents related work. Section 3 introduces the fault model and assumptions, which serves as the basis for our CORBA framework. Furthermore, Section 3 briefly describes our current framework for automatic generation of fault-tolerant CORBA services and its graphical frontend. Section 4 proposes the use of aspect-techniques to describe a component's non-functional properties. Directions for the future work and our conclusions are given in Section 6.

2. Related Work

The idea of providing fault tolerance as additional feature to CORBA implementations has been lately the focus of several research activities. With the request for proposal for a Fault-tolerant CORBA Using Entity Redundancy [1] issued in April 1998, OMG is seeking to incorporate existing approaches for software fault tolerance into future versions of CORBA. However, in contrast to our approach, which focuses on automatic generation and replication of replicated components, most related work implements fault-tolerant behavior by extending the underlying CORBA object request broker.

Electra [2] is a CORBA ORB-implementation for reliable, distributed services. Electra extends the CORBA specification and provides group communication mechanisms, reliable multicasts, and object replication. The Electra-ORB uses services from the underlying ISIS [3] and HORUS [4] systems.

ORBIX+ISIS is an extension of the commercial ORBIX [5] ORB implementation which introduces concepts like object groups and group communication based on the ISIS [3] toolkit. Again, the CORBA standard has been extended to provide for introduction of fault tolerance measures. The programmer has to use special coding techniques to be able to use the fault tolerance features of ORBIX+ISIS.

Phoinix [6] allows for implementation of server objects following the primary/backup approach for fault tolerance. In contrast to changing the ORB, Phoinix defines a new service as part of the Object Management Architecture (OMA). Using this service, a primary's object state can be periodically transferred into the corresponding backup object. Clients communicate solely with the primary object and have to detect failures themselves. Extended skeletons, stubs and libraries are provided to make those fault-tolerant services accessible.

Distributed Object-Oriented Reliable Service (DOORS) [7] implements fault-tolerant behavior based on CORBA objects. DOORS is designed to tolerate crash faults of objects and processors (nodes). It defines object groups via a ReplicaManager-interface. The ReplicaManager identifies a primary inside an object group which handles all client requests. In order to use the service, objects additionally need to support a WatchDog-interface for fault-detection. DOORS supports object migration and checkpointing. Multiple object versions and voting are not supported by DOORS.

3. A Framework for fault-tolerant CORBA Services

3.1 Fault-Model and Assumptions

Fault-tolerance denotes the ability to provide a certain service even in the presence of faults. Assumptions about possible failures as well as expected system load are required in order to give fault-tolerance guarantees. Many sophisticated fault models have been developed at lower levels of computer systems (e.g., gate level or functional level). In context of component-based systems these models are difficult to use. Therefore, we establish our fault model at the component level - the smallest entity, which may fail in our framework is a whole CORBA component. Our work assumes the fault model depicted in Figure 2, which was introduced in [8] and extended in [9]. This fault model distinguishes the following fault classes:

- Crash Fault: A processing element (PE) loses its internal state or halts. The processor is silent during the fault
- Omission Fault: A PE fails to meet a deadline or to begin a task. This fault class includes the crash fault
- Timing or Performance Fault: A PE completes an assignment before or after its specified time frame or never. This fault class includes the omission fault
- Incorrect Computation Fault: A PE fails to produce a correct output in response to a correct input. This fault class includes the timing fault.
- Authenticated Byzantine Fault: A PE behaves in an arbitrary or malicious manner, but is unable to imperceptibly change an authenticated message. This class includes the incorrect computation fault.
- Byzantine Fault: Every possible fault. This class includes the authenticated Byzantine fault.

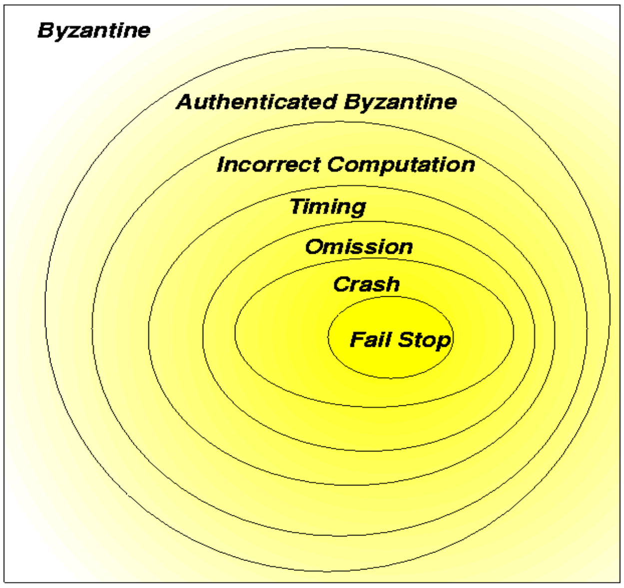

Figure 1: The fault model

To handle faults of different fault classes, different protocols exist, which have to ensure consensus – a consistent view onto system state - among the (non-faulty) processors. Within our framework, we distinguish crash faults (of components of processors) and incorrect computation faults. Due to limitations inherent in CORBA communication (IIOP), the system maps timing and omission faults onto crash faults and stops a faulty CORBA component. Our system does not provide detection mechanisms for Byzantine faults.

Within our approach, we distinguish between design-time (programming) and runtime faults. Multiversion programming [10] and analytic redundancy [11] paired with a consensus protocol (voting) are applied to cope with design-time faults. In contrast, runtime faults (e.g.; lacking resources, processor crashes) are handled on the basis of component (object) replication.

In order to be able to tolerate runtime failures of our system, we assume independent behavior of different computer nodes. Furthermore, we assume independent execution of different versions of a component.

3.2 Component Model for a Fault-Tolerant Service

Within our framework, a fault-tolerant service is implemented by an interface object (proxy) and (replicated) core services. The interface object implements fault-tolerance mechanisms such as group communication protocols, leadership election, and voting. Additionally it initiates state synchronization among core service objects. Client objects communicate solely with the interface object.

The core service objects can be seen as standard CORBA objects. In addition to their service-specific interface, they may implement an interface for state synchronization which allows them to use certain replication techniques. Our toolkit for service configuration as described in Section 3.3 chooses the appropriate fault-tolerance strategy based on the support of the synchronization interface for given core service objects.

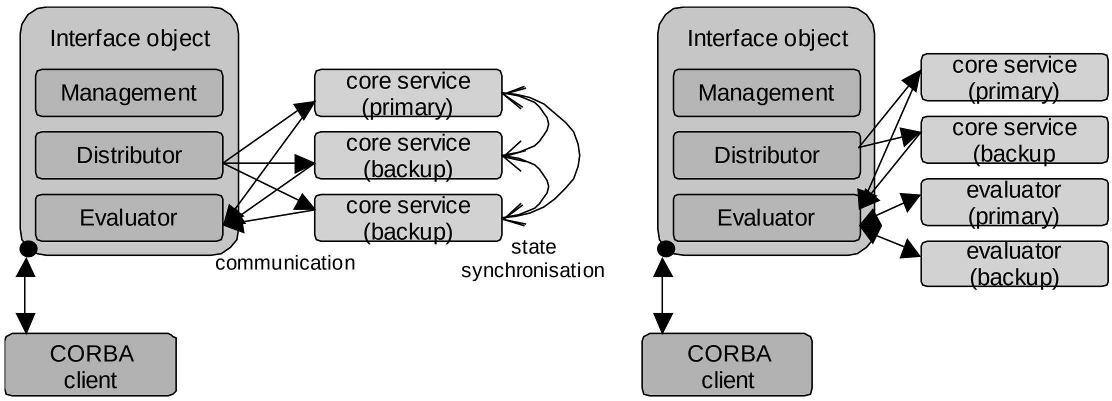

Figure 2: Communication structure for two fault-tolerant services

Figure 2 shows two possible interaction patterns for fault-tolerant services. We have depicted on the left a primary/backup replication scenario where core service objects periodically exchange state information. Our framework supports hot, warm and cold replication. Hot replication is implemented by truly parallel operation of all replicas: every request is handled by each replica. Warm replication is implemented by having one primary object handle all client requests and periodically transferring its state to other objects. Cold replication finally is implemented as checkpointing to stable storage. Backup objects are started only if the primary fails – they are initialized according to the most recent checkpoint data.

The right side of Figure 2 depicts a scenario where the interface object's evaluator is supported by two service-specific evaluator objects. These objects may implement a different version of the service (algorithm) and give hints about the validity of a result to the interface object's evaluator. Usage of service-specific evaluators allows for detection of incorrect computation faults (in addition to a component's crash fault).

The scenarios depicted in Figure 2 show the interface object as a single-point-of-failure. A crash of the interface object will indeed stop the entire fault-tolerant service. Therefore, we assume that the interface object is running on an ultra-reliable system and has been tested and verified. Alternatively, the interface object could be implemented inside a library, which is linked directly to the client object(s). In our case, this would be sufficient to make the client object the single-point-of-failures in the system.

Moving the interface object's functionality into the CORBA object request broker (ORB) would be another option. It is difficult to avoid the ORB as single-point-of-failure in a CORBA-based system anyway and we would like to leave this exercise to the ORB vendors. Our focus here is rather on a generic, configurable framework, which allows the easy enhancement of existing CORBA services with fault-tolerance properties.

3.3 A Toolkit for Component description and Service configuration

Within our framework, a set of fault-tolerance requirements determines the instantiation and configuration steps necessary for those replicated objects implementing a fault-tolerant service. A graphical user interface allows the component programmer to choose fault-tolerance techniques supported by a particular service implementation from a list of options. At the configuration time, the system administrator may use a similar graphical tool to specify properties of the execution environment (number of nodes, etc.) and start the instantiation process for a number of replicated core service objects (based on a previously chosen fault model).

At the configuration time the following requirements can be specified as input to our configuration tool:

- Classes of tolerable faults (crash faults, incorrect computation faults)
- Number of tolerable faults (over runtime of a component)
- Tolerance of programming- or execution-time faults
  (replication vs. multi-version execution)

Our fault model assumes, that all faults are permanent. A faulty component therefore is rendered unusable. The model maps communication failures onto component failures and assumes independence of different fault-tolerant services.

In addition to the fault model, the configuration process takes into account a number of optimization criteria, such as resource usage in the fault-free case, response time in the fault-free case and recovery overhead in case of faults.

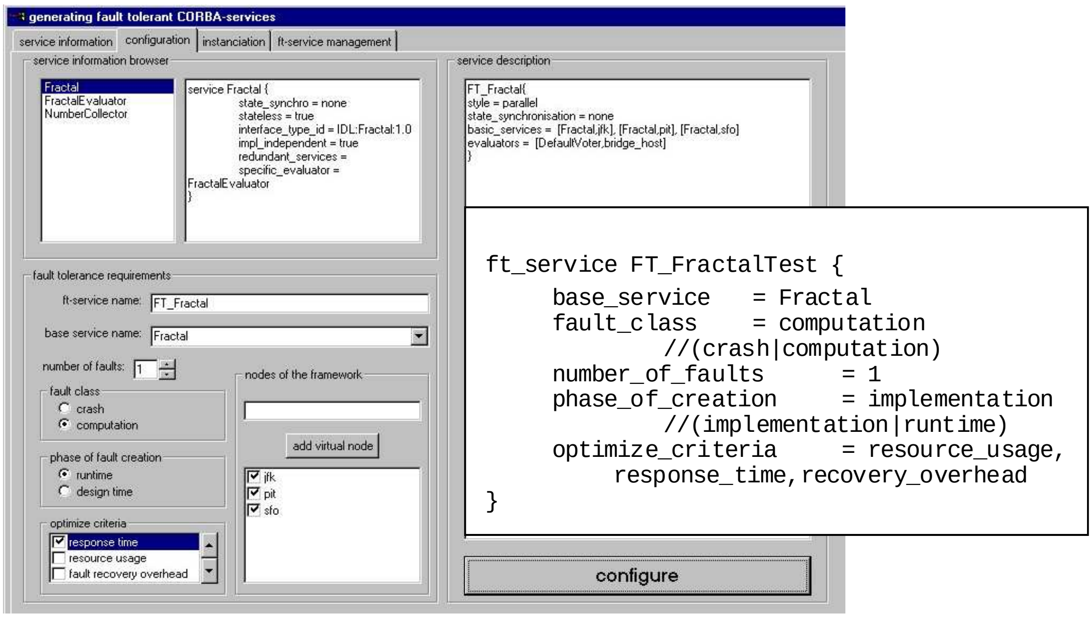

Figure 3: Configuration of a fault-tolerant service

Figure 3 depicts the interface to the configuration tool as well as the service description in a textual representation. Our example shows the FT_FractalTest-service (computation of Mandelbrot sets), which is configured to achieve fault-tolerance using primary/backup replication without state synchronization based on functional redundancy (multi-version). The service may tolerate a single computation fault and uses the Fractal-service as its core service.

The screendump depicted in Figure 3 further shows, that three different computer nodes are available for execution of the core service (namely *jfk, pit, sfo*). By checking the appropriate boxes the service administrator has indicated that he/she wants to set up the service such, that incorrect computation faults are covered. Furthermore, he/she requests protection against malfunction at runtime (computer crashes) but not against programming errors.

Given this information, the configuration tool creates enough instances of the Fractal-core service objects distributed over the hosts *jfk, pit, sfo.* Clients may access the functionality of the Fractal-service via the FT_FractalTest interface which is implemented by the interface object (proxy) and forwards requests to the Fractal-core service objects. Results are returned back to the client after passing the interface object's evaluator.

With the FT_FractalTest service we have just demonstrated how a fault-tolerant CORBA service can be generated automatically from a given sequential service implementation.

4. Description of a Component's non-functional Aspects

In the process of implementing a generic framework for fault-tolerant execution of CORBA components, we had to trade off data abstraction and encapsulation against implementation specific knowledge about a component's timing behavior, resource usage and interaction patterns. These non-functional aspects of a component are crucial for the predictable behavior of failure detection and voting mechanisms. However, in contrast to CORBA's interface definition language (IDL), which describes a component's functional interface, there is no general means to describe a component's non-functional properties.

We currently provide a set of tools with a graphical user-interface for specification of a component's fault assumptions and its execution environment. However, we plan to use the technique of aspect-oriented programming (AOP) [12] to formally describe these properties. In contrast to a component, which can be cleanly encapsulated in a generalized procedure (i.e., object, method, procedure, API), an *aspect* describes a system property, which cannot be expressed in a standard construct used in object-oriented programming. Aspects tend not to be units of the system's functional decomposition, but rather to be properties that affect performance or semantics of components in a systematic way. Examples of aspects include memory access patterns and synchronization of concurrent objects.

In context of fault-tolerant CORBA services, two different classes of aspects can be distinguished. Intra-framework aspects describe the behavior and interaction pattern of components (replicated objects) inside the framework. Inter-framework aspects describe interactions between our system and the outside world.

Within context of this paper we want to enumerate intra-framework aspects, which need to be taken into account for fault-tolerant CORBA services:

- The *fault-tolerance aspect* describes FT-techniques used by a component. It needs to be specified whether a component has internal state and whether a synchronization technique, such as checkpointing is being used. Also, criteria, which indicate that a freshly started component has reached, steady state needs to be given.
- The *timing aspect* describes a component's characteristics, such as timing behavior as expressed by average case and worst-case execution times, periods, dependencies, and resource requirements (memory, disk I/O) for all entities of the component's functional interface. A component's timing behavior has a direct impact on applicable fault-detection mechanisms.
- The *consensus aspect* as implemented currently by (application specific) voter objects describes which algorithm should be used to judge a component's internal state and whether the component functions correctly. This aspect also gives criteria for the replacement of components.

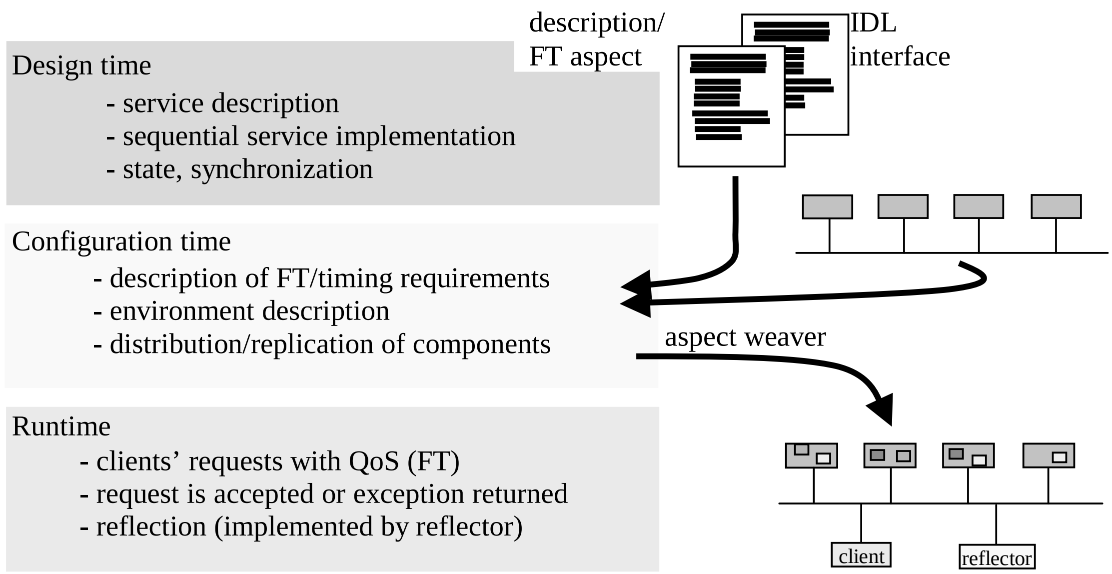

Figure 4: Description of a component's non-functional aspects

As depicted in Figure 4, aspects are taken into consideration at three different levels in the software development process for a component-based system: The programmer specifies aspects like fault-tolerance technique used (FT) or worst case execution time (RT; in terms of instructions) *at design time*. At *configuration time*, design-time information and aspect-information specific to the current environment (number of nodes, processing power, security characteristics) are taken into account to distribute components over the network and configure the system. This step can be automated using a tool such as an *aspect weaver*. Finally, at *runtime* the outcome of the previous steps is stored within an additional component – the *reflector*.

The *reflector* is generated automatically and stores configuration information. It might be used for online reconfiguration (replacement) as a result of a client's request for specific quality-of-service (such as 2-out-of-3 majority voting or distributed execution in order to avoid common-mode failures). We are currently working on definition of an XML-based aspect language and aspect weaver for our generic framework for fault-tolerant CORBA services.

5. Conclusions

We have presented a generic framework, which extends existing CORBA components with fault-tolerant behavior. In adherence with a given, programmer-specified fault model, our framework uses design-time and configuration-time information for automatic distributed, replicated instantiation of components. Our approach employs object replication to tolerate crash faults and uses consensus protocols, such as voting and multi-version programming to tolerate computation faults.

Within our framework, a set of fault-tolerance requirements determines the instantiation and configuration steps necessary for those replicated objects implementing a fault-tolerant service. A graphical user interface allows the component programmer to choose fault-tolerance techniques supported by a particular service implementation from a list of options. At configuration time, the system administrator may use a similar graphical tool to specify properties of the execution environment (number of nodes, etc.) and start the instantiation process for a number of replicated core service objects (based on a previously chosen fault model).

Non-functional aspects of a component, such as internal timing behavior, resource usage and interaction patterns are crucial for the predictable behavior of fault-tolerance mechanisms. However, in contrast to CORBA's interface definition language (IDL), which describes a component's functional interface, there is no general means to describe a component's non-functional properties. Aspect-oriented programming techniques are a promising way to describe non-functional aspects of CORBA components [12]. We are currently studying aspect-techniques for component description in a fault-tolerant distributed tele-laboratory case study.

Acknowledgements

This work has been partially funded by the German Research Network (DFN). We want to thank our students Mario Schröer and Oliver Freitag, who implemented the framework for fault-tolerant CORBA-based services using ORBacus [13] on the Windows NT 4.0 operating system.

References

[1] OMG; "Fault tolerant CORBA Using Entity Redundancy RfP"; OMG Document orbos/98-04-01, at http://www.omg.org, 1998.

[2] S.Maffeis; "A Flexible System Design to Support Object Groups and Object-Oriented Distributed Programming"; in Proceedings of ECOOP'93, Lecture Notes in Computer Science 791, 1994.

[3] K.P.Birman; "The Process Group Approach for Reliable Distributed Computing"; Communications of the ACM, pp.37-53, Vol. 36, No.12, December 1993.

[4] R.van Renesse, K.P.Birman; "Fault-Tolerant Programming using Process Groups"; in F.Brazier, D.Jones (Eds.) "Distributed Open Systems", Computer Society Press, 1994.

[5] see multiple articles at http://www.iona.ie

[6] Y.-S.Chang, D.Liang, W.Lo, G.-W.Sheu, S.-M.Yuan; "A Fault-Tolerant Object Service on CORBA"; Proceedings of International Conference on Distributed Computing Systems (ICDCS'97), Baltimore, May 1997.

[7] P.E.Chung, Y.Huang, and S.Yajnik; “DOORS: Providing fault-tolerance to CORBA Applications”; in Proceedings of Middleware 98, 1998.

[8] F.Christian, H.Aghili, R.Strong, and D.Dolev; “Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement”; IBM, Technical Report RJ 5244 (54244), 1985.

[9] L.Laranjeira, M.Malek, and R.Jenevin; “Nest: A Nested Predicate Scheme for Fault Tolerance”; in IEEE Transactions on Computers, Volume 39, 1993.

[10] Avizienis; “The n-version approach to fault-tolerant software”; IEEE Transactions on Software Engineering, pp. 1491-1501, Dec. 1985.

[11] L.Sha, R.Rajkumar, M.Gagliardi; "Evolving Dependable Real-Time Systems"; in Proceedings of 1996 IEEE Aerospace Applications Conference, IEEE Inc., February 1996, ISBN 0-7803-3196-6.

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin; “Aspect-Oriented Programming”
In proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.

[13] Object Oriented Concepts, Inc; “ORBacus for C++ and Java”, Version 3.1, http://www.ooc.com/ob, 1998.

[14] Andreas Polze, Lui Sha; “Composite Objects: Real-Time Programming with CORBA”; in Proceedings of 24th Euromicro Conference, Network Computing Workshop, Vol.II, pp.: 997-1004, ISBN 0-8186-8646-4, Vaesteras, Sweden, August 25-27, 1998.