Design of a Resource Manager for Fault-Tolerant CORBA

Aad P. A. van Moorsel
Shalini Yajnik

Bell Laboratories Research, Lucent Technologies
600 Mountain Ave., Murray Hill, New Jersey 07974, U.S.A.
aad,shalini@lucent.com, http://www.bell-labs.com/~aad,shalini

Abstract

Steadily, the CORBA fault tolerance service is moving forward on the OMG standardization track. Although not part of the standardization effort, one of the issues that is expected to be under consideration in any future implementation of the fault tolerance service is a 'resource manager.' This decision-making unit determines where to place replicas, what replication style to choose, how often to issue heartbeats, etc. In this paper, we discuss the design of a resource manager. The issue we are most concerned with is the anticipated diversity of platforms and environments in which the fault tolerance service may be used. We therefore argue for an extensible design, which is suited for limited resource managers as well as for full-blown decision-making units in a more extensive management solution. The prominent features of the resource manager design are pluggable decision-making algorithms and data collectors.

1. The CORBA Fault Tolerance Service

CORBA (Common Object Request Broker Architecture) is an architecture for distributed object computing standardized by the Object Management Group (OMG). In April 1998, OMG initiated an effort to standardize Fault Tolerant (FT) CORBA [7]. A consortium of companies is currently working towards this standard [8]. A standard for fault tolerant CORBA will use replication of services to provide transparency from hardware and software-induced service faults. Although it is not expected to be a part of the standards, future FT CORBA implementations must decide whether resource management functionality is necessary. The resource manager provides 'intelligence' at instances where decisions about configuration and reconfiguration of the system need to be made.

Let us first introduce the major features of a fault tolerance service. Since the Fault Tolerant CORBA standardization effort is not yet complete, we will continue our discussion in the context of a fault tolerance service called DOORS (Distributed Object Oriented Reliable Service) that has been developed at Bell Laboratories [1]. Recent OMG documents [8], however, suggest that the following discussion of DOORS will also prove relevant for the future FT CORBA standard.

DOORS provides fault tolerance through replication of objects. It also supports failure detection and transparent recovery. It consists of two main kinds of components, the Replication Manager and the Fault Detectors. There is a single logical Replication Manager in the system. Given the reliability/availability requirements of an application, the Replication Manager is responsible for creating and maintaining a fixed number of replicas of the application. In order to achieve this under failure conditions, it needs to detect failed objects and perform transparent recovery of the failed objects. The Fault Detector modules are responsible for detecting failures of application objects and reporting them to the Replication Manager. There can be several different kinds of Fault Detectors in the system, e.g. object level, process level, host level detectors. Object-level detectors detect failure by monitoring objects using heartbeat or polling-based mechanisms.

A Replication Manager must make some decisions that may affect the performance and reliability of the system. At configuration time, given the reliability and availability requirements of an application, a system designer must come up with

parameters like replication style (Cold, Warm and Hot), degree of replication, monitoring style, and, frequency of monitoring. During registration, these parameters are given as inputs to the Replication Manager, which subsequently decides where to create the replicas. When a failure occurs, the Replication Manager must reconfigure the system to maintain the degree of replication. During this reconfiguration, it must again decide the location of the new replica. DOORS currently expects the system designer to supply a list of locations at registration time. At both configuration and reconfiguration time, it uses a round-robin approach to select the replica locations from this list.

In the following sections, we show that the above simplistic resource management algorithm used by Replication Manager can degrade the QoS characteristics of the system. We also recommend a design where a generic Resource Manager is attached to the Replication Manager and aids the Replication Manager in making intelligent 'QoS-aware' decisions.

2. Design

Figure 1 illustrates our main design, which follows the design of the QoS Query Service [11]. We distinguish two designs: the minimalist base design and the extended general management solution. In the base design, only basic data is fed from the replication manager into the resource manager, and the resource manager only responds to queries from the FT CORBA replication manager. The replication manager queries the resource manager for decisions at failure instances, at initialization and at registration of a service with FT CORBA. As we will argue, the environment in which the fault tolerance service will run is unknown, and the resource manager therefore uses pluggable decision-making algorithms, thus making it possible to make decisions for any desired QoS objective.

For the extended management solution, we envision various additional features (depicted in Figure 1 using the word 'additional'). More advanced algorithms may require additional data collectors, which can be plugged in at will. These data collectors may collect data from other CORBA applications and services, or from other system elements (through the use of SNMP monitoring and the like). In Figure 1, we list some additional management features (differentiating between different platform clients, introduction of a QoS definition language), and many more may be thought of. These additional features bring the design into the realm of generic management solutions (e.g., [6] and [10]), but since the resource manager allows pluggable algorithms and data collectors, it can 'grow' into a decision-making unit suitable for this more generic framework. An interesting option is to initiate management actions (such as object migration) based on various alarms (instead of just failures). In Figure 1, the arrow from the resource manager to the replication manager denotes such actions. Note, however, that FT CORBA does not target extensive management, and therefore its interface definitions must be extended in a proprietary manner to support such actions.

So, we propose an extensible design, based on the QoS Query Service we developed in [11]. The rest of this section further motivates this design. First, we show in Section 2.1 that a resource manager is required in FT CORBA, and then we discuss further the need for an extensible design (Section 2.2). Finally, in Section 2.3, we discuss various more specific issues, such as data collectors, system model, QoS queries and 'alarm' events.

2.1. The Resource Manager in the CORBA Fault Tolerance Service

In one sentence, the function of the resource manager is to *make run-time, QoS-aware, decisions about parameterization and configuration of the fault tolerance service and its clients*. Close inspection of this description, however, reveals many new questions about the actual scope of the definition, and its consequence for the design and implementation of a resource manager. In particular, the term 'QoS-aware' does not say *what* QoS metrics we are interested in, nor *whose* quality of service we try to improve or optimize. However, for the sake of argument, let us first assume that there are some well-defined QoS objectives (related to either services or clients); why, then, do we need run-time decision making, and why do we use a separate unit for this?

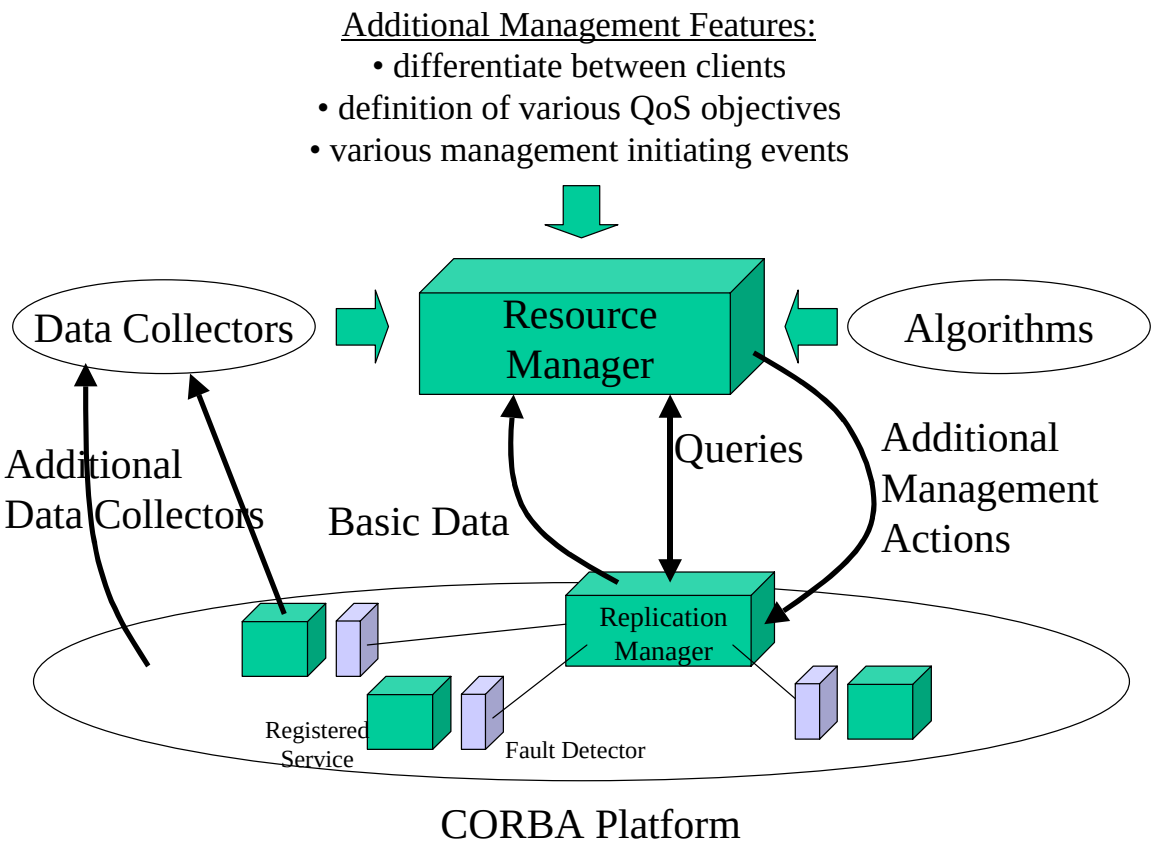

Figure 1. Resource Manager

System Pollution. In case there is any doubt about the need of at least a certain level of 'intelligence' in the fault tolerance service, we quickly 'prove' the existence of, what we call, 'system pollution.' Automated failure recovery leads the systems through various states, moving processes to different machines, etc. Some of these states may be undesirable, or 'polluted' (like a system state in which all processes have been moved to a single CPU), but may nevertheless be reached if there is no control over how the replication manager places the replicas upon initial configuration or failure reconfiguration. In fact, if one accepts the state-transition model as a realistic representation of the system, one can easily prove that, in time, the system will, with probabilistic certainty, end up in a polluted state. For instance, in Markov process theory, only very mild conditions on the recurrence properties of the non-polluted states are required to prove the eventual arrival in one of the polluted states. In other words, if it can go wrong, it will go wrong, eventually.

Separate Decision-Making Unit. Somewhat formally, we thus have established the need for control of the execution of initial configuration and recovery mechanisms. One can establish similar results for intricate QoS metrics, in place of the system pollution notion. Given this need for control/intelligence, we argue that separate decision-making unit (that is, a software component not too tightly integrated with other FT CORBA units) is the preferred way to introduce such control. A separate decision making can more naturally adhere to several principles that improve decision making: base the decisions on the most up-to-date system data, collect data from across the whole system or network, and use the best decision-making

algorithms available. Single applications and services have problems in all three areas, since they cannot anticipate what data and algorithms are available, or do not have permissions or the ability to collect certain data. In addition, the best decision-making algorithms are often not available beforehand, or depend on the peculiarities of the system under consideration, and a facility to plug in algorithms is therefore desirable. Obviously, the fault tolerance service can embed such a unit in an integrated manner, but the above arguments show that it makes engineering sense to deal with decision making in a separate unit.

2.2. QoS Objectives

To understand better the design objectives of a resource manager, we now turn again to the functionality definition of the resource manager. In this definition, we included the term 'QoS-aware': "the resource manager makes run-time, *QoS-aware*, decisions about parameterization and configuration of the fault tolerance service and its clients." We note first that there is some arbitrariness in the use of QoS as the objective. Instead of QoS, we could have considered cost as decision criterion (and thus talk about 'cost-aware decision making'); alternatively, we could have left the criterion vague, using words like 'meaningful decision making' or 'intelligent decision making' (as in [12]). However, our interest is in QoS as bottom-line criterion, and we therefore include it in the functionality definition.

Scope. When introducing QoS in a meaningful way, we must better understand the environment in which the fault tolerance service will be used. The environment very much influences what the scope of a QoS specification is, and whether it is realistic to assume that a resource manager can achieve the QoS objective. The question arises whether a resource manager for the fault tolerance service should only consider dependability metrics, and whether the mechanisms available in the fault tolerance service should only be used for fault tolerance reasons. To illustrate better the dilemma we face, consider the following scenario.

Assume a set-up in which the fault tolerance service runs as part of a general-purpose platform, like an application server. Clients expect a certain quality of service, but the metrics they are concerned about do not only entail dependability metrics. Instead, performance related issues, culminating in delay metrics, often play a prominent role (round-trip delay, access delay). So, if the resource manager makes QoS-aware decisions, it must be able to deal with metrics beyond typical dependability metrics. The question is, should we consider these metrics within the scope of the resource manager, or should such decision making be delegated to a different decision making unit outside FT CORBA?

On the flip side, the mechanisms the fault tolerance service provides may be useful for achieving QoS objectives beyond dependability. For instance, migration of a server through kill and start actions may achieve performance goals through effective load balancing. So, we have similar questions for the FT CORBA mechanisms as we stated above for the QoS metrics. Should the fault tolerance mechanisms only be used for fault tolerance purposes, or should there be a management platform that may trigger the mechanisms for other reasons as well? And, if we allow for the latter, how does this management platform relate to the resource manager?

We answer the above raised questions as follows. There is a clear need for a resource manager in the fault tolerance service to make sure that system pollution does not occur. However, if there exists a comprehensive middleware management solution, this management platform can use the mechanisms provided by the fault tolerance service, and take over (some of) the decision making from the resource manager. Potentially, this eliminates the need for an FT CORBA resource manager, but hierarchical or merged solutions (including a resource manager as well as a management platform) are possible as well. In any case, it leads us to the conclusion that the resource manager must be designed for extensibility, through pluggable algorithms and data collectors. We believe that this design is suitable and natural for both a dedicated resource manager (with single objective and supposed environment), and a more generic decision-making unit.

2.3. Further Design Issues

We briefly discuss various design issues that come up in the design of a decision-making unit like the resource manager.

Alarm Events. The resource manager parameterizes configuration and reconfiguration activities. These activities take place when certain events occur. The base events are:

- **Failures**: when a failure of a registered server is diagnosed, fail-over is initiated by the fault tolerance service. To parameterize the fail-over execution, calls to the resource manager are made. In this way, the appropriate host is chosen for the new primary, and for the possible new backup that is started.
- **Registration**: when a server registers with the fault tolerance service, the resource manager determines where to place the replicas, how often to checkpoint, and how frequently to poll the newly registered server.
- **Initialization**: concerning the placement of replicas, initialization works as described above for registration. Some other issues, like placement of object factories can also benefit from the resource manager.

If we choose a more generic management solution, various other events may lead to actions that need to be appropriately parameterized. We mention:

- **Change of QoS Metric**: the AQuA infrastructure [2] reacts to changes clients require in the level of dependability. The reconfiguration following such change in requirement, is parameterized through a decision-making unit (Proteus [9]). Of course, this scenario extends immediately to other QoS metrics, and changes in metrics (in addition to QoS levels).
- **Failure to Sustain QoS**: since system pollution is an immediate danger when adopting FT CORBA, it makes sense to trigger reconfiguration at instances the QoS gets below the desired level.

Data Collectors. To prevent system pollution, the resource manager must have sufficient data available to facilitate good decision making. At a minimum, the fault tolerance service's replication manager must feed data in the resource manager about the current configuration. In addition, failure statistics may be useful to guide reliability-related decisions. Obviously, if different QoS metrics and/or algorithms that are more advanced are used, other data should be made available. In that case, applications and services may feed data into the resource manager, or the resource managers starts up monitors that collect other system data (these monitors may for instance be based on SNMP).

System Model. The decision-making unit makes its decisions based on some information it gathers, either through data collection or through static input. This information is represented in the 'system model.' Central to the design of a resource manager is the question what the system model looks like. In QQS [11], we developed a system model that is suitable for decision making based on performance or reliability models, such as queuing networks, fault trees, or linear programming. The advantage of using this general system model in the resource manager is that it naturally extends to a more generic management platform.

QoS Queries. The resource manager must provide an interface definition that allows other units to query. The following queries are desirable in FT CORBA:

- What replication style?
- How many replicas?
- On which host to place a replica?
- What monitoring style?
- How long a monitoring interval?
- How long a checkpointing interval?

3. Algorithms

To conclude this paper, we discuss known algorithms that may prove useful for future resource manager implementations. Some of the algorithmic

approaches have been used in related projects, others have been proposed as likely candidates for on-line decision making. We focus on dependability related metrics in this section.

Replication Degree. To determine the appropriate replication degree, Garg *et al.* [3] developed a model that considers as QoS metric either availability (that is, the fraction of down time), throughput (that is, number of client invocations served per time unit) and probability of loss of client invocations through buffer overflow. The derived Markov models allow for easily computable closed-form solutions of the minimum number of replicas required. Variations of this model and approach are also possible, since small Markov chains can be solved very quickly for performability metrics.

Kalogeraki *et al.* [5] introduced an algorithm taking into account difference in importance among clients, the failure probability of (subtasks within) services, and the load and memory requirements clients impose on hosts. The resulting model is an optimization problem for a utility measure that relates to completion probabilities. This optimization problem may be solved through a greedy algorithm. Provided the algorithm solution is computationally efficient, this approach is especially interesting if 'importance' indices for clients are available.

The above-mentioned approaches illustrate the use of two basic techniques for decision-making algorithms: Markov process modeling and analysis, and optimization problem formulation and solution. For different environments and platforms, modeling assumptions may have to be changed compared to the mentioned solutions, but variations of the discussed solutions are likely to remain of use.

Fail-Over Host. In [11], we use tailored IBM SAVE algorithms [4] to solve Markov models for the optimal fail-over host. The metric of interest is either expected mean time to failure or unavailability, and optimization is achieved through computing the metrics for all alternative hosts. SAVE algorithms are likely candidates for quick computation of availability-related metrics. Note also that the SAVE algorithm allows one to work with failure and repair rates of components, which allow a more natural modeling of system failure behavior than failure probabilities, as used in [5]. On the other hand, if a good interpretation of failure probabilities is found, computation becomes simpler, and an approach like that of Kalogeraki becomes feasible.

In addition to the above algorithmic solutions, the literature provides various other approaches (see [5] for references). However, the above algorithms concentrate on dependability-related metrics, and we therefore highlighted those efforts.

4. References

[1] P. E. Chung, Y. Huang, S. Yajnik, D. Liang, J. Shih, DOORS: Providing Fault Tolerance for CORBA Applications, in *Middleware 98*, 1998. (for full paper: http://www.bell-labs.com/~shalini/doors.html)

[2] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, R. E. Schantz, AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects, in *Proceedings of the 17th Symposium on Reliable Distributed Systems*, pp. 245—253, IEEE Computer Society, West Lafayette, IN, USA, Oct. 1998.

[3] S. Garg, Y. Huang, C. M. R. Kintala, K. S. Trivedi, S. Yajnik, Performance and Reliability Evaluation of Passive Replication Schemes in Application Level Fault Tolerance, in *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, pp. 322—329, IEEE Computer Society, Madison, USA, June 15—18, 1999.

[4] A. Goyal, W. C. Carter, E. de Souza e Silva, S. S. Lavenberg, K. S. Trivedi, The System Availability Estimator, in *Proceedings of the 16th Annual International Symposium on Fault-Tolerant Computing*, pp. 84—89, IEEE Computer Society, 1986.

[5] V. Kalogeraki, L. E. Moser, P. M. Melliar-Smith, Dynamic Modeling of Replicated Objects for Dependable Real-Time Distributed Object Systems, *Fourth International Workshop on Object-Oriented Real-Time Dependable Systems*, Santa Barbara, CA, USA, Jan. 1999.

[6] B. Li, K. Nahrstedt, A Control-Based Middleware Framework for Quality of Service Adaptations, *IEEE Journal on Selected Areas in Communications, Special Issue on Service Enabling Platforms*, to appear, 1999. (http://cairo.cs.uiuc.edu/papers.html)

[7] OMG, Fault Tolerance RFP, *OMG document number orbos/98-04-01*, 1998. (http://www.omg.org/techprocess/meetings/schedule/Fault_Tolerance_RFP.html)

[8] OMG, Presentation of the Status of the Fault Tolerance Submission, *OMG document number orbos/99-08-27*, 1999. (ftp://ftp.omg.org/pub/docs/orbos/99-08-27.pdf)

[9] C. Sabnis, M. Cukier, J. Ren, P. Rubel, W. H. Sanders, D. E. Bakken, D. A. Karr, Proteus: A Flexible Infrastructure to Implement Adaptive Fault Tolerance in AQuA, *in Proceedings of the 7th IFIP International Working Conference on Dependable Computing for Critical Applications*, pp. 137—156, San Jose, CA, Jan. 6—8, 1999.

[10] M. Shankar, M. DeMiquel, J. W. S. Liu, An End-to-End QoS Management Architecture, in *Proceedings of the Real-Time Applications Symposium*, June 1999.

[11] A. P. A. van Moorsel, The 'QoS Query Service' for Improved Quality-of-Service Decision Making in CORBA, in *Proceedings of the 18th Symposium on Reliable and Distributed Systems*, Lausanne, Switzerland, to appear, Oct. 1999.

[12] A. P. A. van Moorsel, *Performability Evaluation Concepts and Techniques*, Ph.D. thesis, University of Twente, 1993.