Dimensions for Reliability Contracts in Distributed Object Systems

Jari Koistinen
Software Technology Laboratory
Computer Research Center
HPL-97-119
October, 1997

E-mail: jari@hpl.hp.com

distributed systems, reliability, QoS, contract

Designing and managing distributed systems with predictable reliability and availability is generally difficult. Whenever components are specified, used, and managed it is often unclear what reliability requirements different components are expected to satisfy. The problem of specifying and satisfying reliability requirements needs to be addressed for many different situations and contexts. We need languages and tools that allow design time and run time specification of reliability requirements and offerings. We need quality of service (QoS) contracts that can be negotiated and monitored dynamically. From a management view, we need the ability to define reliability contracts between system components so that they can be effectively managed and charged for in heterogeneous and federated systems.

Common to all these situations is the need for a vocabulary for the specification of reliability characteristics. Such a vocabulary should be complete in the sense that it captures the important aspects of service reliability without getting too complicated. In addition, it should be independent of any implementation techniques and mechanisms. This paper proposes such a vocabulary as a set of dimensions that can be used to characterize the reliability of distributed services. The dimensions are focused on distributed object systems but can easily be abstracted to cover other architectures for distributed systems.

Internal Accession Date Only

© Copyright Hewlett-Packard Company 1997

Dimensions for Reliability Contracts in Distributed Object Systems

Jari Koistinen, jari@hpl.hp.com

Software Technology Laboratory, Hewlett-Packard Laboratories

Technical Report HPL-97-119

October 3, 1997

ABSTRACT

Designing and managing distributed systems with predictable reliability and availability is generally difficult. Whenever components are specified, used, and managed it is often unclear what reliability requirements different components are expected to satisfy. The problem of specifying and satisfying reliability requirements needs to be addressed for many different situations and contexts. We need languages and tools that allow design time and run time specification of reliability requirements and offerings. We need quality of service (QoS) contracts that can be negotiated and monitored dynamically. From a management view, we need the ability to define reliability contracts between system components so that they can be effectively managed and charged for in heterogeneous and federated systems.

Common to all these situations is the need for a vocabulary for the specification of reliability characteristics. Such a vocabulary should be complete in the sense that it captures the important aspects of service reliability without getting too complicated. In addition, it should be independent of any implementation techniques and mechanisms. This paper proposes such a vocabulary as a set of dimensions that can be used to characterize the reliability of distributed services. The dimensions are focused on distributed object systems but can easily be abstracted to cover other architectures for distributed systems.

Table of Contents

1. Introduction 2
1.1 Background 2
1.2 Reliability 3
1.3 Services, Servers, etc. 4
1.4 Error, Failure, Failfast, etc. 4
2. Introduction to Reliability Contracts 5
2.1 Introduction 5
2.2 Client and Server Distinction 5
2.3 Contract Dimensions 6
3. Related Work 8
3.1 Overview 8
3.2 Characterizing Reliability and Availability 8
3.3 Failure Classification and Service Failure Semantics 9
3.4 Some Implementations 11
3.4.1 Introduction 11
3.4.2 Transaction Based Systems 12
3.4.3 Message Store 12
3.4.4 Active Replication 13
3.4.5 Primary-Backup 16
3.4.6 N-version Programming 19
3.5 Reliability and Availability Modeling 19
3.6 Time and Frequency 20
4. Dimensions in Detail 23
4.1 Client View Reliability Specification 23
4.2 Server and System Reliability 26
5. Example 28
5.1 A Telephony Service Execution Framework 28
5.1.1 Introduction 28
5.1.2 System Architecture 28
5.1.3 Reliability 30
5.1.4 Discussion 31
6. Concluding Remarks 32
Acknowledgments 32
References 32

© Hewlett-Packard Company

Do Not Distributed Without Permission

1. Introduction

1.1 Background

The work presented in this document is primarily concerned with three related aspects of reliability in distributed systems: (1) how do you formulate reliability requirements between parties during the design, implementation, and execution of distributed object systems? (2) how do you describe reliability agreements in federated and heterogeneous environments? and (3) how do you describe reliability contracts so that systems can evolve and still satisfy their quality-of-service (QoS) requirements? More specifically, we are interested in finding a set of dimensions that are practical, useful, and adequate for describing the reliability of services in distributed object systems.

Traditionally, different solutions for reliable systems are characterized in terms of their implementations. Such characterizations focus on the types of components in a system: how they fail, how their failures affect the system, and how they recover. In our view, such characterizations are generally not adequate as the basis for the design of reliable distributed systems, characterization of reliable services, or as the basis for the search of new reliability solutions. The reason is that they tend to focus on specific mechanisms too early and do not allow the reliability characteristics of the system and its components to be understood before implementations are considered. We argue that a vocabulary for formulating reliability contracts is necessary for the successful design of reliable distributed systems.

The maintenance of large distributed business systems is increasingly being out-sourced to external organizations. In addition, large business systems are often heterogeneous, and their management is often federated [SLO94]. To identify and correct a failure it is necessary to have well-defined and understood *service level agreements* that are not implementation specific. Again, we argue that we need a vocabulary for expressing quality-of-service contracts and in this document we are concerned with such a vocabulary for reliability.

The explicit definition of interfaces is considered to improve the modularity and therefore the evolvability of software systems. `OMG IDL`[CORBA] is one example such an interface definition language. `IDL` allows designers to define the syntactic interface between clients and servers. The clients and servers can evolve independently as long as no change in the interface is needed. Work is going on to establish similar stability with respect to the semantics of interfaces. We would like to add quality of service contracts. Such contracts would allow servers to evolve as long as they maintain their quality of service obligations. In this paper we are particularly interested in how such contracts can be formulated for distributed services.

The terms *reliability*, *availability*, and *fault-tolerance* are used in many different ways. Sometimes they are used interchangeably and other times they are used generally to capture various concepts. In yet other cases they have been used for very specific concepts and are some times represented by well-defined metrics. We will use the term *reliability* to denote a set of characteristics (*dimensions*) that we believe characterizes what is intuitively and in practice meant by service reliability. We consider *fault-tolerance* as an even wider concept encompassing techniques and tools that allow us to build reliable systems.

We are interested in how the reliability of a service (implemented by for example a `CORBA[CORBA]` or `DCOM[DCOM]` server) can be characterized without revealing or asserting internal details of the service. The benefit of having such a characterization is that clients can understand the reliability of services and adapt their behavior accordingly. Such characterizations are useful both during the design and implementation phases as well as during the deployment of clients and services. They can also be used in service level agreements and consequently enable more effective management of various kinds of systems. Syntactical interface definitions and semantic description—such as pre and post conditions—enable simplified evolution and replacement of components. We argue that QoS characterizations are equally important in this respect and should be specified for many services. As illustrated by the figure below, we therefore argue that QoS characterizations should be treated on an equal footing with syntax and semantic specifications. The challenge is to find a set of dimensions that are complete and valid in the sense that they represent a valid and useful characterization of service reliability.

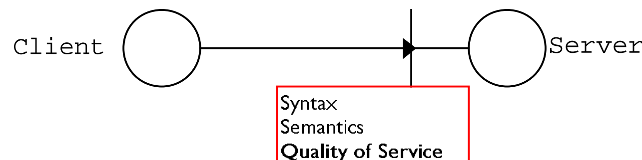


Figure 1 Quality of Service in Client/Server Contracts

Such a set of dimensions will also represent the vocabulary for languages that allow the specification of Quality of Service characteristics. QML [FRKO] is an example of such a language.

This report proposes a dimensionality for reliability based on a survey of techniques for the development of reliable distributed object systems. We believe that understanding how reliability can be characterized will allow us to specify services and more abstractly compare the reliability provided by different technical solutions. An understanding of reliability dimensions is also necessary to develop software-engineering support, middleware quality of service mechanisms, and fault-tolerant systems in a systematic way [AVI97].

It is generally impossible to only consider reliability without taking other aspects such as performance into account. Furthermore, the borderlines between various qualities of service categories such as reliability and performance are quite vague. Nevertheless, we believe it is an advantage to demarcate and understand one quality of service category well before we combine and weight characteristics from different categories. Therefore, this paper will therefore focus only on dimensions for what we consider unique for reliability.

1.2 Reliability

Reliability of software system is becoming increasingly important as the cost of downtime increases [CHOU]. We believe that reliability must taken into account systematically for any system, and not only—in contrast to common practice— systems where crashes has catastrophic consequences. Commonly, the development of reliable systems can be characterized by two different approaches: *fault prevention* and *fault-tolerance* [SHRIV, SOMA97]. Prevention deals with methodologies, testing and other techniques that help us avoid introducing faults into a system. Fault-tolerance acknowledges the existence of faults and provides techniques for dealing with failures. In this section we will set a context for our discussion by providing some common terms and definitions in the reliability realm.

Fault-tolerance is commonly characterized by the following metrics: *reliability*, *availability*, *mean-time-to-failure*, and *expected number of failures* [REIB91]. According to Reibman and Veeraraghavan [REIB91] reliability can be described as:

> *the probability of a system performing its purpose adequately for the period of time intended under the operation conditions encountered.*

Availability can be characterized as the probability that a service will be available when clients attempt to use it. The availability of a service can be high even if it has relatively frequent failures. On other aspect of availability is whether a service will be available every time a client accesses a service during a limited time period. We call this aspect *continuous availability*[1]. An aircraft control system requires high continuous availability during a flight while a telecommunications system can accept short failures if the overall availability remains high.

Mean-time-to-failure (`MTTF`) is the expected time until the system fails [FENTON92, GRAY] and the *expected number of failures* is the number of failures expected during a certain period of time. The time it takes to repair a system is commonly named as *mean-time-to-repair* (`MTTR`). These all represent commonly used reliability measures. Ideally, it would be useful to provide distributions for mean-time-to-failure, mean-time-to-repair, and number of failures.

Improving reliability and availability involves avoiding introducing errors in hardware as well as in software. It also involves making hardware and software less sensitive to failures. In hardware we often replicate resources, such as processors and storage, so that if one unit fails there are other units that can continue. Hardware solutions usually have the advantage that they are highly transparent to software applications. This means that services based on software that has not been designed with reliability in mind can be made more reliable by introducing reliable hardware. Unfortunately, hardware solutions are usually more expensive and often require specialized hardware. In addition, they don't protect against software errors. This means that if an application is running on replicated hardware a software error is likely to occur in all replicated instances simultaneously.

An alternative—and complement—to hardware replication is to introduce reliability in software. Such solutions are usually less expensive to produce since production costs (as opposed to development costs) are lower. In addition, such solutions can take advantage of pre-existing communication and computer infrastructures. There are many different software solutions for reliable systems and they differ greatly in these characteristics. Some of them impose high performance overhead while others have significant recovery times and so forth. Thus, selecting the right solutions is a major design decision in distributed systems development.

---

[1] We use of the term *continuous availability* differently that many other authors. We use if for uninterrupted availability for a limited time. Other uses it as a term for systems with very high-availability involving aspects such as runtime maintenance and upgrade.

Our question is what set of dimension—such as MTTF etc. —is appropriate for describing the reliability of a variety of services independently of how they have been implemented and what mechanisms they use.

1.3 Services, Servers, etc.

To describe reliability mechanisms we will distinguish between *service specifications*, *services*, and *servers*. A *service specification* describes the functionality that service implementations of it are expected to provide. A service specification typically consists of an interface definition and associated semantic specifications.

A *service* represents a particular realization of a service specification and has a well-defined *service reference,* or *reference* for short. Concretely, a service reference could be a reference to a distributed object as defined by CORBA [CORBA] or DCOM. In our view, services conforming to a common service specification should provide the same syntactic and semantic interface but may differ in their quality of service characteristics[2]. Finally, a *server* is a collection of objects that provide a service. A server can for example be implemented by replicated objects. This would, however, be transparent to clients.

To give an example, a CORBA IDL[CORBA] interface *T* would represent a service specification. An implementation of *T* and any other help objects needed to realize the service specified in *T* would represent a *T* server. The reference used by clients to call the server would represent the service reference.

A server *S1* providing a service *Q* may in turn use another service *W* realized by a server *S2*. From the view of a *Q* client *S1* and *S2* together can be seen as the server implementing *Q*. We will, however, still consider *S1* and *S2* as distinct servers if they can be distributed with respect to each other.

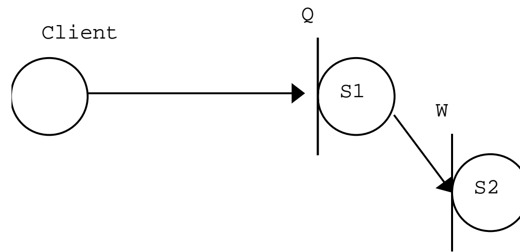


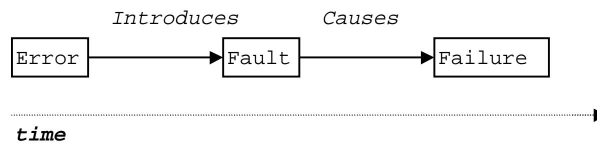
Figure 2 Services and Servers

We will use the term *software system* (or *system* for short) for structures of clients and services that perform some useful function.

1.4 Error, Failure, Failfast, etc.

For the continued discussion it is essential that we establish common terms for errors, failures etc. The following description is mainly drawn from [GRAY] with some adaptations and changes. A service has a *specified* and *observed behavior*. As mentioned in the previous section, a service is specified by a service specification and implemented by one or more servers. There may exist differences between the observed behavior of a service and the individual servers. There can for example be differences in the failures that are masked by individual servers and the failures that a client of the service will actually see. We take a client view; therefore we say a *failure* occurs when the observed behavior of a service is different from its specified behavior.

A *failure* is the result of a *fault*, and the fault occurred because of an *error*. The time between fault and a failure is called the *error latency*. The figure below illustrates the relation between the three concepts. As an example, a programmer can make an error that introduces a fault. When the server executes the fault can cause a failure.



[2] We define *quality of service* broadly to cover performance, reliability, security and other aspects commonly not captured by syntactic and semantic descriptions.

Figure 3 Errors, Faults, and Failures

Different implementations of a service mask different failures. Some services mask failures when servers crash, other services might even mask failures when one or more servers return incorrect values. Then again, there are services that do not mask any failures; thus any failure in the server is visible to the client. The failures that a service implementation exposes to its clients define its *failure masking* characteristics.

There are different kinds of failures, such as omission and response failures. An omission failure occurs when a service fails to respond to a request. A response failure occurs when a service responds with a corrupted value or state transfer. We say that a fault is detected or effective when it actually causes a failure. A server is said to be *failfast* if the failure causes the server to stop and the time between the detection of the fault and the failure is short.

We sometime distinguish between *Heisenbugs* and *Bohrbugs*. A *Heisenbug* is an error that only occurs occasionally for example depending on the execution environment. The term *Heisenbug* has also been used for failures that disappear when a failing application is run in a debugging environment. In contrast to *Hiesenbugs* the occurrence of a *Bohrbug* is deterministic.

2. Introduction to Reliability Contracts

2.1 Introduction

The two main contributions of this paper are the client/implementation distinction in reliability contracts and dimensions for characterizing reliability from a client's perspective. In the following two sections we will summarize these results. The results will be motivated and further explored in sections 2, 3 and 4.

2.2 Client and Server Distinction

Encapsulation and interfaces are well known concepts in software engineering. An entity exposes an interface that mandates what operations clients can perform on the entity. Everything except what is exposed through the interface is hidden from clients and can consequently be changed without affecting them. We want to make the same distinction for reliability contracts by distinguishing between client perspective and a design/implementation perspective. This distinction is usually not made in literature on reliable systems. Rather reliability discussions tend to encompass whole systems. We believe it is important to distinguish the two views for several reasons.

Firstly, a QoS contract can only include dimensions that a client can relate to. Implementation independent contracts enable different implementations for services even if they are supposed to provide the same level of reliability. Client perspective characterizations also allow registration and search of services with certain characteristics in for example trader services [OMGTRAD].

Secondly, we do not want to expose the internal structures of services as parts of QoS contracts. To achieve for example a specific level of availability we can use many different implementation strategies. Thus, we do not want to describe provided reliability in terms that are specific to the technique used to implement a service. As an example, some of the dimensions can be decomposed into sub-components when a particular implementation is considered. For primary-backup solutions the *mean-time-to-repair* could be related to *failover time* and *state recovery time*, but from a client perspective generally only *mean-time-to-repair* is of interest.

As illustrated by the figure below the *client view QoS contract* should only include the dimensions that are relevant to clients. A QoS characterization from an implementation view would typically include additional dimensions such as how components are assembled and how failures in one component affect other components. For clients, however, these details will be irrelevant since regardless of how the system fails it will become unavailable to clients.

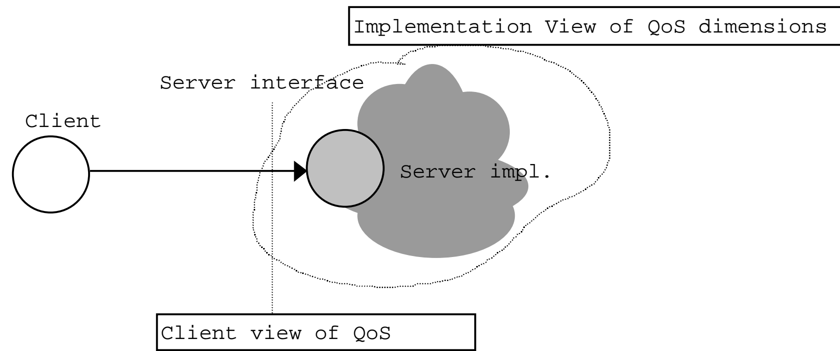


Figure 4 Scope of Client and Implementation QoS Characterizations.

2.3 Contract Dimensions

We are proposing a set of dimensions for characterizing reliability contracts from the client view. These will be presented briefly in this section while more detailed descriptions and motivations are deferred to section 4.

To define an adequate set of dimensions we considered the following criteria as essential: (1) the dimensions should involve generally used characterizations of reliability; (2) they should not expose implementation decisions; (3) they should allow clients to understand how to meet their reliability requirements; (4) they should enable clients to take appropriate actions when a service fails; (5) and it should be possible to determine if a reliability contract is satisfied or not.

The first criteria implies that we should search in the literature and technology for appropriate ways of characterizing reliability. The second criteria states that we should be able to use the dimensions regardless of how a service is implemented. Thus, dimensions should not be applicable only in situations where specific technologies are used. The next criteria states that a client (programmer), by considering the characteristics of services that is uses, should be able to understand what techniques it needs to use in order to satisfy the reliability requirements imposed on it. The fourth criteria requires that the set of dimensions is such that a client (programmer) can understand the failures a service can expose and what actions the client needs to take to recover or mask such failures. The last criteria requires that we can in some way determine whether a contract expressed in these dimensions is in fact satisfied. We do not require that it can be determined statically, rather we expect to need run-time monitoring. The following table summarizes our proposed dimensions.

Description	Definition
MTTR = Mean time to repair	Time in for example milliseconds.
V(MTTR) = Variance of MTTR	Statistically[2] defined as the mean value of the variable defined as (R-MTTR) where R is a stochastic variable for time to repair and MTTR is mean-time-to-repair
Max (MTTR)) = Maximum value allowed for MTTR.	Time in for example milliseconds
MTTF = Mean time to failure	Time in for example milliseconds. The mean value of the stochastic variable *time to* failure (denoted by the variable F).
V (MTTF) = The coefficient of variance[3] when we consider mean time to failure a stochastic variable.	Statistically[2] defined as the mean value of the variable defined as (F-MTTF) where F is a stochastic variable for time to failure and MTTF is mean-time-to-failure.

Number of Service Failures = The expected number of service failures N within a certain time period T.	`N per T`
	`N is an integer`
	`T is time, in for example hours.`

Continuous Availability/T = The probability that a service will be functioning during a time interval T.	Probability ={0….1} Time period = T
Availability/t = The probability that a service will be available at a point in time.	Probability = ={0….1}
Failure Masking = The types of failures masked by service.	`set {omission, lost_response, no_execution, response, response_value, state transition }`

[3] See your favorite statistics book for details.

Service Failure = The way in which the service fails. Does it recover state, start up an initial state, etc.	`enum {halt, initial_state, rolled_back, no_guarantees}.`
Operation Semantics = The semantics of pending request when a service fails	`enum { no_guarantees , least_once, most_once, once}`
Rebinding Policy = Does the client need to rebind to the service after the service failed.	`enum { no_guarantees , rebind, norebind}`
Data Policy = Will data provided by the service be valid after the service has failed.	`enum { no_guarantees , valid , invalidated}`

Table 1

We propose *availability* and *continuous availability* as two of the dimensions used to characterize the reliability of services. Availability is the probability that a service is accessible by clients at any point in time. Continuous availability is the probability that a service available to a client that makes multiple calls during a period of time every time the client attempts to use the service. For continuous availability we also require that the client can rely on preservation of state between subsequent calls. Availability and continuous availability are usually useful for different types of services and express quite different reliability requirements. To improve the accuracy of availability characterizations—and other dimensions that we propose—we could introduce distributions of for example availability over time. It is, however, unclear how practical the usage of such distributions is. One problem is the validity of distributions and the other is simply to understand which one is applicable in a particular situation. We therefore do not include such distributions in our current proposal for dimensions.

In our view a service should be considered available as long as it performs according to its specification. Thus, if a service specification allows degradation, we consider it to be available as long as the specification is meet. Alternatively, if we wish to make distinctions between different modes of operation availability needs to be defined separately for each such mode.

We have also included *mean-time-to-failure* and *mean-time-to-repair*. Mean values do not take variation into account. To cover variation we have also included the *coefficient of variance* for `MTTF` and `MTTR`. Using a distribution would be even more precise, and as for availability we do not exclude such extensions in the future. We also include the *number of service failures* during a specified time interval.

Failure masking is a dimension that characterizes what types of failures a client can be exposed to. A characterization can use zero or more of the values in the failures masking domain. A client that is exposed to a failure of a service has the choice of either propagating the failure to its clients, or to mask the failure.

The *service failure* dimension describes how the server behaves in case of a failure. More precisely we are interested in whether we should expect it to restart and if so in what state it will be in.

Operation semantics describes what happens to pending requests when a service fails. The question is whether the requests will be executed *exactly once*, possibly *more than once*, or *zero* or *one* time.

Rebinding policy describes whether clients need to rebind (to obtain a new service reference) to a service after a failure or if it can use the reference it had before the failure occurred.

Finally, *data policy* describes whether data that we have obtained from the service will be valid or invalid after a failure.

No guarantees is a value used in dimensions as an indication that the service in question can not guarantee any specific quality of service with respect to that particular dimension. Using this value is different from leaving the quality of service unspecified.

The following section will describe work and implementation techniques for reliable systems. The dimensions that we propose are selected to be suitable in describing and distinguishing between the reliability that these different techniques provide. It is therefore valuable to understand what characteristics common techniques have.

3. Related Work

3.1 Overview

Fault-tolerance is an active research area which focuses mostly on new technologies and techniques for reliable systems, reliability modeling and other related areas. The following sections will discuss related works, each of which focus on some specific type of reliability mechanism or aspect of reliability. The purpose of this discussion is to provide a background and motivation for the dimensions we have selected to characterize reliability solutions.

We first look at some definitions and characterizations that are used for reliability and availability. After that we study some work on how failures can be classified and characterized. In the following section we look at various implementation techniques, such as active replication and primary-backup. Finally, we study some common frequency and time measures used for various characterizations of reliability solutions.

3.2 Characterizing Reliability and Availability

To recapitulate, we use the term *reliability* to denote a set of dimensions that collectively describes an intuitive notion of service reliability. We also introduce two different flavors of availability. *Continuous availability* is defined as the probability that a service is performing adequately during a specific period of time. *Availability* on the other hand is defined as the probability that a service is available at a certain time. Sometimes availability is calculated from *mean-time-to-failure*, *mean-time-to-repair* etc.

In a draft document ISO/IEC[ISO/IEC] presents availability as the "portion of time in service" and reliability as the mean time between failures. Resnick[RESNICK] couples availability and masking, and describes them as follows:

> **Manual masking**: Following a failure some manual action must be taken to make the service available again.
>
> **Cold Standby**: Following a failure, users of the component are disconnected and lose any work in progress. An automatic fault detection and recovery mechanism detects the fault and brings in a redundant server. Once the redundant server is initialized, it can start accepting requests. Clients need to reconnect to the server to issue new requests.
>
> **Warm Standby**: This works as cold standby, except that the redundant server is partially initialized. Thus, recovery times are shorter than for cold standby.
>
> **Hot Standby/Active replication**: In this solution there are several servers with updated states. Thus, the recovery time is minimal, although not necessarily zero.

Resnick's[RESNICK] definitions of *cold standby*, *warm standby*, and *hot stand by,* etc. are oriented towards implementation, rather than characterization, of reliability. The standby techniques mentioned focus on different levels for recovery support and masking, and will thus provide different recovery times.

Resnick also distinguishes between *high* and *continuous* availability. The difference is that continuous availability encompasses both *planned* as well as *unplanned outages*, while planned outages are purposely initiated due to, e.g., maintenance. Resnick argues that, in general, availability is only concerned with unplanned outages. We do not fully agree with Resnick's definitions. What is especially bothersome is the intermixing of failure masking and availability. In addition, his characterizations are very implementation-oriented and consequently not suitable as the basis for reliability contracts.

We prefer to decouple the availability dimension and the failure masking. Availability is only concerned with how often a client would experience failures using the service. This view is also taken by Gray and Reuter [GRAY]. They define system availability as:

> *The fraction of the offered load that is processed with acceptable response time.*

This definition also takes performance problems into account. In addition, they suggest that system availability should be expressed as a percentage. If a service is unavailable one day out of a year the availability could be expressed as 364/365 = 0.997, i.e., 99.7% of the time. In addition, they present a table in which systems are grouped in *availability classes*. The table looks as follows:

System Type (Availability class)	Unavailability (min./year)	Availability
Unmanaged	52,560	90%

Managed	5,256	99%
Well-managed	526	99.9%
Fault-tolerant	53	99.99%
High-availability	5	99.999%
Very-high-availability	0.5	99.9999%
Ultra-availability	0.05	99.99999%

Table 2

A common formalization of availability is to define it as the relation between mean-time-to-failure and mean-time-to-repair. The assumption is basically that a system is either up or is being repaired and therefore the total elapsed time is the sum of `MTTF` and `MTTR`. Availability can then be defined as:

$$Availability \equiv \frac{MTTF}{MTTF + MTTR}$$

In a sophisticated model we could take client usage patterns into account. Assume that one client accesses a service during normal hours and another client accesses the service after hours. The availability for these clients could be different because of different loads, planned outages, etc. We currently do not consider such variations as part of our proposed reliability dimensions.

Reliability is sometimes used to denote the probability for fault free operation during a specific time interval. We could express a probability 0.999 of failure within a two-hour operation period as `0.999/2h.` This kind of reliability can not be measured simply as the time between failures. Even if `MTTF` for a service is longer than the time period specified as a reliability requirement, this does not mean the component is reliable enough. The variability of `MTTF` could be such that the probability of a failure before the allotted time has elapsed is more likely. We will use the term *continuous availability* to denote that a service must provide fault-free operation for a specific time interval. This means that during this time interval a client should be able to successfully use the service every time it attempts to do so.

A limitation of the availability definitions above is that these do not reflect the number of failures that occur, and that they do not reflect the variability or distributions of `MTTF` or `MTTR`. In addition, it could be interesting to allow different levels of distribution over time.

It is common to introduce an attribute defined as the *expected number of failures* within a certain time period. This metric is useful from both a client and an implementor's perspective, although it should be noted that the two values are different. From the client's perspective we include only failures that are visible to the client and therefore must be handled by the client. From the implementation perspective we also use the measure to determine how many replicates, back-ups, etc. are needed. We propose that reliability contracts include the number of failures from a client's perspective.

We have chosen to introduce the expected number of failures as well as the variances for `MTTF` and `MTTR` from the client's view. In fact, even more elaborate statistical measures such as percentiles and distributions can be used as well. We propose that such be used based on the need and the knowledge that we have about the characterized dimension.

3.3 Failure Classification and Service Failure Semantics

The failures exposed by a service represent an important dimension both from the view of a client of a service as well as the designer of the service. For a discussion on the relationship between specified behavior, error, failure, etc. see section 1.4.

A service failure occurs when the service does not behave in the specified manner. To describe the types of failures seen by the client we need a classification and description of the *types of failures* that services may expose to their clients. A service is said to *mask* a failure if it can handle internal failures or failures exposed by services it depends on without exposing any failures to its clients. A description of masked and exposed failures represents a specification of failures the service is expected to detect and handle internally. It is also describes to the client what failures the client should be able to handle or propagate.

Failure semantics define what the implications of a failure will be. We are particularly interested in what happens to the server and its state. This section will present some failure types and failure semantics as they are described in the literature.

Cristian[CRIS91] identifies a number of different failure types. As an example, he describes an *omission* failure as the failure of a server to respond to a request. He also identifies *timing* failures as a separate type although it is not always clear how clients differentiate timing and omission. He identified *late* and *early* timing failures as sub-types of timing failures. According to Cristian, a *response* failure occurs when the server either returns an incorrect (or corrupt) value or makes an incorrect state transition in response to a request. The table below summarizes the types of failures that he proposes.

Failure Type	**Description**
Omission failure	The server fails to respond to a request.
Timing failure	The server response is functionally correct, but untimely. Thus, the server fails to fulfill real-time requirements.
Early timing failure	Sub-type of timing failure. Occurs when the server responds too early.
Late timing failure	Sub-type of timing failure. Occurs when the server responds too late.
Response failure	The server responds incorrectly.
Response state failure	A sub-type of response failure. The state transition that took place in the server is incorrect.
Response value failure	A sub-type of response failure. The output value of the response is incorrect.

Table 3

The Figure 5 illustrates a hierarchy of failure types. As an example, early and late timing failures are sub-types of the more general notion of timing failures. It is preferable to get as specific as possible when failure masking and contracts are specified.

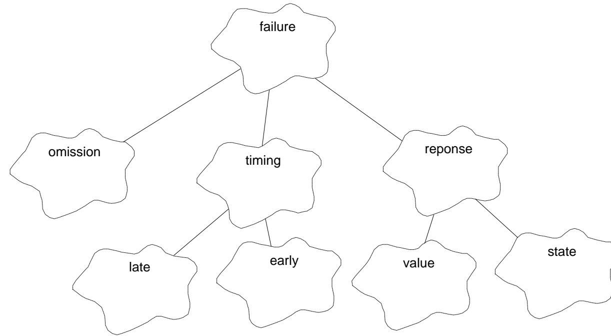


Figure 5 Failure Types

Gray et al. [GRAY] divide the set of possible failures into *expected* and *unexpected* failures. Expected failures are the ones that the system can handle and detect, while unexpected failures can not be recognized or effectively handled. Unexpected failures can be divided into *dense* and *Byzantine* failures. A dense failure means that there are a larger number of expected failures than the system can handle. A Byzantine failure includes all failures that are not masked. For example, if a service only masks omission failures it will not be able to mask response failures. In this situation—and according to the above definition—response failures are considered Byzantine failures.

We also want to characterize the behavior of the server after the failure occurs. Cristian[CRIS91] describes a set of distinct failure semantics, which he calls *crash*, *amnesia*, *partial amnesia*, *pause-crash*, and *halting-crash*. As an example, a server is said to have *crash failure* semantics if it fails to respond to any request after an initial omission failure. Another failure semantics example is *amnesia crash semantics* in which the server fails to remember the state it had before the failure. The following table describes the failure semantics as defined by Cristian[CRIS91]:

Service failure Semantics	**Description**
Crash failure	The server fails to respond to any requests after an initial omission failure.

Amnesia failure	After a failure the server restarts in a state independent of the state before the failure.
Partial amnesia failure	After a failure the server restarts in a state that is partially the same as the one before the failure and partially an initial state that is independent of the state it had before the failure.
Pause-crash failure	The server restarts, in the state it had immediately before the failure.
Halting-crash failure	The server never restarts.

Table 4

In our view the service failure semantics proposed by Cristian cover in an abstract and sufficient way different kinds of behavior. As an example, transactional systems can be viewed, as partial amnesia-failure where the state loss depends on to what point the system is rolled-back. In other cases, services truly lose their state, which obviously is a property clients want to know about. It can, however, be beneficial to add information to the characterization so that clients more fully understand how the server fails. Such characteristics are usually application specific and are therefore not included as part of our proposed dimensionality.

Like Cristian[CRIS91] we believe that the failure semantics of a service is important and worth specifying, thus making part of the *service specification*. Cristian states that if the specification of a service *s* prescribes that the failure behaviors (likely to be observed by clients of *s)* are of type *F*, we say that *s* has *F* failure semantics. As an example a service that is likely to delay requests is said to have performance failure semantics. We describe a service that has performance failure semantics and that also might lose requests as having omission/performance failure semantics. Generally, if a service may exhibit failures in the union of the types F_1,F_2…F_i, we say that it has $F_1/F_2/../F_i$ failure semantics. The larger the set of failures expose, the weaker the failure semantics. When a service may expose any failure we say that it has arbitrary (or Byzantine) failure semantics. This is also the weakest possible failure semantics.

Failstop—also called *failfast*— is a commonly assumed failure semantic in various theoretical models. Failstop services expose omission failures but with some additional characteristics. Schneider[SCHN84] states that a process (or processor) is failstop, if it fulfills the following properties:

Halt on Failure: In response to a failure, the process halts rather than perform an erroneous state transformation that will be visible to other processes.

Failure Status: Any process can detect when any other process has failed and therefore halted.

Stable Storage: The storage is partitioned into stable and volatile storage. The contents of the stable storage are unaffected by any failure and can always be read by any processor.

If we can assume that a process is failstop, the construction of fault-tolerant computing systems will be significantly easier. Unfortunately failstop, is quite unrealistic in real computing environments. Therefore, we will not include failstop as in the possible failure semantics.

3.4 Some Implementations

3.4.1 Introduction

Classifying the implementations of fault-tolerant systems is an extensive survey in itself. This paper is therefore restricted to the description of some common architectures for fault-tolerant systems. Our purpose is to present different software-based solutions to fault-tolerance. Most current implementations of fault-tolerant software systems adhere closely to one of these architectures. We do not cover hardware solutions at all.

A software system supporting fault-tolerant computing usually has the following components: a failure *detection* function, a *recovery* function, and a *coordinator* function. The failure detection function detects when a server has failed. It is often difficult to detect a failure because of the difficulties in distinguishing omission failures and performance problems. In advanced systems, failure detection functions detect if one in a set of replicated servers provides incorrect answers. The recovery function helps the service recover from a failure. In a transaction system this means reading logs and updating the state of a newly created server. In a replicated system it means updating the state of a newly started replica. The coordination function may, for example, coordinate the answers received from different replicas. The importance and exact roles of these functions vary in the techniques used for reliable software systems.

3.4.2 Transaction Based Systems

In a transaction system we may treat a series of actions as one atomic unit, called a *transaction*. If all the actions are successful, the transaction is *committed.* The requested changes are guaranteed when the commit has been completed successfully. If the transaction fails, none of the actions within the transaction will be visible outside the transaction. Thus, it is fault-tolerant in the sense that the system can expose failures without going to an inconsistent state. The transaction model assumes we can restart the system after a failure, expect it to function correctly, and be in a consistent state. Database systems the classic examples of transaction-based systems [GRAY].

The usage of transactions is based on the assumption that most failures are *Heisenbugs*—a term used for transient, non-deterministic failures. A Heisenbug failure is one that is not likely to occur again. If we reinitiate the transaction that failed it is likely to succeed. These failures are usually not hidden from a client; rather the client needs to explicitly reinitiate the failed transaction or take some other action.

Most transaction-based systems use databases, and databases have a significant recovery time. One reason for this is that databases must recover by reading logs etc. In addition, transaction-based systems do not mask failures very well; even single failures are usually not masked to clients. Rather, clients must reissue those requests that were not committed when the failure occurred.

The table below characterizes database transaction services according to the dimensions proposed in section 2.3. Observe that many dimensions are application dependent (A/D), although we can recognize certain patterns. As an example, we claim that the recovery time usually is long, although it ultimately is application-dependent.

In terms of service failure semantics the databases usually are expected to have roll-back semantics. Since databases are used as persistent storage we expect data to be valid even after failures. Thus, commonly, the value of data policy is `valid`.

Dimension	**Typical value**
MTTR =Mean-time-to-repair	`A/D, but usually long for databases systems because of elaborate recovery procedures.`
V(MTTR)	`A/D`
Max(MTTR))	`A/D`
MTTF = Mean-time-to-failure	`A/D`
V(MTTF)	`A/D`
Expected Number of Service Failures	`A/D`
Continuous Availability/T	`A/D`
Availability/t	`A/D`
Failure Masking	`omission`
Service Failure	`rolled_back`
Operation Semantics	`most_once`
Rebinding Policy	`A/D`
Data Policy	`Commonly valid, since all retrieved data is already committed.`

Table 5

3.4.3 Message Store

Message store[COU94] systems deal with the problem of reliably delivering messages from one process to another. The basic principle is that a local message queue handler that can store messages on non-volatile storage. Once the client delivers the message to the message queue handler, the client is relieved from any additional concerns of delivering the message. The message queue handler delivers the message to the server. If the server is down at the time the client sends the message, the message queue handler simply waits until the server comes up. If the message-queue handler crashes, the message remains in the storage. The message will be delivered to the final (or next) destination when the message queue has recovered.

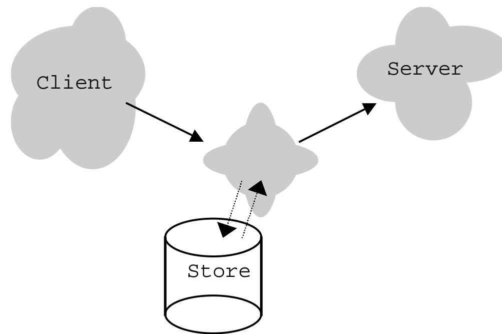


Figure 6 Message Store

Although this approach provides reliable delivery of messages it does not guarantee any availability or fast recovery. We believe a system can be classified as a message store system if it has the following properties:

- When a client gets an acknowledgment from the message queue, it can rely on the message queue to eventually forward the request to the server.
- Any requests received by the message queue will be stored until they are delivered to the server.

The advantage of message store systems is that they are reasonably simple to implement and are useful for asynchronous store and forward communication is acceptable. Message store does not facilitate highly available services and provides only one-way communication.

The table below gives a rough idea of what characterizes message store services. For most dimensions we can not assign concrete values until we have a specific instantiation of the service in mind. But we can notice that they usually provide an exactly once operation semantics, they provide data validity, they use persistent storage, and can recovery state.

Dimension	Typical value
MTTR =Mean-time-to-repair	A/D
V(MTTR)	A/D
Max(MTTR))	A/D
MTTF = Mean-time-to-failure	A/D
V(MTTF)	A/D
Expected Number of Service Failures	A/D
Continuous Availability/T	A/D, usually no guarantees.
Availability/t	A/D, usually no guarantees.
Failure Masking	omission
Service Failure	rolled_back
Operation Semantics	once
Rebinding Policy	A/D
Data Policy	valid

Table 6

3.4.4 Active Replication

Transaction-based systems often have relatively long recovery times and lack processing redundancy. A long recovery time means they fail to provide a high degree of availability even though they provide fault-tolerance. The lack of processing redundancy means that database server crashes will not be masked to clients and thus decrease the availability of the service. Highly available services can be achieved by replicating the servers and thereby introducing redundancy. If one server fails, the service is still available since there are other servers that are able to process incoming requests. There are two main techniques for achieving such software-based redundancy*: active replication* and *primary-backup* [GUE]. The principal difference is that in primary-backup, there is one process that

receives and responds to all messages. In addition, there are backup processes that receive state updates from the primary and are thus kept up-to-date and ready to takeover. In active replication a group of processes all receive and all respond to the same messages. The distribution of messages and responses is handled by another abstraction commonly called *process group*. One goal of both techniques is to make the redundancy as transparent to clients as possible. Replication and primary-backup represent two ends of the replication spectrum, but there are technologies that are hybrids of the two. The Somersault protocol [HARRY95] developed at Hewlett-Packard Laboratories is an example of such a hybrid. This and the following sections describe the active replication and primary-backup techniques respectively.

Active replication is also called *process replication* or *the state-machine approach* [SCHN90, MULL93]. The principle is that one abstraction represents a set of replicated processes. All the processes within the set provide the same service and are therefore interchangeable. In addition, they all have the same state or can be synchronized so that they do. Thus, at any given time it is likely that there is at least one process that can accept requests and thus provide the service. The set of replicated processes is commonly called a *process* or *object groups* [LANDIS].

In order to make such a system function correctly we must manage the ordering *of messages*, *detection of server crashes*, and *reliable multi-cast* to all servers in the group.

Ordering of messages means that all processes within the group must receive the requests in the same order. The *ordering* can be defined in many different ways. *Total* and *casual* ordering are two common kinds of message ordering within object groups [BIRMAN96, MAFF95B]

Processes within a group must be able to detect that other processes within the group have failed. It can sometimes be hard to distinguish between real failures and network delays and other problems. A wrong diagnosis may lead to increased overhead and performance penalties. Often in active replication, detection is part of the communication protocol but it could also be performed an external monitoring system such as the Piranha[MAFF97B] prototype.

Finally, we must guarantee that all or none of the processes within a group receive a request. This is usually called *request atomicity*. Request atomicity is necessary to keep the states of processes within a group consistent.

Processes in a process group sometimes have different roles. Commonly, there is a primary process and one or more secondary processes. The primary process can actually provide the response. Sometimes the primary does other things like indeterministic choices. The division of functionality and responsibility among primary and secondary processes is system- and replication-protocol dependent. The figure below illustrates the object group concept and shows that all requests to the group are distributed to all processes that are members of the group. The picture also shows that from a client perspective, the group is accessed from a single service reference.

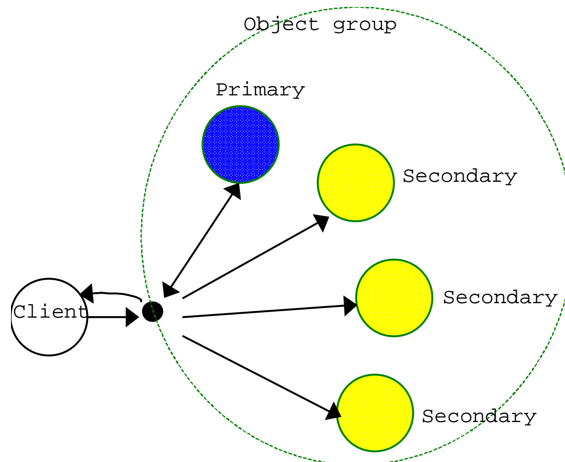


Figure 7 Replication Using Group Communication.

Virtual synchrony [BIRMAN89, BIRMAN96, MAFF97] is probably the best-known execution model for active replication. Virtual synchrony makes all significant events appear to happen at the same time in all replicated servers. The set of significant events includes delivery of requests, failures, group membership changes, etc. It is easy to see that the coordination and synchronization needed in replicated systems commonly implies communication overhead and consequently, a performance penalty. There are alternative solutions that trade the benefits of active replication for better performance and less delay. One tradeoff is to allow inconsistency among the states of the replicas. By doing so, some of the communication and synchronization among replicated servers can be omitted, and the

performance will therefore improve. The down side is that recovery times will be longer and some states might get lost in case of a failure, since the states of replicas are not up-to-date.

Active replication with virtual synchrony is generally best suited for systems with volatile states. Services implemented with active replication can fail transparently to a client since messages are not lost if one server fails. In addition, active replication can provide very high availability with recovery times around seconds or even less. The delays that occur in case of failures often have to do with failure detection. As an example, a detection mechanism must be confident that a delayed response or heartbeat is caused by a failure rather than a network delay. To determine this with acceptable probability the detection mechanism needs to wait for a while to see if the heartbeats was just delayed.

Degrees of replication range from active replication (also called hot standby) to cold standby. The difference lies in how well synchronized the states of the primary and the secondary processes are. If they are totally synchronized the system can be considered hot standby since we can fall over to a secondary process momentarily. In other cases, the primary logs the state changes and secondary omits updating their respective states. This decreases the communication overhead since the protocol does not need to worry about request atomicity. The down side is that the failover takes longer and thus decreases availability. These kinds of solutions is called cold or warm standby depending on when the state of the secondary was last updated. For a cold standby system the main part of the recovery time will be spent recovering the process state. There are, however, other activities that take time as well. One such activity is called *failover*. The failover involves detecting that a replicated server has failed and taking the appropriate action. The figure below gives an idealized picture of the relationship between state replication and recovery time, where recovery time includes state recovery, fail-over etc. In practice there is no linear relationship; rather it all depends on additional aspects such as what implementation techniques are used, etc.

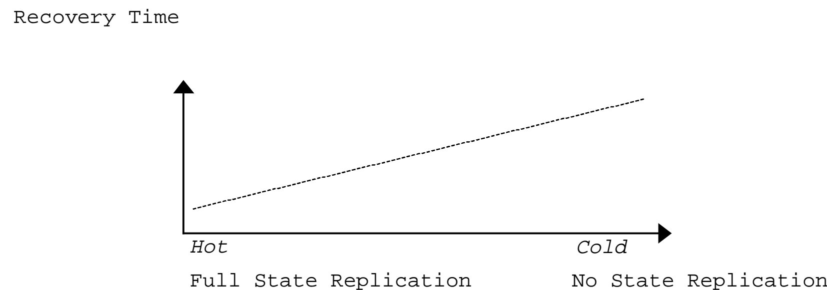


Figure 8 Continuum from Hot to Cold Standby

There are a number of systems that implement replication. Consul [MISH93] is one such communication mechanism. Consul is built according to the state machine approach [SCHN90] and includes fundamental functions such as reliable multi-cast, membership services, and recovery services.

ISIS [BIRMAN89] is probably the most widely known communication package supporting active replication. The virtual synchrony concept originates from the ISIS project. There are a some recognized problems with ISIS. Firstly, ISIS does not handle partitioned networks well, that is what happens if replicated servers are unable to communicate with each other. Secondly, ISIS does not allow simple composition of replicated services. The problem is that a service that has replicated clients will see duplicated requests from each client although there logically there should only be one. ISIS not only a replication mechanism, rather it can be viewed as a general reliable multi-cast mechanism that can be used for different purposes such as load sharing or replicated processes. ISIS has been used to develop an object request broker supporting actively replicated distributed objects [Orbix+ISIS]. ISIS—and it successor Horus— has also been used in research prototype object request brokers supporting active replication[MAFFEIS95A].

Transis [DOL95] also supports reliable multi-cast based on group communication. In contrast to ISIS, Transis allows continued operation in partitioned networks. This means that when a network is partitioned the separate parts of a divided group continue to operate according to the virtual synchrony model.

Somersault [HARRY95] is an active replication system with some primary backup-like mechanisms. It is an active replication system in the sense that all the replicated processes receive almost all the messages. A Somersault service has two redundant processes. Although they are basically identical, they do have slightly different roles. The two processes are a *primary* and *secondary* server. Somersault is a primary-backup approach in the sense that some

processing—indeterministic choices—are made in one of the processes. Furthermore, responses are sent only from the secondary process. This protocol, which decreases communication overhead, is called the *secondary sender* [FLEM95] protocol. The secondary sender protocol improves the efficiency with which a replicated system responds to clients and keeps the states of both servers (primary and secondary) synchronized. In addition, the secondary server algorithm allows Somersault-based services to be composed. The figure below illustrates the architecture used in Somersault.

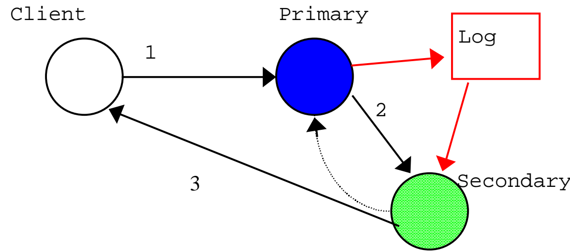


Figure 9 Somersault

The response is actually returned from the secondary. Indeterministic choices are made in the primary and logged so that the secondary can get the same value as the primary. An example of an indeterministic choice would be to get the time of day.

The following table provides characteristics of a system implemented with an active replication mechanism.

Dimension	**Typical value**
MTTR =Mean-time-to-repair	A/D, but usually short. Depends on the failure detection mechanism and how many state values that need to be transferred.
V(MTTR)	A/D
Max(MTTR))	A/D
MTTF = Mean-time-to-failure	A/D, usually long since a failure from the client view requires that all replicated servers fail simultaneously.
V(MTTF)	A/D
Expected Number of Service Failures	A/D, usually very low.
Continuous Availability/T	A/D, usually very high.
Availability/t	A/D, used to provide very high availability.
Failure Masking	Response. There are no guarantees that response failures will not occur. But messages are generally not lost, depending on the protocol.
Service Failure	Initial. If the service fails in a way not masked to clients (all replicated servers fail), it is usually that the state is lost due to the type of application when active replication is used.
Operation Semantics	most_once
Rebinding Policy	A/D
Data Policy	A/D. When all replicated servers fail, it is application dependent whether the data is still valid or not.

Table 7

3.4.5 Primary-Backup.

A service implemented with the *primary-backup* approach has a *primary server* and one or more *backup servers*. Clients send requests to the server that they believe is the primary server. This is different from replication where the fact that there are several servers is transparent to the client. Replication uses *a group abstraction* mechanism or *recovery unit* mechanism with which the client appears to communicate. In contrast, a primary-backup client communicates with one particular server. All requests sent by a client to a server that is not a primary server will be

lost. Therefore, clients need to be notified or be able to detect if a primary fails. When a primary fails, the client needs to figure out which process is the new primary.

For each received request the primary server sends state updates to its backup servers. It also sends regular heartbeats so that the backups are able to detect when the primary server fails. The figure below illustrates a client communicating with a primary server directly and that the primary server sends state updates and heartbeats to its backup servers.

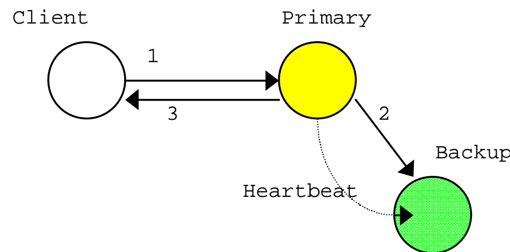


Figure 10 Primary-Backup

The advantages of using a primary-backup mechanism rather than active replication are that there is generally less communication overhead and less redundant computation. A disadvantage is that failures are not masked as efficiently to clients. Typically, messages can be lost, and consequently, clients need to be able to handle such situations.

The main cost-factors associated with primary-backup are [BUD] *degree of replication*, *blocking time*, and *failover time.* The degree of replication is measured as the number of servers used in a service implementation. The blocking time is the worst case delay caused by the primary-backup mechanism between request and response in a failure free system. Finally, the failover time is the maximum time that there is no well-defined primary. The failover time is relevant since messages can be lost during this period.

The primary-backup approach has been characterized quite precisely. One purpose of having such a strict characterization is to detect problems in the specification and implementation features that claim to support primary-backup. Another reason is that we can characterize formally the degree of redundancy, blocking time, and failover time. The primary-backup approach is described by the following four properties (see [BUD] for more formal descriptions):

1. There is at most one server that is identified as the primary server at anyone time.
2. Each client maintains the identity of the server it believes is the primary server. All requests are sent to this server.
3. If a request arrives at a server that is not the primary server, the request is discarded.

The fourth and last property deals with the case when all requests are lost. Properties 1 through 3 do not prohibit such behavior. Assuming all clients are supposed to receive an answer, a *server outage* is defined as the time when a client sends a request but does not receive an answer. The last property then states that:

4. There is a fixed number k of server outages not exceeding the time Δ in length.

These four properties can be used to verify that a reliability solution is in fact a primary-backup. The characteristics can also be used to derive the degree of redundancy we need to obtain a certain level of failure tolerance, called *n-fault tolerance*. A system is *n* fault tolerant if it can tolerate *n* components failing simultaneously. The level of fault tolerance is determined by the types of failures that we wish to mask, and the degree of redundancy. Budhiraja et al. [BUD] introduces five failure models and derives the number of components required to support each of them. The five models (They use the term *link* for the communication connection between a client and the primary server.) are:

Crash Failure: The primary server fails by halting and does not respond to any further requests**.** Crash failures are tolerated by having redundant servers.

Crash and Link Failure: The primary server fails by halting and does not respond or a link may loose requests. We do not include failures where links delay, corrupt, or duplicate requests. Link failures are tolerated by having redundant links.

Receive Omission Failure: A primary server does not *receive* requests despite a non-faulty link. This could be caused by for example in-buffer overflow. This type of failures is tolerated by correcting problems at the server side, by for example increasing buffer size.

Send Omission Failure: A primary server does not *send* requests despite a non-faulty link. This could be caused by for example out-buffer overflow. We can tolerate this kind of failures by correcting the error in the server, or in the client. In the out-buffer overflow case, the problem is in the client application rather than in the server.

Request Omission Failure: If a primary server exhibit both send and receive omission failures say that it exhibits *request omission failures*.

To a client many of these distinctions are invisible. We will, however, need to make these distinctions to analyze the level of redundancy that is needed to make a primary-backup based service n-fault tolerant. The following table from [BUD] summarizes the lower bounds L for the number of components needed to achieve *n*-fault tolerance with a specific failure model.

If we wish to tolerate *n* simultaneous *crash failures,* the number of servers needs to be greater that *n*. If we wish to tolerate *n* crash and link failures, the number of servers needs to be greater than *n+1,* and so forth, see Table 8. The lower bounds described here are also based about some assumptions on the primary-backup protocol (see [BUD] for further details).

Failure Model	Lower bound for redundancy
Crash	L > n
Crash and Link	L > n+1
Receive Omission	L > ceiling(3f/2)
Send Omission	L > f
Request Omission	L > 2*f

Table 8 Relationship between n-fault Tolerance and Failure Models

Table 9 can be used to understand the relation between failure models and what is required to make a service n-fault tolerant with a specific failure model. Similarly it describes what characteristics a primary-backup server has with respect to blocking time and failover time. We will, however, defer this discussion to section 3.6. These aspects are more implementation-oriented, and are, in our view not relevant to clients.

From the previous description we can see that primary server solutions commonly have less communication overhead than active replication solutions. The performance characteristics will, however, be different depending on whether the primary needs an acknowledgment from backups when their states are updated. The performance characteristics for primary backups are usually better than those for active replication. On the other hand, active replication usually provides better failure-masking characteristics. With primary-backup, clients need to know the identity (from a client perspective) of the primary server. If the primary server fails, the client needs to know how to recover and switch over to the new primary server. There is also a fail-over period during which messages can be lost.

Dimension	Typical value
MTTR =Mean-time-to-repair	A/D, usually short but longer than for active replication.
V(MTTR)	A/D
Max(MTTR))	A/D
MTTF = Mean-time-to-failure	A/D, usually long.
V(MTTF)	A/D
Expected Number of Service Failures	A/D. Server failures are more frequent than service failures. Server failures are not masked to clients. Service failures, on the other hand, are expected to be rare.
Continuous Availability/T	A/D, usually high.
Availability/t	A/D, usually high.
Failure Masking	Omission, response. Messages can be lost especially during rollover. No guarantees are provided for the correctness of responses.
Service Failure	rolled_back

Operation Semantics	most_once, messages can be lost, especially during rollover.
Rebinding Policy	Rebind. Rebinding is usually required if the primary fails.
Data Policy	A/D

Table 9

3.4.6 N-version Programming

N-version programming means that we have *N* implementations of a specification that are run in parallel. All servers receive all requests and a majority vote is taken on the response. The goal with n-version programming is to remove some of the problems associated with the errors introduced during the development of software systems. By having several separately developed versions of a program, the hope is that they will not fail in the same way. Thus, there should be a high-probability that a majority of the *N* servers will provide the correct response.

It seems that the same underlying techniques—multicast, request ordering, etc.—is required for N-version programming. The difference is that each of the servers in an object group is implemented differently and by different people.

N-version programming usually leads to much higher development and maintenance costs. It has also been shown that although independently developed, errors are likely to be introduced in related parts of the server code. This means that different programmers are likely to introduce errors for the same functions of a service specification. The following figure illustrates different implementations of servers. Each request is forwarded to each of them and they all respond. The responses are compared, and the response received from a majority of servers is sent back to the client.

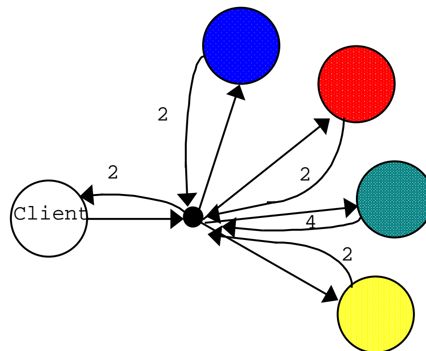


Figure 11 N-version Programming

In summary, N-version programming requires voting and coordination among several servers. It is therefore likely that it will have performance characteristics similar to replication. The advantage of n-version programming is that programming and design errors can be neutralized. Unfortunately, due to human nature, the outcomes will not be as good as could be expected. It is, however, a viable technique to improve the availability of replicated systems.

In this brief survey of implementation techniques we have pointed out the main characteristic of each techniques. Although the exact characteristics are application dependent, we have also indicated that the set of proposed dimensions are indeed useful in characterizing reliability contracts.

In the topics of the next two sections *are reliability modeling* and *time and frequency*. The purpose of these sections is to point out main ideas and uses of reliability modeling and generally discuss some time and frequency measures for reliability.

3.5 Reliability and Availability Modeling

One of the difficulties of designing reliable and available systems is that of choosing between design alternatives. It is also difficult to predict how reliable and available a particular solution will be. Reliability modeling attempts to address these issues by providing techniques to model various aspects of system reliability [LITT91]. Reliability modeling can help designer with the following [REIB91]:

- To set and interpret reliability requirements
- To predict the reliability of different system configurations
- To identify weak points and bottle necks with respect to reliability
- To evaluate alternative designs with respect to reliability, cost performance, etc.

There are three basic types of reliability models [REIB91]: *Parts-count* models, *Combinatorial* models, and *State-space* models.

A *parts-count model* assumes that the failure of any component in a service can cause the failure of the entire service. The reliability of a service can therefore be assessed as the sum of the reliability of its components. A parts-count model is quite conservative and not sophisticated enough to take the interesting aspects of fault-tolerance features into account.

Combinatorial models are more advanced than parts-count models in that they can model simple redundant systems. There are variations of combinatorial models called, for example *fault tree*, *success tree*, and *reliability block diagrams*. A success tree provides a model of what is required for the system to function. A success tree can be described as a series of AND gates from which inputs are components vital for the functioning of the system. The weakness of combinatorial models is that they do not model certain aspects, such as whether a system can handle a specific fault, system repairs or other properties of fault-tolerant systems.

State-space models are more expressive than combinatorial models and are therefore often used in advanced modeling of fault-tolerant systems. In a state-space model all combinations of functioning and non-functioning components are represented by a state. The model is used to evaluate the probability that the system to is a specific state. Based on this information other values such as mean-time-to-failure can be predicted.

For software, reliability modeling generally focuses on predicting the reliability and fault content of software systems. More specifically, the focus is on predicting the number of faults in a system and the frequency with which they cause failures. Gokhale et al. [GOKHALE] provides an overview of popular techniques and models used for software reliability modeling. Such models estimate reliability and availability at specified times, based on available software failure data. There are mainly two classes of such models: *data* domain models and *time* domain models. The prevalent definition of reliability for both these models is consistent with the one we have been using previously in this paper: *the probability of fault-free operation, provided by the software product under consideration, over a specific period of time in a specified operational environment* [GOKHALE].

The fundamental principle of data domain models is that, if the set of all input combinations can be identified, we can estimate the reliability by observing the relationship between inputs and outputs. Finding and executing every input combination is, in practice, not feasible. Thus, finding a representative set of inputs is one of the major challenges of this method.

Time domain models model the underlying failure process of the software under consideration. This model, with an observed failure history, is used to estimate both the residual number of faults in the software and the test time required to detect them.

Reliability modeling for software is focused on giving us trustworthy information so that we can make a statement about the reliability of a system. A final assessment of reliability can not be made until the system has been running for a period of time. Therefore reliability modeling helps us gain confidence that a particular software product will indeed satisfy the reliability requirements. Fault injection[HSUEH97] is a complement to reliability modeling. The goal of fault-injection is to make different components fail and observe the overall reliability of the system.

Reliability modeling techniques do not directly provide us with dimensions for characterizing and specifying reliability of services. They do, however, give us tools to analyze and predict the reliability of a specific service. Reliability modeling is therefore essential during the design of reliable services.

3.6 Time and Frequency

Various time measures are often used to characterize reliability and fault-tolerance. These measures are also used to derive new metrics for availability and reliability. In this section we will describe some common measures and their usage.

Name	Abbreviation	Description
Mean-Time-To-Repair	MTTR	The mean time it takes to repair a service after it failed.
Mean-Time-To-Failure	MTTF	The mean time between any two failures.

Table 10

The measure *mean-time-to-failure* (MTTF) is often used as a measure for the reliability of a system. Time-to-failure is the time between two subsequent failures of a service. MTTF is, consequently, the *mean* time between each consecutive failure. MTTF does not take the variability or distribution of the time between failures into account. If the necessary information is available, it can be very beneficial to describe the distribution for MTTF. Variance is a weaker, but still useful, measure. Variance indicates how an actual value will vary from the mean. In addition, percentiles could is useful way of characterizing many different kinds of attributes, such as MTTF.

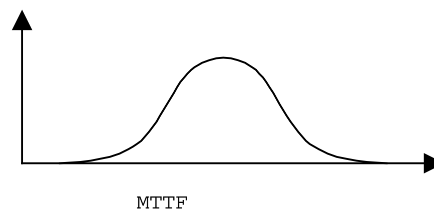


Figure 12 Distribution of MTTF

The time that the service is interrupted —and thus unavailable—is called the *time-to-repair*. It is often measured statistically as the *mean-time-to-repair* (MTTR). MTTF in combination with MTTR can be used to get a derived metric often used for availability [GRAY].

$$Availability \equiv \frac{MTTF}{MTTF + MTTR}$$

While MTTF and MTTR characterize reliability from the perspective of a client there are other time measures, such as *worst-case-failover-time* and *mean-state-recovery-time*, that are relevant from the server perspective. These measures are usually interesting for particular solutions and are therefore not useful as general characterizations. In some cases these times are part of MTTF or MTTR. As an example, *mean-state-recovery-time* is usually a component in *mean-time-to-repair*.

In primary-backup there is a period of time when there is no appointed primary-server and clients may not have detected yet that the primary-server has failed. During this period, requests issued to the primary server will be lost. We call the upper boundary for this time the *worst-case-failover-time*. The *worst-case-failover-time* is relevant in primary-backup solutions, but not in active replication. Blocking time is the worst-case elapsed time between the receipt of a request and sending the associated —failure free—response. This also assumes that the number of simultaneous failures has not exceeded what the system is designed to manage. In some cases, such as for primary-backup, different boundaries for some of the time attributes have been analyzed. Such analysis is valuable in that it provides insight into how a particular solution behaves and what the inherent timing constraints are.

Name	Abbreviation	Definition
Worst Case Failover Time	WCFT	The worst case time occurs when a request is lost because of failures.
Worst Case Blocking Time	WCBT	The worst case time between the time a server receives a request and the time the reply is sent.
Mean State Recovery Time	MSRT	The mean time to recover the state of a failed service. This is usually application dependent.

Table 11 Additional Time Measures

The table below indicates the relationship between failure models and the lower bounds for blocking time for primary-backup based systems. The table basically states that if you build a primary backup mechanism assuming a specific failure model you will have an associated lower bound on the blocking time. Recall from section 3.4.50 that *n* denotes the maximum number of faulty components that is tolerated (N-fault tolerance), and *s* is the number of servers. The lower bounds depend on the time (τ) it takes to transfer messages between processes, the degree of fault tolerance (n), and the number of servers used in the implementation (s). The blocking time defines the delay for executions when no failures occur. Although we focus on lower bounds, it should be noted that the bounds defined below are also the upper bounds in many cases. See the chapter by Buhiraja et al. [BUD] or the paper [BUD92] for details.

Failure Model	Blocking Time (Lower Bound)
Crash	0 (immediate response)
Crash and Link	0 (immediate response)
Receive Omission	τ when $n = 1$ and $s = 2$
	$2 * \tau$ when $n > 1$ and $s <= 2n$
Send Omission	τ when $n = 1$
	2τ when $n > 1$
Request Omission	τ when $n = 1$
	2τ when $n > 1$

Table 12

For systems assuming Crash or Crash/Link failure models there is no delay involved. The primary can respond immediately after it has sent state updates to the backups.

For *receive* omission failures we have two different cases related to two possible designs. If we only intend to allow one failure and we have two servers, we can use something that is similar to secondary sender in replicated systems. Concretely this means that the backup rather than the primary sends the response; thus, we only have the overhead of one message between the primary and the backup. The other solution requires that the primary gets messages from backups verifying the state updates, etc. The bound is dependent on the failure model (backups might lose received messages) and the number of failures we want to tolerate. The same reasoning applies to *send* and *request* omission failures. Budhiraja and Marzullo [BUD92] also describe optimal implementations of these mechanisms (except receive omission failures) and show that they are tight [BUD].

We have not seen any similar analysis for replicated systems. To do this we need to analyze individual solutions. As an example, we could derive the lower bound for the blocking time from the design of Somersault [FLEM95, HARRY95]. We will, however, not attempt to do such an analysis in this paper.

The fail-over time is the worst-case elapsed time when messages are lost because a server has failed and the service is recovering. For replicated systems this time is essentially zero, because if one server fails there are already other servers in the group receiving, processing, and responding to requests. Since the time is zero the possibility of losing messages is also zero, as long as the number of failures is less than *n*. For primary-backup solutions the situation is different. In primary-backup there is a period of time when a backup server needs to detect that the primary server has failed. Usually requests are lost if they are issued to the primary server between the time it failed and the time a backup server takes over, and the time clients are notified.

The following table—taken from [BUD]—shows the relationship between some failure models described in Section 3.4.5 and the lower bounds of failover times. τ is the maximum time it takes to transfer one request from one process to another.

Failure Model	Failover Time
Crash	$n * \tau$
Crash and Link	$2 * n * \tau$
Receive Omission	$2 * n * \tau$

Send Omission	2 * n * τ
Request Omission	2 * n *τ

Table 13 Failover Time for Primary Backup

In the case of request store systems, the client can rely on the delivery after it has delivered request to the message queue. Thus the failover time should be zero.

Our conclusion from looking at various time measures is that `MTTF` and `MTTR` capture what seems to be the appropriate characterization from a client's perspective. Other time measures are interesting in understanding a particular solution, but are usually also components of `MTTF` or `MTTR`. Another approach would be to include a much richer set of time measures in a client view contract. For many solutions these measures would be not applicable, or they would be zero. As an example, failover time is in principal zero for message store systems. This would also require that we describe the relationships between various measures. This description is complicated by the fact that the measures might aggregate differently depending on the solution in question. We have seriously considered including a measure for the time between the occurrence of the failure the time it is detected by a client.

One problem with `MTTF`, `MTTR`, etc. is that it can not be directly verifed that a service indeed satisfies certain `MTTR` and `MTTF` values. Rather, such requirements can only be verified until deployment when the system reliability is measured.

4. Dimensions in Detail

4.1 Client View Reliability Specification

Previously in this paper we surveyed various aspects of reliability that we use as the basis for selecting a set of representative dimensions. The table below (the same as in section 2.3) summarizes the dimensions that we have proposed. In this section we will describe the proposed dimensions in greater detail and provide some additional motivation and discussion.

Name	Description	Definition
MTTR	The mean time it takes to repair a service after it has failed.	Time in for example milli-seconds.
V(MTTR)	The variance that exists for mean-time-to-repair.	Statistically defined as the mean value of the variable defined as (R-MTTR), where R is a stochastic variable for time to repair and MTTR is mean-time-to-repair.
Max(MTTR)	The maximum time that a repair is allowed to take.	Time in for example milli-seconds
MTTF	The mean time between any two failures.	Time in for example milli-seconds.
V(MTTF)	The variance that exists in mean time to failure.	Statistically[2] defined as the mean value of the variable defined as (F-MTTF) where F is a stochastic variable for time to failure and MTTF is mean-time-to-failure.
Continuous Availability/T	The probability that a service will be functioning during a specific time interval T. Clients should be able to count on state information being preserved until subsequent calls.	Probability ={0….1} Time period = T
Availability/t	The probability that a service will be available at a point in time.	Probability ={0….1}
Failure Masking	The types of failures masked by a service. The client must handle failures not masked by the service.	`set {omission, lost_response, no_execution, response, response_value, state transition }`
Service Failure	The way in which the service fails. The state of the server after recovery.	`enum {halt, initial_state, rolled_back, no_guarantees}.`

Operation Semantics	The semantics of pending request when a service fails. Will they be executed or not.	`enum {least_once, most_once, once}`
Rebinding Policy	Will a reference to the service before a failure be valid after the service has recovered?	`enum { no_guarantees , rebind, norebind}`

Expected Number of Service Failures	The expected number of service failures N within a certain time period T. This should only include failures from which the client must recover from.	`N per T`
		`N is an integer`
		`T is time in for example hours.`

Data Policy	Will data supplied by the service before a failure be valid after the service has recovered from the failure?	`enum { no_guarantees , valid , invalidated}`

Table 14

Mean-time-to-failure (MTTF) and mean-time-to-recover (MTTR) are both measures that indicate reliability and availability. Generally, we define MTTF to represent the mean time over the total life of a system including periods of stable and less stable states. MTTR is the time a service is inaccessible to clients due to a failure. Neither MTTF nor MTTR reflects variance or distribution. We therefore propose that the estimated variance be included for both MTTF and MTTR. We denote the variances by V(MTTF) and V(MTTR), respectively. It also seems reasonable to state the maximum time of a repair; we denote it by max(MTTR). Statistical distributions for MTTF and MTTR can be defined, but we do not insist on them being a mandatory part of a contract.

MTTF and MTTR are used to derive the *availability* measure that we adopt. The availability measure can be informally defined as the probability that a service will be available when a client attempts to access it (at a specific time). Although there is a direct relationship between MTTR/MTTF and availability, we propose that all three of them be incorporated in to client contracts. An extension to availability could take various other parameters into account. As an example, the availability could be different at different times of the day. It could also differ among different users and usage patterns. At this point we do not elaborate on these aspects, rather we assume that multiple alternative contracts will be sufficient in such situations or dealt with by other mechanisms.

Continuous availability is the likelihood *P* that a service performs correctly during a specific time period T. This means that a client will be able to access a service arbitrarily many times for the period T with probability P. Continuous availability can also be thought of as the probability that a service will be available for all of a sequence of requests. Viewed this way, availability can be regarded as a special case of continuous availability when the sequence consists of just one request. As with availability we could introduce distributions to more accurately characterize the probability that a service will be continuously available.

By being explicit about which failures a server may expose, clients can be prepared to handle these failures. Assume that a server fails to mask failure of type *F*, then clients must be prepared to detect and handle failures of type *F*. In some cases it is trivial for a client to detect a failure, for example, whether a reference is invalid or not. In other cases it is more difficult, such as when the response value is corrupted. It is, however, important that a reliability contract specify the failure masking properties so clients understand the failure semantics of a service. We propose that *failure masking* be part of the reliability protocol and we currently draw the domain for this dimension largely from the work by Cristian [CRIS91] (described in Section 3.3). We will, however, modify the types of failures slightly, as illustrated by the failure type hierarchy shown below:

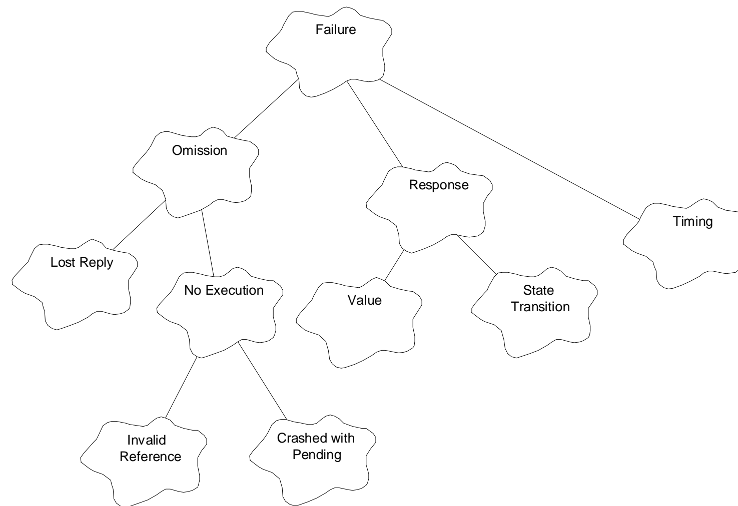


Figure 13 Failure Types

We add two sub-types of omission failures: *lost reply* and *no execution*. Lost reply means that the request was executed but the reply was lost. *No execution* indicates that the request was actually never processed by the service. It is important to notice that, while more specific failure types are preferable, the level at which a client can distinguish between failures depends on the support provided by the computing environment.

Service failure is concerned with the state of the server after an unmasked failure. The client will depend on this information for continued operation and recovery. The spirit of our dimension is similar to the one described by Cristian [CRIS91]. Cristian defines the set of possible states as crash, amnesia, partial amnesia, pause-crash, and halt (see section 3.2 for further details). Our domain is slightly different from that of Cristian's proposal. As an example, we believe that crash failures are relevant for all failure types, not only omission failures as he suggests. We will therefore introduce a slightly different set of service failure semantics: *halt*, *initial_state*, *rolled_back*, *no_guarantees*.

If the semantics is *halt,* the server will be down for an unknown time after the failure. The *initial_state* means that the server is reset into a predefined initial state that is known to be consistent. The *rolled_back* value indicates that the service state is rolled back, but we do not specify how far back. The roll back could be to the previous committed transaction or to some other well-defined point in time. Although we omit specifying the state to which the service is rolled back here, we believe it should be a part of the service specification. Finally, *no_guarantees* states that we can assert nothing about the state of the recovered server.

Operation semantics captures the semantics of issued, but not finished, requests to a service that fails. We can, for example, guarantee that the request will be executed once the service is up again. In other situations no such guarantees can be made and thus we have weaker semantics such as *least once* and *most once*.

The *expected number of service failures* is a generally useful measure in reliable systems. The measure is useful to get a more complete picture of the failure behavior together with other availability measures. For contracts this value only includes failures visible to the clients—failures not masked by the service.

From a client perspective, failure might or might not be visible. If the server masks the failure the client will not see it. On the other hand, if it is not masked, the client must (in its role as server) handle the situation and either mask the failure to its clients, or propagate the failure. Whether a client can mask a failure is dependent on the validity of the requests that the client has previously received from the service.

We are concerned primarily with distributed object-oriented systems. In such systems we use references to issue requests for objects in other processes. We call obtaining a reference *binding* to a service. Clients are certainly interested in how references are affected by the fact that a service fails. We propose the usage of a *rebind policy* to specify whether a reference is valid after a crash or not. Assume that the client has a reference to a service and that the service fails. It must then be clear to the client whether this reference is still valid or if the client needs to obtain a new reference to the same logical service. The concept of rebinding policy can be generalized to other kind of bindings in distributed non-object oriented systems as well:

rebinding policy = **enum** {rebind, norebind , no_guarantees}

For each unmasked service failure type we associate a rebinding policy. This means that, if a failure of that type happens, the associated rebinding policy should be used. Assume that a service does not mask response failures. We might associate *no rebind* with those response failures. This would mean that even if a response fails, the client doesn't need to get a new object reference to the service.

Another example is whether data previously received from a service will still be valid after the service has recovered. As an example, assume a client that has received a ticket from a service as part of a response. If the service fails it must be clear to the client whether the ticket is still valid. We call this aspect *data policy* and we want to be able to associate data policies with individual return values:

data policy = **enum** {valid, invalidated, no_guarantees}

Data policies are also associated with failure types and with certain data elements provided by the service. Thus, different data elements could have different associated policies. In the end though, the language used to specify such contracts will determine on what level of granularity (interface, operation, attribute etc.) these dimensions are applied.

A third important aspect is concerned with what happens to pending requests in case of failures. We have selected three basic forms of semantics that have been used for remote operations: *exactly once*, *at most once*, and at *least once*. The *exactly once* semantics guarantees that a request is always executed exactly once, which is hard to achieve in practice. The *at most once* semantics states that the call was executed if the service did not fail, otherwise it might not have been executed at all. Finally, *at least once* guarantees that the call is always executed once. This semantics is useful for idempotent operations, i.e., operations causing no harm even if they are executed more than once.

operation semantics = **enum** {least_once, most_once, once}

4.2 Server and System Reliability

We like to make the distinction between three views on reliability: *Client*, *Server*, and *System*. Client view reliability focuses on how reliable a client considers a service to be. Server reliability focuses on how a server provides a service with a particular reliability. Finally, the system reliability view considers the reliability of a system, which consists of multiple clients and servers that are inter-dependent. Server and system reliability are related but emphasize different aspects. They tend to focus on how to implement a service and how to compose a system respectively. We are mainly concerned with client and server reliability. This distinction is also described in section 2.2.

We have made a distinction between the reliability dimensions that are appropriate in client view contracts and those that are appropriate from the server view. In the server view we consider the failures that can occur and what we can do to satisfy the client view contract. The server reliability specification is a superset of the client view. Thus, everything used to characterize reliability from a client perspective is also useful from the implementation perspective. In addition, there are other dimensions, models, and information that are more closely related to the internal structure of a service. In this section we will briefly describe some of the additional dimensions and considerations that are useful from the implementation view. This is not intended as a complete set. Rather we would like to point out that client contracts are not enough to determine whether a server satisfies its reliability requirements and how those requirements are traded against other qualities.

From a server perspective we are interested in what the reliability bottlenecks are and generally how a service behaves when one or more components fail. We are also interested in how various dimensions can be broken down and mapped to smaller elements that are solution specific. As an example, for some solutions the timing dimensions described in client contracts can be divided into smaller pieces that all contribute to some value, such as `MTTR`.

The table below presents some useful server view dimensions. As an example, we propose an alternative definition of expected number of failures that measures the number of component failures. . Not all such failures result in a failure of the service. An implementation needs to consider how many failures and what combinations of failures it is able to handle.

There is a variety of time dimensions that can be used, such as blocking time, state recovery time etc. They are commonly solution-specific (primary-backup, replication, etc.) or depend on the application.

.

Name	Description	Definition

Expected Number of Failures	The expected number N of failures of components within a time period T.	N per T N is an integer T is time, in for example hours.
Blocking Time	The time T between request and response in a failure free execution	T is time, for example in milli-seconds.
State-recovery-time	The time T needed to recover the state of a crashed service so that the service can operate normally again.	T is time, for example in milli-seconds
Failure Time	The time interval T that requests can be lost because the service in a state between normal operation and failure.	Time in for example milli-seconds.

Table 15

One way of understanding how well an implementation handles failures and how likely the system is to fail is to use reliability modeling. The kind of reliability model to use for a specific system depends on the precision we need in predicting reliability and the effort we are willing to spend on the modeling. See section 3.5 for brief descriptions of and references to various modeling techniques.

Reliability can usually not be considered in isolation. In particular we can not consider solutions to reliability requirements without considering the impact they may have on other dimensions. Typically, different reliability solutions have different performance characteristics. As an example, replicated servers typically have execution overhead due to synchronization.

It is quite clear that reliability can not be considered without taking performance into account. The solutions that have been outlined in previous sections all have different performance characteristics. They differ in communication overhead, recovery time, failover time, etc. In addition performance characteristics commonly depend on the application. Recovery time depends on the size of the state that needs to be recovered; failover depends on configured time-outs and heartbeat rates; communication overhead depends on the degree of redundancy, etc.

It is a common perception that reliability and performance are tightly related. Although we have focused on reliability dimensions, these should generally be used with performance dimensions for a more complete characterization of solutions and applications. Throughput and delay are two commonly used dimensions for performance.

There are, of course many other quality aspects that need consideration when we implement a service and a system. Modularity, composability, and reusability are examples of such qualities. To take an example, let us consider composability. By composability we generally mean how simple it is to compose new systems out of existing self-contained modules, in our case, servers. As an example, let us think about a fictitious application with a *cassette player* (`CP`) abstraction. Such an abstraction could make use of a *recorder* (`R`) abstraction and a *player* (`P`) abstraction. When we build and initiate a cassette-player abstraction, we provide it with references to a player object and to a recorder object. The cassette player would use these objects to provide its own functionality. The figure below illustrates how we would compose such objects:

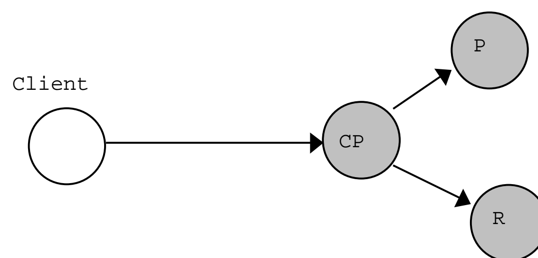


Figure. Composition of Services

Assume that in order to meet reliability requirements we intend to introduce redundancy. Thus, we could implement all the server objects using a replication or primary-backup approach. The question we need to answer is whether a redundant object is in fact composable. With some replication techniques composition will involve significantly higher complexity. Assume, for example, that the cassette player is replicated; thus, it consists of more than one actual object. In a non-composable replicated implementation each of these objects call the player service. Thus, the player service must handle several invocations from the cassette player.

Composability is not only the concern of service implementors, but is highly relevant to clients that are required to provide a service with references to complementary services such as in the case of initiating the cassette-player

described above. Composability is not a dimension to characterize reliability, but it is a characteristic that differs among reliability solutions.

Reliable systems generally have higher development costs than systems for which reliability has not been addressed explicitly. The tradeoff between cost and reliability is important and usually customer specific. To understand what reliability is worth we must estimate the downtime costs in lost business, productivity, and goodwill.

5. Example

5.1 A Telephony Service Execution Framework

5.1.1 Introduction

To illustrate the proposed dimensions and their expressability, we will use them to specify the QoS properties of an example system.

This example is a simplified version of a system for executing telephony services, such as telephone banking, ordering, etc. The purpose of having such an execution system is to allow rapid development and installation of new telephony services. The system must be scalable in order to be useful both in small businesses and for servicing several hundreds of simultaneous calls. More importantly—especially from the perspective of this paper—the system needs to provide services with sufficient availability.

Executing a service typically involves playing messages for the caller, reacting to keystrokes, recording responses, retrieving and updating databases, etc. It should be possible to dynamically install new telephone services and upgrade them at run-time without shutting down the system.

The system answers incoming telephone calls and selects a service based on the phone number that was called. The executed service may, for example, play messages to the caller and react to events from the caller or from resources allocated to handle the call.

Telephone users generally expect plain old telephony to be reliable, and they commonly have the same expectations for telephony services. A telephony service that is unavailable will therefore have a severe impact on customer satisfaction, in addition the service provider will lose business. Consequently, in this case we assert that the system needs to be highly available.

Following the categorization by Gray et al [GRAY], we want the telephony service to be a highly *available* system which means it should have a total maximum downtime of 5 minutes per year. The availability measure will then be 0.99999. We assume the system is built on a general-purpose computer platform with specialized computer telephony hardware. The system is built using a CORBA [CORBA] Object Request Broker (ORB) to achieve scalability and reliability through distribution.

5.1.2 System Architecture

We call the service execution system module *PhoneServiceSystem*. As illustrated by the figure below, it uses an *EventService* module and a *TraderService* module. We use the Booch notation[BOOCH] for our models.

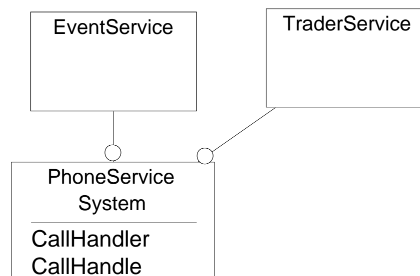


Figure 14 High-level Architecture

Opening up the *PhoneServiceSystem* module the Figure 16 below, we see its main classes. The important classes are *CallHandle*, *ServiceExecutor*, and *Resource*. The model also shows the dependencies to the *EventService* and

TraderService interfaces, represented as classes in the diagram. The *CallHandler* constitutes the abstraction of the service execution system as seen from the telephone switch. We assume there is an external system that detects incoming calls and issues the appropriate request to a *CallHandler* instance. The *CallHandler* is responsible for mapping the incoming phone number to a service identifier and creating a *CallHandle*. The *CallHandle* holds data essential to the call, such as the channel on which messages are played. The *CallHandle* must be passed around since it must be available to resources, such as players and recorders.

The *ServiceExecutor* class defines the operations to actually start and stop service executions for incoming calls. The *ServiceExecutor* is responsible for mapping service identifiers to a service description. The service description represents what the *ServiceExecutor* will in fact execute.

A resource is something that the *ServiceExecutor* needs to execute services. Databases, players, and recorders are examples of resources. The *Resource* defines a general interface supported by all resources. In addition, each type of resource provides specific operations defined in resource type specific interfaces inheriting from *Resource*. The *ServiceExecutor* calls resources directly and asynchronously. When a resource has finished a task it notifies the *ServiceExecutor* by sending an event by using the *EventService*. This communication model allows the *ServiceExecutor* to wait for many different types of events simultaneously, such as resource request completions, hang-up, keystrokes etc.

The *Service Executor* uses a *Trader* to find the resources it needs for a specific service execution. In the model we have added a reference to the reliability contracts that we will describe later. Each reliability contract includes a description of the reliability that the client can expect and that the service it agrees to provide.

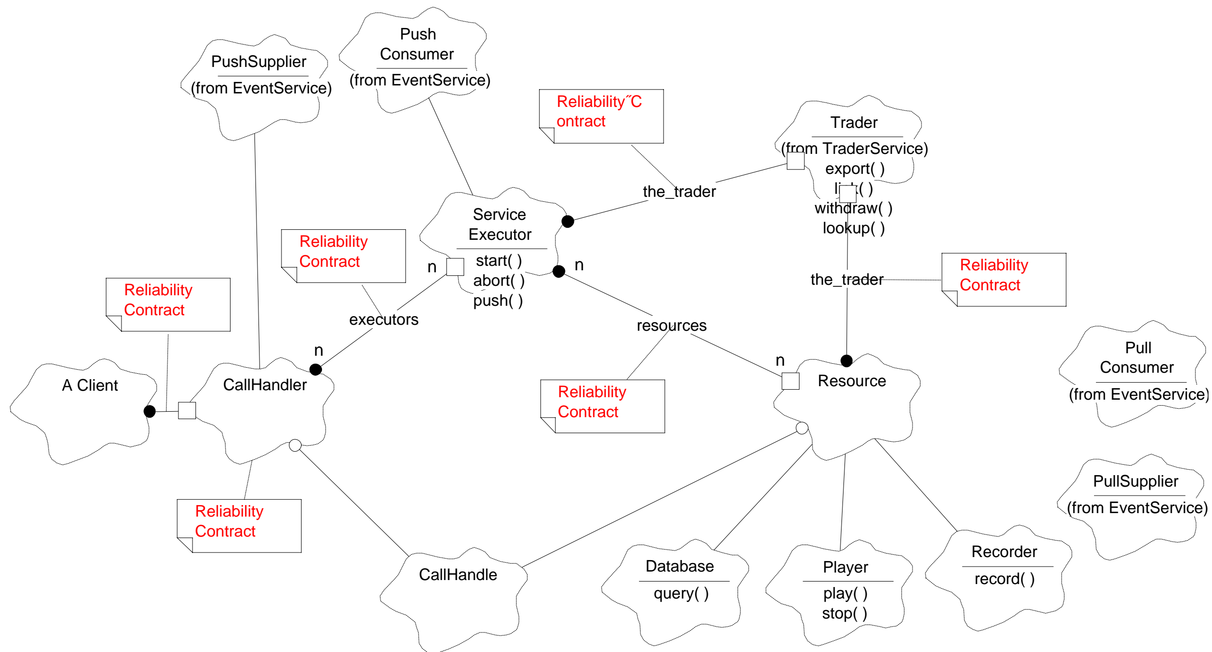


Figure 15 Class Diagram for PhoneServiceSystem

The scenario diagram in Figure 16 shows a sequence of events for a call. The *CallHandler* receives an incoming call (not shown) that triggers a call to the *ServiceExecutor* and conveys the service identifier. The *Service Executor* figures out what resources it needs and calls the trader to get them. After receiving the right resources, the *Service Executor* executes the service specification, which involves calling various resources and responding to user interactions.

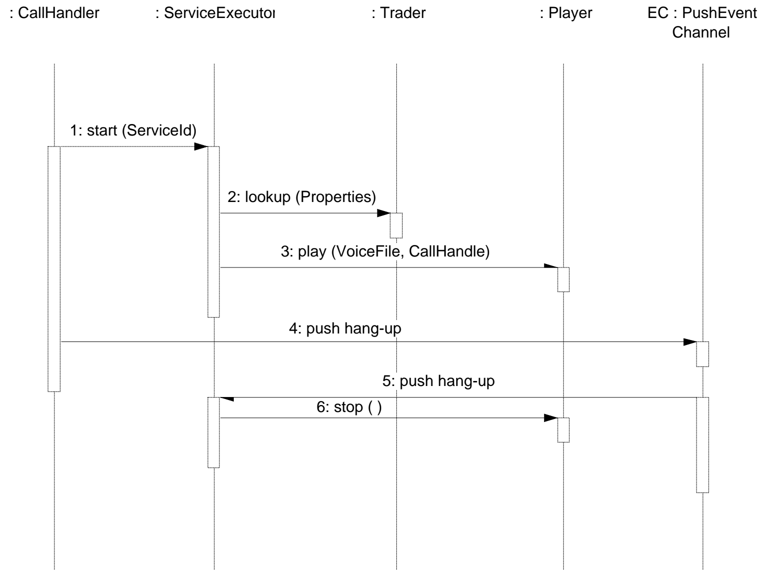


Figure 16 Scenario Diagram for an Incoming Call.

In this case the *Service Executor* calls the *Player* with a voice file and a call handle. While playing the voice file, the end-user hangs-up, which triggers the *CallHandler* to push a hang-up event. The event is received by the *Service Executor* instance that associates it with a particular service execution. The hang-up event causes the *Service Executor* to interrupt the playing of the message.

5.1.3 Reliability

In this section we discuss reliability and the reliability contracts associated with the phone service system. We do not providing a methodology for developing contracts; rather we illustrate one situation for which the dimensions we propose are useful. Neither do we consider tradeoffs between reliability and other QoS categories, such as performance nor, how the design affects the overall development cost.

To users of telephone services the *CallHandler* represents the actual execution service. Thus, to provide high-availability that *CallHandler* service must be highly available. The figure below summarizes the reliability contracts that we have defined. For simplicity we have omitted some dimensions such as those indicating the variance. The arrows are inferred from the class diagram and represent service dependencies.

The *CallHandler* relies on the *ServiceExecutor*, which in turn depends on the *Trader*, *EventService,* and the Resources. In the case of a failure services currently executing and their associated connections will be discontinued. This means among other things that the service executor need not recover its state when restarted. Users consider it more annoying if a session is interrupted due to a failure than if they are unable to connect to the service in the first place. We therefore wish the *ServiceExecutor* to be reliable in the sense that it functions adequately over the duration of a typical service call. In this fictitious system, calls are estimated to have a mean time of 3 minutes with 80% of the calls lasting less than 5 minutes.

Although the *ServiceExecutor* itself can recover rapidly and thereby provides high-availability, it depends on the *Trader* and resources for the whole system to function correctly. The *ServiceExecutor* can simply not execute services without a *Trader* from which it can get the resources it needs for a particular requested service.

A *Trader* is expected to have a state of significant size that needs to be recovered. First it needs to make sure the data on persistent storage is consistent and complete; then it needs to build the necessary memory structures for the efficient search and retrieval of resources. Therefore, the *Trader* has a longer recovery time than some of the other services. This means the mean-time-to-failure time must also be longer than for *Service Executor*. Failing resource services are expected to have long recovery times, which implies they need to have in principle infinite mean-time-to-failure to satisfy the availability requirements. This does not mean that individual resource can not fail, but it does mean that there must be enough redundancy that the service can mask such failures.

We are going to insist that all types of services be capable of execution when the system is up which, means that all types of resources must be available. The figure below shows reliability contracts as well as proposed dimensions

and dependencies. Observe that these represent client view contracts, which means that they specify the failures as seen by a client. As an example, players have a mean-time-to-failure of seven years and a mean-time-to-repair of 2 hours. This does not mean that we believe that one particular player will have a `MTTR` of seven years, but that we believe this must be true for the player service provided to its clients.

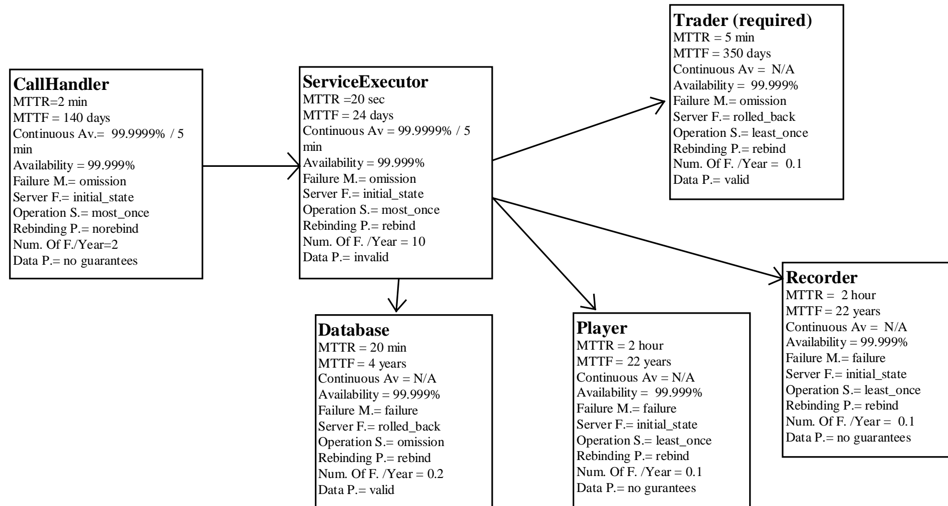


Figure 17 b Reliability Contracts

For the *CallHandler* to provide a highly available service it needs to rely on its components. We do not consider partial functionality; rather we are interested in the *PhoneServiceSystem* can providing full functionality with high reliability and availability. Thus, the requirements on *CallHandler* are propagated down to the resources and the *ServiceExecutor*. We have selected a solution for which we assume that resources and traders can be implemented reliably, thus relieving the *ServiceExecutor* can be relieved from some of the failure handling.

The *Trader* is required to recover in 5 minutes or less. It must be highly available since each execution of a service depends on getting resources from the trader. The *Trader* has a persistent storage of offered resources and in case of failure, it will be rolled back to the previous consistent state. Any references returned by the *Trader* before a *Trader* failure are valid still after the recovery. The *Trader* must be extremely stable and only fail once a year. This requirement is necessary so that the *CallHandler* may meet its reliability requirements.

Database resources must be highly available to ensure that the *Service Executor* can provide services. The mean recovery time is expected to be around 20 minutes, which will impose a very long mean-time-to-failure. If the service fails, clients are expected to rebind when the service has been restarted. The state of the database is also rolled back to the most recent consistent state. The contract also states that the service can show any omission failure but should never expose response failures.

The player and the recorder services represent resource pools that are used by the execution engines. To use a player or recorder, the *Service Executor* queries the *Trader* to get one that is available. Player and recorder services are expected to have a mean recovery time of up to two hours. Since the availability requirements are strict, these services must be very reliable. If there is a hardware or software failure, the service masks the failure from the *Service Executor*. However, if the service itself fails it can expose any type of failures.

Instances of *ServiceExecutor* have a short recovery time but do—as has been mentioned previously—lose existing service executions and their connections. We expect their continuous availability to be high for the average time period of five minutes. This means that if a service execution takes five minutes or less, the probability of a service execution failure should be small. After a failure, the *Service Executor* is restarted in a well-defined initial state. The contract also specifies that restarted service executors have new references; therefore clients must rebind. Finally, we note that data returned by the *Service Executor* would be invalidated if the *Service Executor* instance fails.

5.1.4 Discussion

We have presented one set of contracts that, if satisfied, makes the system highly available. We have not presented a methodology by which such contracts can be identified. Neither have we shown how the requirements on one service

can be used to identify necessary contracts for the components on which it relies. Rather we have illustrated that the dimension we propose can be used to understand what the reliability characteristics of a service. These characteristics will in turn guide the design, implementation, deployment, and management of systems using such services.

In order to characterize services defined with for example OMG IDL [CORBA], we will need more elaborate languages. They need to deal with the characterization on a detailed level (operations and arguments) and language concepts such as inheritance. In addition, they need to allow us to easily determine if one contract implies that another is also satisfied. One such situation occurs when we want to use an existing component that provides quality-of-service according to a well-defined contract. Clients are interested in whether the contract provided by the service indeed satisfies their requirements. The *QML* language described in [FRKO] is such a language.

6. Concluding Remarks

To understand, specify, and implement systems with a predictable reliability, we need to have a vocabulary set—dimensions— for expressing reliability contracts. The main goal of this paper is to present such a set of dimensions for reliability contracts based on what we have learnt from the characteristics of various reliability solutions and other related work. Our experience has been that proposing such a set of dimensions is quite controversial and spawns a great deal of discussion. Questions are often raised as to whether the set of lists is too narrow or too broad. Another question is whether the types of the dimensions are right. As we have mentioned in the paper, some dimensions can be more accurately described with statistical distributions for which we have proposed only mean and variance. Unfortunately, a distribution requires evidence as to what a realistic distribution would be. Another question that has been brought up is whether it makes sense to specify measures such as *mean-time-to-failure*, or *availability* since they can not be verified until the system is deployed.

We do not claim that the dimensions that are proposed represent a canonical set or that they have the most appropriate types for all situations. But we do believe—and our experiments indicate— that they constitute a quite complete and practical set of dimensions that are useful in the specification of distributed services. The example in Section 5illustrates how the dimensions can be used in the design of a system. The dimensions can clearly also be used more dynamically in mechanisms that negotiate quality-of-service agreements in runtime.

We have not seen any other proposals for any quality-of-service dimensions with the exception of performance characterizations in specific domains such as multi-media. We hope that this proposal will inspire continued work on reliability contracts and contracts for other quality-of-service dimensions. Such sets of dimensions are necessary for the continuing work on quality-of-service aware distributed systems.

Acknowledgments

This work has benefited largely from discussions with Svend Frølund on QoS specification languages. I also wish to thank Dean Thompson for his valuable input. My colleagues in the Object-Oriented Communication Infrastructure project at Hewlett-Packard Laboratories have also supplied valuable comments on earlier versions of this document. In addition, I have had several interesting discussion on reliability issues with Jay Kasi, Paul Harry, and Paul Murray. Finally, I would like to thank Mary Loomis and Patricia Markee for their very valuable and detailed comments.

References

[AVI97] Algirdas Avizienis. Toward Systematic Design of Fault Tolerant Systems. IEEE Computer, April 1997, Vol(30), No(4).

[BAL90] Henri Bal. Programming Distributed Systems. Prentice Hall, 1990.

[BABA86] Özalp Babaoglu. On the Reliability of Fault-Tolerant Distributed Computing Systems. Department of Computer Science, Cornell University, TR86-738, May 1986.

[BIRMAN89] Kenneth P. Birman. ISIS: A System for Fault-Tolerant Distributed Computing. Department of Computer Science, Cornell University. TR86-744, April 1986. URL http://www.cs.cornell.edu/Info/Faculty/Kenneth_Birman.html

[BIRMAN96] Kenneth P. Birman. Building Secure and Reliable Network Applications. Manning Publications Co. 1996.

[BOOCH] Grady Booch. Object-Oriented Analysis and Design. Benjamin-Cummings, 1988.

[BUD] Navin Budhiraja, Keith Marzullo, Fred B. Schenider, and Sam Toueg. The Primary-Backup Approach. In *Distributed Systems. Addison-Wesley*, Second Edition 1993.

[BUD92] Navin Budhiraja and Keith Marzullo. Highly-Available Services Using the Primary-Backup Approach. Second Workshop on the Management of Replicated Data. November, 1992. ISBN 0 8186 3170 8.

[CHOU] Timothy C. K. Chou. Beyond Fault Tolerance. IEEE Computer, April 1997, Vol(30), No(4).

[COU94] G. Coulouris, J. Dollimore, and Tim Kindberg. Distributed Systems: Concepts and Design. Second Edition. Addison-Wesley, 1994.

[CORBA] Common Object Request Broker Architecture. Object Management Group. Revision 2.0, July 1995.

[CRIS91] Flaviu Christian. Understanding Fault-Tolerant Distributed Systems. Communications of the ACM, February 1991, Vol(34), No(2).

[DCOM] Richard Grimes. Professional DCOM Programming. WROX Press, 1997.

[DOL95] Danny Dolev and Dalia Malki. The Design of the Transis Systems. Proceedings of Dagstuhl Workshop on Unifying Theory and Practice in Distributed Computing, September, 1995.

[FENTON92] Norman E. Fenton. Software Metrics: A Rigorous Approach. Chapman & Hall, 1992.

[FLEM95] R. A. Fleming. Secondary Sender. Hewlett-Packard Laboratories, June 1995.

[FRKO] Frølund and Koistinen. Quality of Service Specification in Distributed Object Systems Design. Hewlett-Packard Laboratories. August 1997.

[GOKHALE] S. S. Gokhale, Peter N. Marinos, and K. S. Trivedi. Important Milestones in Software Reliability Modeling. Technical Report, Duke University, Department of Electrical Engineering and Computer Engineering.

[GRAY] J. Gray and A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1993.

[GUE] Rachid Guerraoui and André Schiper. Software-Based Replication for Fault-Tolerance. IEEE Computer, April 1997, Vol(30), No(4).

[HARRY95] Paul Harry. An Introduction to Somersault. Hewlett-Packard Laboratories, December 1996.

[HSUEH97] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault Injection Techniques and Tools. IEEE Computer, April 1997, Vol(30), No(4).

[ISO/IEC] ISO/IEC JTC 1/SC 21 N QoS. Working Draft for Open Distributed Processing—Reference Model—Quality of Service. July, 1997.

[LANDIS] Sean Landis and Silvano Maffeis. Building Reliable Distributed Systems with CORBA.

[LITT91] Bev Littlewood. Software Reliability Modelling. In Software Engineer's Reference book. Section 31, Butterworth-Heinemann Ltd.,1991.

[MAFF95A] Silvano Maffeis. Run-Time Support for Object-Oriented Distributed Programming. Doctoral Dissertation, University of Zurich, 1995.

[MAFF95B] Silvano Maffeis. Adding Group Communication and Fault-Tolerance to CORBA. Proceedings of USENIX Conference on Object-Oriented Technologies. June 1995.

[MAFF97] Silvano Maffeis and Douglas C. Schmidt. Constructing Reliable Distributed Communication Systems with CORBA. IEEE Communications Magazine, Vol(14), No(2), February 1997.

[MAFF97B] Silvano Maffeis. PIRANHA: A Hunter of Crashed CORBA Objects.IEEE Computer, Vol(30), No(4), April, 1997.

[MISH93] S. Mishra, L. L. Peterson, and R. D. Schlichting. Consul : A Communication Sunstrate for Fault-Tolerant Distributed Programs. Technical Report, Department of Computer Science, The University of Arizona, 1993.

[MULL93] Sape Mullender (ed.). Distributed Systems. Second Edition, Addison-Wesley, 1993.

[Orbix+ISIS] Orbix+ISIS getting started tutorial. Isis Distributed Systems, Inc and Iona technologies, Ltd. Part Number D070, July 1995

[OMGTRAD] OMG Trader Service Specification. OMG Document number formal/97-07-26. July, 1997. http://www.omg.org/library/public-doclist.html

[POW94] D. Powell, P. Barrett, G. Bonn, M. Chérèque, D. Seaton, and P. Veríssimo. The Delta-4 Distributed Fault-Tolerant Architecture. In Readings in Distributed Systems, T. L. Casavant and M. Singhal, editors, IEEE Computer Society Press, 1994.

[REIB91] Andrew L. Reibman and Malathi Veeraraghavan. Reliability Modeling: An Overview for System Designers. IEEE Computer, April 1991.

[RESNICK] R. I. Resnick. A Modern Taxonomy of High-Availability. HTML page, March, 1997. (click to load local copy, some figures missing) URL : http://www.interlog.com/~resnick/HA.htm

[SCHN84] Fred B. Schneider. Byzantine Generals in Action: Implementing Fail-Stop Processors. ACM Transactions on Computer Systems, May 1984,Vol(2), No(2), Pages 145-154.

[SCHN90] Fred B. Schneider. Replication Management using the State-Machine Approach. ACM Computing Surveys, 1990, Vol(22).

[SHRIV] S. K. Shrivastava. Fault-Tolerant System Structuring Concepts. In Software Engineers Reference Book, Chapter 61.

[SLO94] Morris Sloman (ed.). Network and Distributed Systems Management. Addison-Wesley, 1994.

[SOMA97] Arun K. Somani and Nitin H. Vaidya. Understanding Fault-Tolerance and Reliability. IEEE Computer, Vol(30), No(4), April 1997.[SOM] Arun K. Somani. Understanding Fault-Tolerance and Reliability. IEEE Computer, April 1997, Vol(30), No(4).

[VEMU] Ranga Vemuri, Ram Mandayam, and Vijay Meduri. Performance Modeling Using PDL. IEEE Computer, August 1996.