

# Towards Distributed Applications Stability Engineering

Pankaj Garg, Svend Frolund, Allan Shepherd, Nalini Venkatasubramanian

Hewlett-Packard Laboratories  
1501 Page Mill Road  
Palo Alto, CA 94304  
garg@hpl.hp.com

**Abstract** - We develop concepts of a stable distributed application and present an engineering approach for the life cycle of such applications. The definition of stability is motivated by the need to address changes in the environment in which a distributed application is embedded—a network of computers, software environment, and user access patterns. Current approaches for distributed applications engineering do not adequately address the stability issues raised in this paper. Although there are approaches for reliability, performance engineering, and scalability, hitherto they have not been brought together under a perspective such as stability engineering. We illustrate the concepts of stability and stability engineering with several example distributed applications.

## I. INTRODUCTION

With the proliferation of personal computer workstations, the programming model of choice for large applications is to distribute them over several workstations. The design, implementation, deployment, and use of such distributed applications, however, poses several problems. We are particularly interested in the problems arising from the distribution of these applications. Some of the reasons for complexity due to distribution are [16]:

- *Asynchrony*. Different parts of the application can be executing with different speeds at the same time on different machines.
- *Time*. Global synchronization of clocks is difficult.
- *Delay*. There is a finite and non-determinate time delay associated with communication.

An effective configuration for the distributed application must be chosen initially and over a period of time. Initially an application must be configured to optimally utilize the network and hosts to provide appropriate service to its users. In a heterogeneous environment, where some hosts have more compute power than others, an optimal configuration for the application could be where compute intensive parts of the application reside on high-powered machines. If different links of the network have different performance characteristics, communication patterns of the application can make use of such

performance differences. Moreover, if the hardware or software changes, an application can be reconfigured to accommodate the changes.

To address the above issues of distributed applications, we have defined the term *stable distributed application* and the corresponding *stability engineering* of distributed applications. Roughly speaking, a distributed application is stable when it can provide an intended level of service over time, as the underlying hardware platforms, networks, and usage patterns change. The engineering processes that result in stable distributed applications collectively define stability engineering.

Existing programming tools and methodologies do not provide effective design rules and decision aids for stability engineering. Previously, designers of database systems [6] and file systems [7] have used techniques that resemble our approach for stability engineering. These techniques are, however, application specific. Moreover, users of distributed applications have little guidance to appropriately configure their application's use. We believe that a uniform approach is needed to address the stability issues across all life cycle stages, from requirements, design, implementation, deployment, to use. To be viable, such an approach should be applicable to a wide variety of distributed applications. In this paper we motivate the need for work in stability engineering of distributed applications, provide a preliminary definition of stability and an approach for achieving stability.

The rest of the paper is structured as follows: Section II defines aspects of stability of distributed applications using a Quality of Service (QoS) framework. Section III describes aspects of engineering the life cycle of a distributed application to achieve stability and to be assured of stability. In Section IV we survey work in distributed systems relevant to stability engineering. Finally, in Section V we conclude with directions for future work.

## II. DISTRIBUTED APPLICATIONS STABILITY

The stability of a distributed application is defined

with respect to the Quality of Service (QoS) that the application provides over a period of time. Typically, system QoS attributes are derived from high level user statements and specifications of what is desired. QoS is an emerging ISO standard that defines a reference framework for quantifying the desired behavior of a distributed system [14, 18]. Briefly, the framework defines three concepts:

- *Activity*. Aspects of the application to which measurements can be attached. For example processes, communications, and information flow.
- *Dimension*. Properties of activities that can be measured, for example, response time and mean time to failure.
- *Categories*. A set of related dimensions. For example, the reliability dimension could include mean time to failure and mean time to repair.

**Definition (Stable Distributed Application)** For a given network  $N$ , a user environment,  $U$ , and a distributed application  $A$ ,  $A$  is **stable** if the specified QoS of  $A$  remains within certain specified bounds as  $N$  and  $U$  evolve.

Being stable means that an application is not going to exhibit chaotic or catastrophic behavior when there are some perturbations in  $N$  or  $U$ .  $N$  and  $U$  may evolve as the system changes; stability engineering analyzes the impact of this evolution on some predefined QoS dimensions. The changes anticipated in a proposed deployment determines which stability mechanisms must be employed. We broadly classify  $N$  and  $U$  into three categories: *network topologies*, *software environment*, and *workload demands*. The network topology is the hardware nodes and interconnections on which the application relies for its communication and computing. The software environment is the interaction with other software systems, both the ones on which the application depends (such as the distribution infrastructure and the operating system) and the ones with which it co-exists on the distributed system and competes for resources. Finally, the workload is the demands that users of a distributed application impose on the application. In the rest of this section we elaborate further on each of these three broad factors and present examples to illustrate the stability issues.

## A. Network Topology

Distributed applications have to be stable with respect to dynamically changing system topologies where the underlying infrastructure can add or remove nodes and alter links. One of the most common changes in networks is the upgrading of networks, with either faster technology or addition of new nodes and links. With the current spur in the direction of multimedia technology, networks are rapidly being changed to accommodate more information per second [1]. With such rapid changes in networks, management of the networks is an important issue. Moreover, any management software for networks needs to be stable with respect to such changes over a period of time. Below we discuss the example of network management software for stability considerations.

### 1) Example: Enterprise Network Management

We consider the network of a large, geographically distributed, enterprise. Such networks interconnect a number of *devices* such as computers, printers, routers, etc., and typically have a hierarchical structure as depicted in Fig. 1. A number of local area networks (LAN's) each cover a site in the enterprise. These islands of LAN's are interconnected by a wide area network (WAN). Each LAN is itself a hierarchical entity consisting of a number of subnets, e.g. each department may have its own subnet. The hierarchical structure isolates local traffic.

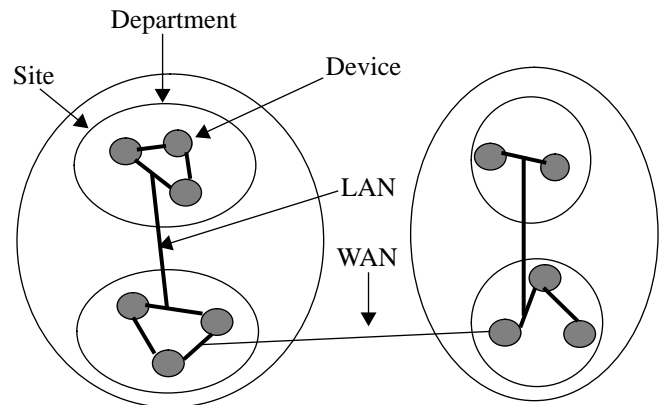


Fig. 1. The logical structure of an enterprise network

Since computers are becoming increasingly critical for the operation of enterprises, it is important that enterprise networks are *managed*. Management involves monitoring the operation of the network in order to detect device failures, determine device load, and detect link failures. Management also involves taking corrective actions when failures and overload occurs. For simplicity, we shall only consider the monitoring activity.

We assume that a *management system* communicates with the devices in the network in order to monitor the operation of the network. A device may consist of many different logical parts. For example, the same computer may be a file-server and a gateway at the same time. Because of this multi-faceted nature of devices, each device has a number of *interfaces* with which the management system communicates.

The SNMP protocol [19] is an evolving standard for the communication between a management system and the managed interfaces. With SNMP, the management system can poll interfaces for different values, or the management system can arrange for interfaces to generate *traps* if certain events occur. Polling can be used to periodically check the status (up or down) of interfaces, and to check the general condition such as load and configuration of interfaces. Polling can be initiated periodically by the management platform itself or it can be initiated directly by a human network operator in order to check the condition of suspiciously behaving devices. Traps are asynchronous communications generated by an interface in order to notify the management system that an important event has occurred. For example, a device can continuously monitor its own load and generate a trap if the load exceeds a certain threshold. Notice that the conditions to monitor are defined by the management system, not the interface itself.

We can characterize the behavior of a management system along certain quality of service dimensions:

- Number of lost traps
- Propagation time for traps
- Time for the periodic status check to detect and report a device failure
- Response time for operator initiated queries
- Consumed WAN bandwidth by the management system
- Consumed LAN bandwidth by the management system

These quality of service dimensions characterize the behavior of the management system both in terms of its delivered service and in terms of the demands it imposes on the environment in order to deliver this service. The environment of the network management system consists of the managed interfaces, the network infrastructure which allows communication with these interfaces, and the hardware on which the management system runs.

Stability is concerned with the continued delivery of sufficient quality of service despite changes in the environment. Because its environment is inherently changing and growing, stability is an important property of the management system: a management system must be able to deliver its intended functionality when the network size increases and the types of managed devices change; it is not enough that the management system can deliver a sufficient quality of service for a specific network.

## B. Software Environment

The software environment of an application is the software that co-exists with the application. It includes the software that the application depends on for different services, such as distributed name service and disk read and write. Also, an application typically co-exists with other application software that compete for the same resources, both hardware and software. Over time, the software environment of an application changes in a variety of ways:

- Dynamic invocation and revocation of application services, for example, a name service can be dynamically changed.
- Changes in the stability mechanisms employed to deal with varying QoS dimensions, for example, moving from a replication-based strategy to a migration-based strategy.
- Failure in the infrastructure software, where parts of the infrastructure may fail and still maintain some QoS for the application [3,4]

### 1) Example: Multimedia servers.

Consider the case of a media server that provides interactive video services to multiple clients across a WAN. Media servers scattered at different locations in the network provide video, image and online information access services to clients, as shown in Fig. 2.

An important QoS dimension in this situation is that a client gets unimpaired media service over extended time periods. Usually the client base for such services is so large that it is impractical and undesirable to interrupt service in order to perform software upgrades. Therefore, the stability issue is that even though the video transfer technology may change and need upgrading, the media server must continue to provide service to its clients.

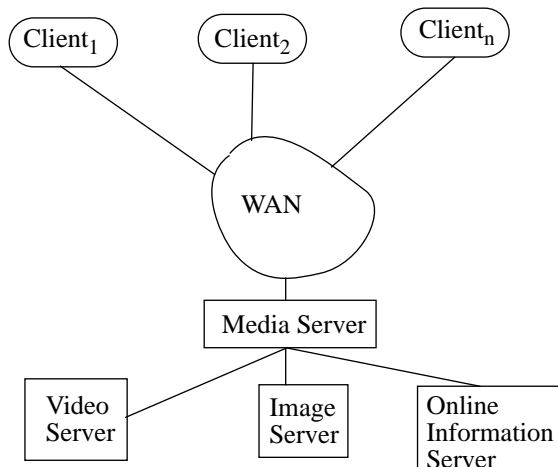


Fig. 2. A typical architecture for a media server.

### C. User Workload

User workload is a characterization of the demands that user tasks put on an application. Traditional single-user and single-host applications like editors, and spreadsheets usually have a single user generating their entire workload. In distributed applications, especially with multiuser applications, the demands on the application are diverse and can vary significantly over time as the user population can change over time. The variety of access patterns of a user population can be translated into quantifiable attributes of the application. For example, Carey et al. [6] classified the access patterns on their database system as one of four workload types:

- **HOTCOLD:** 80% of the access to specified region, 20% elsewhere
- **UNIFORM:** all over
- **HICON:** same access skew—more contention
- **PRIVATE:** the cold portion of HOTCOLD is read-only.

Usually applications are designed and deployed for one such user workload pattern, expecting a homogeneous usage environment. Stability considerations, however, may require several such load patterns for a single application as we expect changes in the work load patterns over a period of time.

#### 1) Example: License Servers.

As an example we consider the use of a distributed license server to illustrate the stability issues with varying user workload.

In a traditional mainframe environment, applications

ran on a single machine, possibly being shared by several users. With the shift towards personal computers, one model of application use is to consider each application running on a separate machine for a distinct user. This organization is unsatisfactory for enterprises and large groups of users of similar applications as each person in the group has to purchase an individual copy of the application. If there are  $N$  users, then  $N$  copies of the application have to be purchased. The problem is that at any given time, only  $n$  active users may actually be using the application, where  $n$  is usually much less than  $N$ , and is a function of time. The solution to this problem is to distinguish an application from a license to use the application. Then, even with a population of  $N$  users, an organization can choose to buy  $m$  licenses, where  $m$  is less than  $N$ .

One particular application that uses this mechanism is the FrameMaker wordprocessing application [8]. To provide this service, FrameMaker has a Floating License Server (FLS) that runs on a networked host to which clients can send requests for licenses, as shown in Fig. 3.

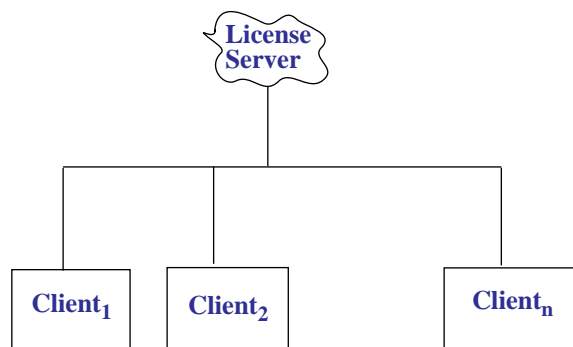


Fig. 3. FrameMaker license server for  $n(t)$  active clients out of  $N$  users.

When a user wants to use FrameMaker on his machine, he starts off a client process on his machine to request a license from the FLS. Depending on how many licenses the user's organization has purchased, and how many active users currently have licenses, a user's request for a license is either denied or acceded. As an additional optimization, the FLS may gather back licenses from users who have not actively used their licenses for some specified time interval.

An important Quality of Service issue for an FLS is that any user who wants a FrameMaker license should be able to get it within a few ( $\sim 2$ ) seconds. If a user cannot get a license within some short time, then the purpose of sharing the application among several users is defeated. With this QoS, an important deployment sta-

bility parameter for the FLS is, *how many licenses to purchase for a user population of  $N$  users*. With fewer licenses than needed, there will be frustrated users who cannot use the application when they want to. With more licenses than needed, more money than necessary is spent this application. This issue is not a one time decision: over a period of time we need to monitor the use of licenses, and the growth of the user population to understand the need for additional licenses.

### III. STABILITY ENGINEERING

Stability engineering brings together several properties of distributed systems that have been previously dealt with in isolation. Depending on the QoS specified by the end-user, any combination of the following aspects can be relevant.

- *Reliability*. Accuracy, timeliness and availability of results is required in applications with stringent QoS criteria. Given an understanding of the anticipated failures or load patterns, end-users need tools and techniques to understand the implications on the availability of their applications.
- *Performance*. Providing predictable and stable performance over time is challenging for distributed applications. It is important for the end-user to calculate the cost of design decisions and maximize resource utilization.
- *Scalability*. The ability to scale applications without loss of stability implies that we must design for scalable systems and introduce stability mechanisms that are scalable right from the early stages of the software's life cycle.

We would like to identify, isolate and encapsulate stability mechanisms at a layer independent of the application. This permits designers of multiple applications to invoke appropriate stability mechanisms to satisfy the QoS of a particular client request.

Providing stability involves providing predictable, cost-effective, reliable service given that the environment will change. We distinguish between building a stable distributed application and providing assurance to designers and users of an application of its stability. Our goal is to provide a framework for understanding and analyzing the concept of stability and design a methodology that will help us to predict and construct "stable" distributed systems.

#### A. Overheads of introducing stability

Stability, however, comes at a price. There is an obvious trade-off between system utilization, throughput and the stability of service that can be offered. It is the task of the system designer to create stability mechanisms to generate a cost-effective solution. As an example, there is a general trade-off between the system utilization and cost and the degree of assurance that can be provided. A general rule of thumb is that the most expensive resource/component of the system be well utilized. For compute-intensive applications, the CPU should be optimally utilized.

Typical cost-metrics to define and measure stability criteria need to be established. One such parameter is the maximum cost that can be incurred by the system in order to satisfy a particular user request. If the implementation of a stability solution exceeds this cost, it is rejected. This can lead to the definition of a stability calculus that captures the various trade-offs involved.

There are a number of factors that must be included in the cost of providing stability. Since failures will cause stability violation, the cost of failure detection, failure rollback, recovery and service renewal are cost factors. Furthermore, in order to provide uninterrupted service to client requests, there is an underlying state engine that maintains state information. This will introduce an additional overhead. To ensure guaranteed quality of service, every user request must pass an admissions control process. In order to ensure stability, the ability to monitor stability is critical.

Stable distributed software is a result of engineering for stability through every step of the software engineering life cycle. In the rest of this section, we describe the stages of the software life cycle and the implications of stability engineering on each stage.

#### B. Stability engineering life cycle

A typical software life cycle consists of many phases, e.g., see [11]. For the purpose of this work, we distinguish between the following main stages: requirements engineering, specification, architecture design, development, deployment, management, and audit. To realize stability engineering of distributed applications, we augment each of these stages with a functional model of the distributed application. Hence, we call our approach for stability engineering *model-based*. The model is functional in the sense that it is representative of the behavior of the application from the viewpoint of that life cycle

stage, and can be used to experiment with different parameters relating to the environment. We describe below the purpose and use of models in the different stages. We note that at different life cycle stages, even though the same model may not be applicable, the same abstract perspective on stability should be.

*Requirements and Specifications:* At the requirements and specifications stages, in addition to the traditional approaches, it is important to collect the stability characteristics for the proposed application.

*Architecture Design:* In this stage the high-level architecture of the distributed application needs to be understood [12]. This is an important stage for distributed applications, as major distribution decisions are made, and possibly frozen, at this stage. For stability engineering, we require building a model of the distributed application architecture, with each major component of the application as an entity in the model, and a simulation of the underlying network and usage patterns. The model can then be used to help understand whether the proposed architecture is stable with the given QoS requirements as the simulated network and usage patterns are varied.

*Development:* In the development stage, the model is used to influence key design and implementation decisions, e.g. timeout values, number of retries, and so forth.

*Deployment:* This determines a distributed application's stability, and is less significant in traditional single-machine applications. The introduction stated some problems of correctly deploying the application given a particular network. The model that was useful in the architectural design of the system could also be useful at this stage.

*Audit:* The environment of a distributed application is quite likely to change over time—a machine that was up once may be down now, a new high-performance server may be added to the network, and so forth. Therefore, the initial deployment of the application may or may not work over a period of time. For the purpose of adjusting the application to such changes, the application needs to be monitored and audited over a period of time. The data collected from such monitorings can be utilized by the model developed in the deployment stage to: (1) reconfigure the application as needed, and (2) validate the model with field data, instead of the simulated data used in the architectural design stage.

#### IV. RELATED WORK

A model-based approach has been applied for engineering of several distributed systems, e.g., file systems [7,17], and databases [6]. This approach, however, is not readily available to application programmers with tight deadlines and schedules. We want to build tools and libraries to make such engineering viable for application developers through stability engineering.

PROTOB is an object-oriented CASE tool for modeling and prototyping distributed systems [2]. The goals of PROTOB system are in some ways similar ours: To model a distributed application at a high-level and perform discrete event simulation on the model to understand different aspects of the system. There are, however, some high-level differences between our approaches, as described below.

The PROTOB approach assumes a development life cycle of modeling, emulation, and application generation. The issues of stability at the deployment and audit stages are, therefore not considered in PROTOB. This leads to a significant difference as the PROTOB method does not discuss the monitoring of a running application—a major focus for us. The PROTOB method does, however, discuss the use of monitoring of the model's emulation. This could possibly be extended to monitoring a running application.

The PROTOB approach was developed using primitive *send* and *receive* commands as the basis for application distribution. With the ever increasing popularity of the RPC model [5], it is not clear how applicable PROTOB will be to modern distributed applications.

Distributed systems management is emerging as a problem area for which technical solutions are being devised, e.g. see [15]. The goal for these approaches is to simplify the task of managing and deploying distributed applications. In so far as our models assist in these tasks, they are applicable to distributed systems management as well. The stability considerations, however, makes our approach quite different.

Several tools have been developed for the monitoring of distributed applications. Miller, Macrander, and Sechrest describe a tool for monitoring distributed applications in the Unix environment [16]. Their tool was built with the important goals of (1) transparency—measurement of events causes minimum perturbation in the way the event happens, and (2) consistency—the measurement model should be consistent with the programming model for distributed programming. Their measurement facility consists of three parts: (a) meter-

ing, (b) filtering, and (c) analysis. The metering of a process is done transparently by the operating system, based on user-specified event masks. Filtering is performed for the user by a filter process that can be tuned with high-level filtering specifications. The analysis routines are supplied by the user. We can make use of this kind of monitoring facility for ongoing assurance of stability.

A companion project at HP Labs is defining Distributed Measurement System (DMS), as a measurement architecture for distributed applications [9]. The design of DMS is driven by the need for a scalable measurement architecture. As such, DMS defines a hierarchical organization of sensors (instrumentation) at the distributed object level, observers at the process level, and collectors at the host level. As a proposed standard for DCE [10], DMS defines a set of standard sensor objects, which any application can choose from. We are collaborating with the DMS project to understand the measurement needs of stability engineering.

Z4/SIMPLE is a hybrid hardware and software monitoring approach for distributed programs [13]; Z4 is the hardware component that monitors hardware events, and SIMPLE is a suite of software tools to analyze the data generated from the monitoring. The hybrid approach is used in Z4/SIMPLE to do performance tuning and debugging of distributed and parallel application programs. In a hybrid approach the collection of monitoring data does not unduly compete for the same resources as the monitored program itself. At the same time the events of interest can be specified in a high-level problem-oriented manner. Each new application monitoring, however, does need both hardware and software design—hardware for identifying the events of interest and software for analyzing the results. In their future work section Hofmann et al. [13] propose using a model-based approach similar to ours. We expect that techniques for stability engineering may be built on top of systems like Z4/SIMPLE.

## V. FUTURE WORK

In this paper we have defined the key elements of stability of distributed applications and proposed a software engineering life cycle that provides stability engineering. These concepts open up several research directions that need further investigation:

- *Stability engineering life cycle of distributed applications.* In Section IV we have outlined an engineering life cycle that promises to achieve stability for dis-

tributed applications and platforms. Further work needs to be done to validate this life cycle and develop general concepts with wide applicability.

- *Lower layers instrumentation.* An important aspect of our approach for stability engineering is to collect data from an application for stability predictions. As discussed in the Related Work section, there are several approaches to collect performance data from a distributed applications.
- *Stability Prediction Technology.* The linchpin of our approach is to develop simulation models of distributed application behavior. The modeling technologies available today do not seem to be at a high enough level of abstraction for rapid deployment. We need high-level tools for rapid development of models.

Our group is actively pursuing research in these directions.

## REFERENCES

- [1] H. Armbruster and K. Wimmer. Broadband Multimedia Applications Using ATM Networks High-Performance Computing, High-Capacity Storage, and High-Speed Communication. *IEEE Journal Selected Areas in Communications*, pages 1382–1396, 1992.
- [2] M. Baldassari, G. Bruno, and A. Castella. PROTOB: an Object-oriented CASE Tool for Modelling and Prototyping Distributed Systems. *Software—Practice and Experience*, 21(8):823–844, August 1991.
- [3] K. Birman. Replication and Fault-Tolerance in the ISIS System. In *Proc. of the 10th ACM Symposium on Operating Systems Principles*, pages 79–86, December 1985.
- [4] K. Birman and T. A. Joseph. Reliable Communication in the Presence of Failures. In *Proc. of the 11th ACM Symposium on Operating Systems Principles*, pages 123–138, November 1987.
- [5] A. Birrel and B. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.
- [6] M. J. Carey, M. J. Franklin, and M. Zaharioudakis. Fine-Grained Sharing in a Page Server OODBMS. In R. T. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, volume 23 of *SIGMOD Record*, pages 359–370, Minneapolis, Minnesota, May 1994.
- [7] Maria R. Ebling and M. Satyanarayanan. SynRGen: An

- Extensible File Reference Generator. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, volume 22 of *Performance Evaluation Review*, pages 108–117, Nashville, Tennessee, May 1994.
- [8] Frame Technology Corporation, 1010 Rincon Circle, San Jose, CA 95131. *FrameMaker User Guide*.
- [9] R. Friedrich, J. Martinka, T. Sienknecht, and S. Saunders. Integration of Performance Measurement and Modelling for Open Distributed Processing. In *Proc. IFIP International Conference on Open Dist. Processing 95*, 1995.
- [10] R. Friedrich, S. Saunders, G. Zaidenweber, D. Bachmann, and S. Blumson. Standardized Performance Instrumentation and Interface Specification for Monitoring DCE Based Applications. OSF DCE Request for Comment: 33.0, December 1994.
- [11] P. K. Garg and W. Scacchi. A Hypertext System to Manage Software Life Cycle Documents. *IEEE Software*, pages 90–98, May 1990.
- [12] David Garlan and Mary Shaw. An Introduction to Software Architecture. Technical Report CMU/SEI-94-TR-21, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890, 1994.
- [13] R. Hofmann, R. Klar, B. Mohr, A. Quick, and M. Siegle. Distributed Performance Monitoring: Methods, Tools, and Applications. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):585–598, June 1994.
- [14] David Hutchison, Geoff Coulson, Andrew Cambell, and Gordon S. Blair. *Network and Distributed Systems Management*, chapter Quality of Service Management in Distributed Systems, pages 273–302. Addison-Wesley, 1994.
- [15] Keith Marzullo, Robert Cooper, Mark D. Wood, and Kenneth P. Birman. Tools for Distributed Application Management. In Thomas Casavant and Mukesh Singhal, editors, *Readings in Distributed Computing Systems*, pages 310–345. IEEE Computer Society, 1994.
- [16] Barton P. Miller, C. Macrander, and S. Sechrest. A Distributed Programs Monitor for Berkeley UNIX. *Software-Practice and Experience*, 16(2):183–200, February 1986.
- [17] Brian D. Noble and M. Satyanarayanan. An Empirical Study of a Highly Available File System. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, volume 22 of *Performance Evaluation Review*, pages 138–149, Nashville, Tennessee, May 1994.
- [18] OSI. Quality of Service Framework - Outline. ISO/IEC JTC1/SC21/WG1 N1145.
- [19] W. Stallings. Simple Network Management Protocol. In M. Sloman, editor, *Network and Distributed Systems Management*. Addison-Wesley, March 1994.

# Towards Distributed Applications Stability Engineering

Pankaj Garg, Svend Frolund, Allan Shepherd, Nalini Venkatasubramanian

Hewlett-Packard Laboratories  
1501 Page Mill Road  
Palo Alto, CA 94304  
garg@hpl.hp.com

**Abstract** - We develop concepts of a stable distributed application and present an engineering approach for the life cycle of such applications. The definition of stability is motivated by the need to address changes in the environment in which a distributed application is embedded—a network of computers, software environment, and user access patterns. Current approaches for distributed applications engineering do not adequately address the stability issues raised in this paper. Although there are approaches for reliability, performance engineering, and scalability, hitherto they have not been brought together under a perspective such as stability engineering. We illustrate the concepts of stability and stability engineering with several example distributed applications.

## I. INTRODUCTION

With the proliferation of personal computer workstations, the programming model of choice for large applications is to distribute them over several workstations. The design, implementation, deployment, and use of such distributed applications, however, poses several problems. We are particularly interested in the problems arising from the distribution of these applications. Some of the reasons for complexity due to distribution are [16]:

- *Asynchrony*. Different parts of the application can be executing with different speeds at the same time on different machines.
- *Time*. Global synchronization of clocks is difficult.
- *Delay*. There is a finite and non-determinate time delay associated with communication.

An effective configuration for the distributed application must be chosen initially and over a period of time. Initially an application must be configured to optimally utilize the network and hosts to provide appropriate service to its users. In a heterogeneous environment, where some hosts have more compute power than others, an optimal configuration for the application could be where compute intensive parts of the application reside on high-powered machines. If different links of the network have different performance characteristics, communication patterns of the application can make use of such

performance differences. Moreover, if the hardware or software changes, an application can be reconfigured to accommodate the changes.

To address the above issues of distributed applications, we have defined the term *stable distributed application* and the corresponding *stability engineering* of distributed applications. Roughly speaking, a distributed application is stable when it can provide an intended level of service over time, as the underlying hardware platforms, networks, and usage patterns change. The engineering processes that result in stable distributed applications collectively define stability engineering.

Existing programming tools and methodologies do not provide effective design rules and decision aids for stability engineering. Previously, designers of database systems [6] and file systems [7] have used techniques that resemble our approach for stability engineering. These techniques are, however, application specific. Moreover, users of distributed applications have little guidance to appropriately configure their application's use. We believe that a uniform approach is needed to address the stability issues across all life cycle stages, from requirements, design, implementation, deployment, to use. To be viable, such an approach should be applicable to a wide variety of distributed applications. In this paper we motivate the need for work in stability engineering of distributed applications, provide a preliminary definition of stability and an approach for achieving stability.

The rest of the paper is structured as follows: Section II defines aspects of stability of distributed applications using a Quality of Service (QoS) framework. Section III describes aspects of engineering the life cycle of a distributed application to achieve stability and to be assured of stability. In Section IV we survey work in distributed systems relevant to stability engineering. Finally, in Section V we conclude with directions for future work.

## II. DISTRIBUTED APPLICATIONS STABILITY

The stability of a distributed application is defined

with respect to the Quality of Service (QoS) that the application provides over a period of time. Typically, system QoS attributes are derived from high level user statements and specifications of what is desired. QoS is an emerging ISO standard that defines a reference framework for quantifying the desired behavior of a distributed system [14, 18]. Briefly, the framework defines three concepts:

- *Activity.* Aspects of the application to which measurements can be attached. For example processes, communications, and information flow.
- *Dimension.* Properties of activities that can be measured, for example, response time and mean time to failure.
- *Categories.* A set of related dimensions. For example, the reliability dimension could include mean time to failure and mean time to repair.

**Definition (Stable Distributed Application)** For a given network  $N$ , a user environment,  $U$ , and a distributed application  $A$ ,  $A$  is **stable** if the specified QoS of  $A$  remains within certain specified bounds as  $N$  and  $U$  evolve.

Being stable means that an application is not going to exhibit chaotic or catastrophic behavior when there are some perturbations in  $N$  or  $U$ .  $N$  and  $U$  may evolve as the system changes; stability engineering analyzes the impact of this evolution on some predefined QoS dimensions. The changes anticipated in a proposed deployment determines which stability mechanisms must be employed. We broadly classify  $N$  and  $U$  into three categories: *network topologies*, *software environment*, and *workload demands*. The network topology is the hardware nodes and interconnections on which the application relies for its communication and computing. The software environment is the interaction with other software systems, both the ones on which the application depends (such as the distribution infrastructure and the operating system) and the ones with which it co-exists on the distributed system and competes for resources. Finally, the workload is the demands that users of a distributed application impose on the application. In the rest of this section we elaborate further on each of these three broad factors and present examples to illustrate the stability issues.

## A. Network Topology

Distributed applications have to be stable with respect to dynamically changing system topologies where the underlying infrastructure can add or remove nodes and alter links. One of the most common changes in networks is the upgrading of networks, with either faster technology or addition of new nodes and links. With the current spur in the direction of multimedia technology, networks are rapidly being changed to accommodate more information per second [1]. With such rapid changes in networks, management of the networks is an important issue. Moreover, any management software for networks needs to be stable with respect to such changes over a period of time. Below we discuss the example of network management software for stability considerations.

### 1) Example: Enterprise Network Management

We consider the network of a large, geographically distributed, enterprise. Such networks interconnect a number of *devices* such as computers, printers, routers, etc., and typically have a hierarchical structure as depicted in Fig. 1. A number of local area networks (LAN's) each cover a site in the enterprise. These islands of LAN's are interconnected by a wide area network (WAN). Each LAN is itself a hierarchical entity consisting of a number of subnets, e.g. each department may have its own subnet. The hierarchical structure isolates local traffic.

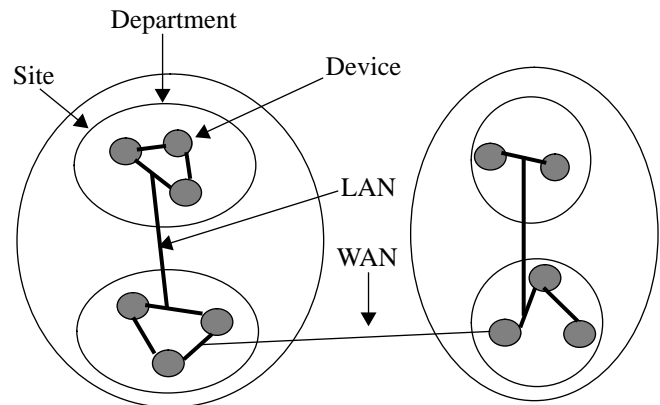


Fig. 1. The logical structure of an enterprise network

Since computers are becoming increasingly critical for the operation of enterprises, it is important that enterprise networks are *managed*. Management involves monitoring the operation of the network in order to detect device failures, determine device load, and detect link failures. Management also involves taking corrective actions when failures and overload occurs. For simplicity, we shall only consider the monitoring activity.

We assume that a *management system* communicates with the devices in the network in order to monitor the operation of the network. A device may consist of many different logical parts. For example, the same computer may be a file-server and a gateway at the same time. Because of this multi-faceted nature of devices, each device has a number of *interfaces* with which the management system communicates.

The SNMP protocol [19] is an evolving standard for the communication between a management system and the managed interfaces. With SNMP, the management system can poll interfaces for different values, or the management system can arrange for interfaces to generate *traps* if certain events occur. Polling can be used to periodically check the status (up or down) of interfaces, and to check the general condition such as load and configuration of interfaces. Polling can be initiated periodically by the management platform itself or it can be initiated directly by a human network operator in order to check the condition of suspiciously behaving devices. Traps are asynchronous communications generated by an interface in order to notify the management system that an important event has occurred. For example, a device can continuously monitor its own load and generate a trap if the load exceeds a certain threshold. Notice that the conditions to monitor are defined by the management system, not the interface itself.

We can characterize the behavior of a management system along certain quality of service dimensions:

- Number of lost traps
- Propagation time for traps
- Time for the periodic status check to detect and report a device failure
- Response time for operator initiated queries
- Consumed WAN bandwidth by the management system
- Consumed LAN bandwidth by the management system

These quality of service dimensions characterize the behavior of the management system both in terms of its delivered service and in terms of the demands it imposes on the environment in order to deliver this service. The environment of the network management system consists of the managed interfaces, the network infrastructure which allows communication with these interfaces, and the hardware on which the management system runs.

Stability is concerned with the continued delivery of sufficient quality of service despite changes in the environment. Because its environment is inherently changing and growing, stability is an important property of the management system: a management system must be able to deliver its intended functionality when the network size increases and the types of managed devices change; it is not enough that the management system can deliver a sufficient quality of service for a specific network.

## B. Software Environment

The software environment of an application is the software that co-exists with the application. It includes the software that the application depends on for different services, such as distributed name service and disk read and write. Also, an application typically co-exists with other application software that compete for the same resources, both hardware and software. Over time, the software environment of an application changes in a variety of ways:

- Dynamic invocation and revocation of application services, for example, a name service can be dynamically changed.
- Changes in the stability mechanisms employed to deal with varying QoS dimensions, for example, moving from a replication-based strategy to a migration-based strategy.
- Failure in the infrastructure software, where parts of the infrastructure may fail and still maintain some QoS for the application [3,4]

### 1) Example: Multimedia servers.

Consider the case of a media server that provides interactive video services to multiple clients across a WAN. Media servers scattered at different locations in the network provide video, image and online information access services to clients, as shown in Fig. 2.

An important QoS dimension in this situation is that a client gets unimpaired media service over extended time periods. Usually the client base for such services is so large that it is impractical and undesirable to interrupt service in order to perform software upgrades. Therefore, the stability issue is that even though the video transfer technology may change and need upgrading, the media server must continue to provide service to its clients.

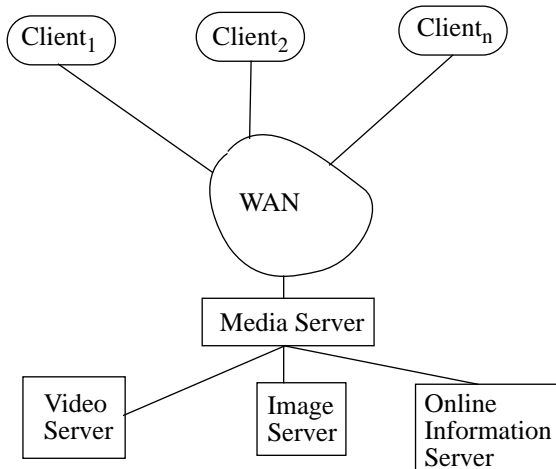


Fig. 2. A typical architecture for a media server.

### C. User Workload

User workload is a characterization of the demands that user tasks put on an application. Traditional single-user and single-host applications like editors, and spreadsheets usually have a single user generating their entire workload. In distributed applications, especially with multiuser applications, the demands on the application are diverse and can vary significantly over time as the user population can change over time. The variety of access patterns of a user population can be translated into quantifiable attributes of the application. For example, Carey et al. [6] classified the access patterns on their database system as one of four workload types:

- **HOTCOLD:** 80% of the access to specified region, 20% elsewhere
- **UNIFORM:** all over
- **HICON:** same access skew—more contention
- **PRIVATE:** the cold portion of HOTCOLD is read-only.

Usually applications are designed and deployed for one such user workload pattern, expecting a homogeneous usage environment. Stability considerations, however, may require several such load patterns for a single application as we expect changes in the work load patterns over a period of time.

#### 1) Example: License Servers.

As an example we consider the use of a distributed license server to illustrate the stability issues with varying user workload.

In a traditional mainframe environment, applications

ran on a single machine, possibly being shared by several users. With the shift towards personal computers, one model of application use is to consider each application running on a separate machine for a distinct user. This organization is unsatisfactory for enterprises and large groups of users of similar applications as each person in the group has to purchase an individual copy of the application. If there are  $N$  users, then  $N$  copies of the application have to be purchased. The problem is that at any given time, only  $n$  active users may actually be using the application, where  $n$  is usually much less than  $N$ , and is a function of time. The solution to this problem is to distinguish an application from a license to use the application. Then, even with a population of  $N$  users, an organization can choose to buy  $m$  licenses, where  $m$  is less than  $N$ .

One particular application that uses this mechanism is the FrameMaker wordprocessing application [8]. To provide this service, FrameMaker has a Floating License Server (FLS) that runs on a networked host to which clients can send requests for licenses, as shown in Fig. 3.

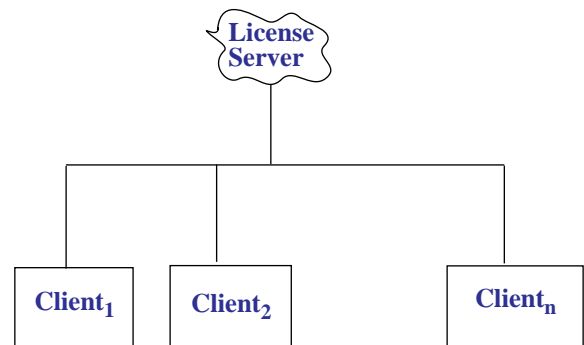


Fig. 3. FrameMaker license server for  $n(t)$  active clients out of  $N$  users.

When a user wants to use FrameMaker on his machine, he starts off a client process on his machine to request a license from the FLS. Depending on how many licenses the user's organization has purchased, and how many active users currently have licenses, a user's request for a license is either denied or acceded. As an additional optimization, the FLS may gather back licenses from users who have not actively used their licenses for some specified time interval.

An important Quality of Service issue for an FLS is that any user who wants a FrameMaker license should be able to get it within a few ( $\sim 2$ ) seconds. If a user cannot get a license within some short time, then the purpose of sharing the application among several users is defeated. With this QoS, an important deployment sta-

bility parameter for the FLS is, *how many licenses to purchase for a user population of  $N$  users*. With fewer licenses than needed, there will be frustrated users who cannot use the application when they want to. With more licenses than needed, more money than necessary is spent this application. This issue is not a one time decision: over a period of time we need to monitor the use of licenses, and the growth of the user population to understand the need for additional licenses.

### III. STABILITY ENGINEERING

Stability engineering brings together several properties of distributed systems that have been previously dealt with in isolation. Depending on the QoS specified by the end-user, any combination of the following aspects can be relevant.

- *Reliability*. Accuracy, timeliness and availability of results is required in applications with stringent QoS criteria. Given an understanding of the anticipated failures or load patterns, end-users need tools and techniques to understand the implications on the availability of their applications.
- *Performance*. Providing predictable and stable performance over time is challenging for distributed applications. It is important for the end-user to calculate the cost of design decisions and maximize resource utilization.
- *Scalability*. The ability to scale applications without loss of stability implies that we must design for scalable systems and introduce stability mechanisms that are scalable right from the early stages of the software's life cycle.

We would like to identify, isolate and encapsulate stability mechanisms at a layer independent of the application. This permits designers of multiple applications to invoke appropriate stability mechanisms to satisfy the QoS of a particular client request.

Providing stability involves providing predictable, cost-effective, reliable service given that the environment will change. We distinguish between building a stable distributed application and providing assurance to designers and users of an application of its stability. Our goal is to provide a framework for understanding and analyzing the concept of stability and design a methodology that will help us to predict and construct "stable" distributed systems.

#### A. Overheads of introducing stability

Stability, however, comes at a price. There is an obvious trade-off between system utilization, throughput and the stability of service that can be offered. It is the task of the system designer to create stability mechanisms to generate a cost-effective solution. As an example, there is a general trade-off between the system utilization and cost and the degree of assurance that can be provided. A general rule of thumb is that the most expensive resource/component of the system be well utilized. For compute-intensive applications, the CPU should be optimally utilized.

Typical cost-metrics to define and measure stability criteria need to be established. One such parameter is the maximum cost that can be incurred by the system in order to satisfy a particular user request. If the implementation of a stability solution exceeds this cost, it is rejected. This can lead to the definition of a stability calculus that captures the various trade-offs involved.

There are a number of factors that must be included in the cost of providing stability. Since failures will cause stability violation, the cost of failure detection, failure rollback, recovery and service renewal are cost factors. Furthermore, in order to provide uninterrupted service to client requests, there is an underlying state engine that maintains state information. This will introduce an additional overhead. To ensure guaranteed quality of service, every user request must pass an admissions control process. In order to ensure stability, the ability to monitor stability is critical.

Stable distributed software is a result of engineering for stability through every step of the software engineering life cycle. In the rest of this section, we describe the stages of the software life cycle and the implications of stability engineering on each stage.

#### B. Stability engineering life cycle

A typical software life cycle consists of many phases, e.g., see [11]. For the purpose of this work, we distinguish between the following main stages: requirements engineering, specification, architecture design, development, deployment, management, and audit. To realize stability engineering of distributed applications, we augment each of these stages with a functional model of the distributed application. Hence, we call our approach for stability engineering *model-based*. The model is functional in the sense that it is representative of the behavior of the application from the viewpoint of that life cycle

stage, and can be used to experiment with different parameters relating to the environment. We describe below the purpose and use of models in the different stages. We note that at different life cycle stages, even though the same model may not be applicable, the same abstract perspective on stability should be.

*Requirements and Specifications:* At the requirements and specifications stages, in addition to the traditional approaches, it is important to collect the stability characteristics for the proposed application.

*Architecture Design:* In this stage the high-level architecture of the distributed application needs to be understood [12]. This is an important stage for distributed applications, as major distribution decisions are made, and possibly frozen, at this stage. For stability engineering, we require building a model of the distributed application architecture, with each major component of the application as an entity in the model, and a simulation of the underlying network and usage patterns. The model can then be used to help understand whether the proposed architecture is stable with the given QoS requirements as the simulated network and usage patterns are varied.

*Development:* In the development stage, the model is used to influence key design and implementation decisions, e.g. timeout values, number of retries, and so forth.

*Deployment:* This determines a distributed application's stability, and is less significant in traditional single-machine applications. The introduction stated some problems of correctly deploying the application given a particular network. The model that was useful in the architectural design of the system could also be useful at this stage.

*Audit:* The environment of a distributed application is quite likely to change over time—a machine that was up once may be down now, a new high-performance server may be added to the network, and so forth. Therefore, the initial deployment of the application may or may not work over a period of time. For the purpose of adjusting the application to such changes, the application needs to be monitored and audited over a period of time. The data collected from such monitorings can be utilized by the model developed in the deployment stage to: (1) reconfigure the application as needed, and (2) validate the model with field data, instead of the simulated data used in the architectural design stage.

#### IV. RELATED WORK

A model-based approach has been applied for engineering of several distributed systems, e.g., file systems [7,17], and databases [6]. This approach, however, is not readily available to application programmers with tight deadlines and schedules. We want to build tools and libraries to make such engineering viable for application developers through stability engineering.

PROTOB is an object-oriented CASE tool for modeling and prototyping distributed systems [2]. The goals of PROTOB system are in some ways similar ours: To model a distributed application at a high-level and perform discrete event simulation on the model to understand different aspects of the system. There are, however, some high-level differences between our approaches, as described below.

The PROTOB approach assumes a development life cycle of modeling, emulation, and application generation. The issues of stability at the deployment and audit stages are, therefore not considered in PROTOB. This leads to a significant difference as the PROTOB method does not discuss the monitoring of a running application—a major focus for us. The PROTOB method does, however, discuss the use of monitoring of the model's emulation. This could possibly be extended to monitoring a running application.

The PROTOB approach was developed using primitive *send* and *receive* commands as the basis for application distribution. With the ever increasing popularity of the RPC model [5], it is not clear how applicable PROTOB will be to modern distributed applications.

Distributed systems management is emerging as a problem area for which technical solutions are being devised, e.g. see [15]. The goal for these approaches is to simplify the task of managing and deploying distributed applications. In so far as our models assist in these tasks, they are applicable to distributed systems management as well. The stability considerations, however, makes our approach quite different.

Several tools have been developed for the monitoring of distributed applications. Miller, Macrander, and Sechrest describe a tool for monitoring distributed applications in the Unix environment [16]. Their tool was built with the important goals of (1) transparency—measurement of events causes minimum perturbation in the way the event happens, and (2) consistency—the measurement model should be consistent with the programming model for distributed programming. Their measurement facility consists of three parts: (a) meter-

ing, (b) filtering, and (c) analysis. The metering of a process is done transparently by the operating system, based on user-specified event masks. Filtering is performed for the user by a filter process that can be tuned with high-level filtering specifications. The analysis routines are supplied by the user. We can make use of this kind of monitoring facility for ongoing assurance of stability.

A companion project at HP Labs is defining Distributed Measurement System (DMS), as a measurement architecture for distributed applications [9]. The design of DMS is driven by the need for a scalable measurement architecture. As such, DMS defines a hierarchical organization of sensors (instrumentation) at the distributed object level, observers at the process level, and collectors at the host level. As a proposed standard for DCE [10], DMS defines a set of standard sensor objects, which any application can choose from. We are collaborating with the DMS project to understand the measurement needs of stability engineering.

Z4/SIMPLE is a hybrid hardware and software monitoring approach for distributed programs [13]; Z4 is the hardware component that monitors hardware events, and SIMPLE is a suite of software tools to analyze the data generated from the monitoring. The hybrid approach is used in Z4/SIMPLE to do performance tuning and debugging of distributed and parallel application programs. In a hybrid approach the collection of monitoring data does not unduly compete for the same resources as the monitored program itself. At the same time the events of interest can be specified in a high-level problem-oriented manner. Each new application monitoring, however, does need both hardware and software design—hardware for identifying the events of interest and software for analyzing the results. In their future work section Hofmann et al. [13] propose using a model-based approach similar to ours. We expect that techniques for stability engineering may be built on top of systems like Z4/SIMPLE.

## V. FUTURE WORK

In this paper we have defined the key elements of stability of distributed applications and proposed a software engineering life cycle that provides stability engineering. These concepts open up several research directions that need further investigation:

- *Stability engineering life cycle of distributed applications.* In Section IV we have outlined an engineering life cycle that promises to achieve stability for dis-

tributed applications and platforms. Further work needs to be done to validate this life cycle and develop general concepts with wide applicability.

- *Lower layers instrumentation.* An important aspect of our approach for stability engineering is to collect data from an application for stability predictions. As discussed in the Related Work section, there are several approaches to collect performance data from a distributed applications.
- *Stability Prediction Technology.* The linchpin of our approach is to develop simulation models of distributed application behavior. The modeling technologies available today do not seem to be at a high enough level of abstraction for rapid deployment. We need high-level tools for rapid development of models.

Our group is actively pursuing research in these directions.

## REFERENCES

- [1] H. Armbruster and K. Wimmer. Broadband Multimedia Applications Using ATM Networks High-Performance Computing, High-Capacity Storage, and High-Speed Communication. *IEEE Journal Selected Areas in Communications*, pages 1382–1396, 1992.
- [2] M. Baldassari, G. Bruno, and A. Castella. PROTOB: an Object-oriented CASE Tool for Modelling and Prototyping Distributed Systems. *Software—Practice and Experience*, 21(8):823–844, August 1991.
- [3] K. Birman. Replication and Fault-Tolerance in the ISIS System. In *Proc. of the 10th ACM Symposium on Operating Systems Principles*, pages 79–86, December 1985.
- [4] K. Birman and T. A. Joseph. Reliable Communication in the Presence of Failures. In *Proc. of the 11th ACM Symposium on Operating Systems Principles*, pages 123–138, November 1987.
- [5] A. Birrel and B. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.
- [6] M. J. Carey, M. J. Franklin, and M. Zaharioudakis. Fine-Grained Sharing in a Page Server OODBMS. In R. T. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, volume 23 of *SIGMOD Record*, pages 359–370, Minneapolis, Minnesota, May 1994.
- [7] Maria R. Ebling and M. Satyanarayanan. SynRGen: An

- Extensible File Reference Generator. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, volume 22 of *Performance Evaluation Review*, pages 108–117, Nashville, Tennessee, May 1994.
- [8] Frame Technology Corporation, 1010 Rincon Circle, San Jose, CA 95131. *FrameMaker User Guide*.
- [9] R. Friedrich, J. Martinka, T. Sienknecht, and S. Saunders. Integration of Performance Measurement and Modelling for Open Distributed Processing. In *Proc. IFIP International Conference on Open Dist. Processing 95*, 1995.
- [10] R. Friedrich, S. Saunders, G. Zaidenweber, D. Bachmann, and S. Blumson. Standardized Performance Instrumentation and Interface Specification for Monitoring DCE Based Applications. OSF DCE Request for Comment: 33.0, December 1994.
- [11] P. K. Garg and W. Scacchi. A Hypertext System to Manage Software Life Cycle Documents. *IEEE Software*, pages 90–98, May 1990.
- [12] David Garlan and Mary Shaw. An Introduction to Software Architecture. Technical Report CMU/SEI-94-TR-21, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890, 1994.
- [13] R. Hofmann, R. Klar, B. Mohr, A. Quick, and M. Siegle. Distributed Performance Monitoring: Methods, Tools, and Applications. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):585–598, June 1994.
- [14] David Hutchison, Geoff Coulson, Andrew Cambell, and Gordon S. Blair. *Network and Distributed Systems Management*, chapter Quality of Service Management in Distributed Systems, pages 273–302. Addison-Wesley, 1994.
- [15] Keith Marzullo, Robert Cooper, Mark D. Wood, and Kenneth P. Birman. Tools for Distributed Application Management. In Thomas Casavant and Mukesh Singhal, editors, *Readings in Distributed Computing Systems*, pages 310–345. IEEE Computer Society, 1994.
- [16] Barton P. Miller, C. Macrander, and S. Sechrest. A Distributed Programs Monitor for Berkeley UNIX. *Software-Practice and Experience*, 16(2):183–200, February 1986.
- [17] Brian D. Noble and M. Satyanarayanan. An Empirical Study of a Highly Available File System. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, volume 22 of *Performance Evaluation Review*, pages 138–149, Nashville, Tennessee, May 1994.
- [18] OSI. Quality of Service Framework - Outline. ISO/IEC JTC1/SC21/WG1 N1145.
- [19] W. Stallings. Simple Network Management Protocol. In M. Sloman, editor, *Network and Distributed Systems Management*. Addison-Wesley, March 1994.