An Architecture for Versatile Dependability

Tudor Dumitraş and Priya Narasimhan*
Carnegie Mellon University
Pittsburgh, PA 15213
USA
tdumitra@ece.cmu.edu priya@cs.cmu.edu

Abstract

Versatile dependability is a software architecture that aims to quantify the trade-offs among a system's fault-tolerance, performance and resource requirements, and that provides system-level "knobs" for tuning these trade-offs. We propose a visual representation of the trade-off space of dependable systems, and discuss design principles that allow the tuning of such trade-offs along multiple dimensions. Through a case study on tuning system scalability under latency, bandwidth and fault-tolerance constraints, we demonstrate how our approach covers an extended region of the dependability design space.

1. Introduction

Oftentimes, the requirements of a dependable system are conflicting in many ways. For example, optimizations for high performance usually come at the expense of using additional resources and/or weakening the fault-tolerance guarantees. It is our belief that these conflicts must be viewed as *trade-offs* in the design space of dependable systems and that only a good understanding of these trade-offs can lead to the development of useful and reliable systems. Unfortunately, existing approaches offer only point solutions to this problem because they hard-code the trade-offs in their design choices, rendering them difficult to adapt to changing working conditions and to support evolving requirements over the system's lifetime.

As an alternative, we propose *versatile dependability*, a novel design paradigm for dependable distributed systems that focuses on the three-way trade-off between fault-tolerance, quality of service (QoS) – in terms of performance or real-time guarantees – and resource usage. This framework offers a better coverage of the dependability design-space, by focusing on an operating region (rather than an operating point) within this space, and by providing a set of "knobs" for tuning the trade-offs and properties of the system. This paper makes three main contributions:

- A new concept, versatile dependability, directed at achieving tunable, resource and QoS aware fault-tolerance in distributed systems (Section 2);
- A software architecture for versatile dependability with four design goals: tunability, quantifiability, transparency and ease of use (Section 3);
- A case study on using our architecture to tune an important system-level property, scalability (Section 4).

2. Versatile Dependability

We visualize the development of dependable systems through a three-dimensional *dependability design-space*, as shown in Figure 1, with the following axes: (i) the *fault-tolerance* "levels" that the system can provide, (ii) the *high performance* guarantees it can offer, and (iii) the amount of *resources* it needs for each pairwise {fault-tolerance, performance} choice. In contrast to existing dependable systems, we aim to span larger regions of this space because the behavior of the application can be tuned by adjusting the appropriate settings. We evaluate the wide variety of choices for implementing dependable systems, and we quantify the effect of these choices on the three axes of our {Fault-Tolerance $\times$ Performance $\times$ Resources} design space. From this data, we can extract the interdependencies among the three conflicting properties, and we can

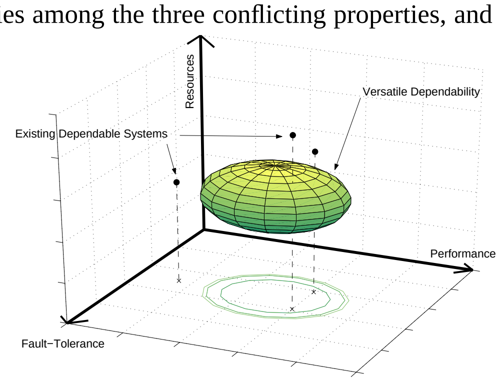

Figure 1. Design space of dependable systems.

*This work has been partially supported by the NSF CAREER grant CCR-0238381 and also in part by the General Motors Collaborative Research Laboratory at Carnegie Mellon University.

Table 1. From high-level to low-level knobs

High-level Knobs	Scalability	Availability	Real-Time Guarantees
Low-level Knobs	Replication Style, #Replicas	Replication Style, Checkpointing Frequency	Replication Style, #Replicas, Checkpointing Frequency
Application Parameters	Frequency of Requests, Size of Requests and Responses, Resources	Size of State, Resources	Frequency of Requests, Size of Requests and Responses, Size of State, Resources

learn how to tune, appropriately, the trade-offs among fault-tolerance, performance and resource usage. The general versatile dependability framework consists of:

1. Monitoring various system metrics (e.g., latency, jitter, CPU load) in order to evaluate the conditions in the working environment [9];
2. Defining contracts for the specified behavior of the overall system;
3. Specifying policies to implement the desired behavior under different working conditions;
4. Developing algorithms for automatic adaptation to the changing conditions (e.g., resource exhaustion, introduction of new nodes) in the working environment.

Versatile dependability was developed to provide a set of control knobs to tune the multiple trade-offs. There are two types of knobs in our architecture: high-level knobs, which control the abstract properties from the requirements space (e.g., scalability, availability), and low-level knobs, which tune the fault-tolerant mechanisms that our system incorporates (e.g., replication style, number of replicas). The high-level knobs, which are the most useful ones for the system operators, are influenced by both the settings of the the low-level knobs that we can adjust directly (e.g., the replication style, the number of replicas, the checkpointing style and frequency), and the parameters of the application that are not under our control (e.g., the frequency of requests, the size of the application state, the sizes of the requests and replies). Through an empirical evaluation of the system, we determine in which ways the low-level knobs can be used to implement high-level knobs under the specified constraints (see Table 1), and we define adaptation policies that effectively map the high-level settings to the actual variables of our tunable mechanisms.

3. The System Architecture

Our versatile dependability framework is an enhancement to current middleware systems such as CORBA or Java (which lack support for tunable fault-tolerance). The tuning and adaptation to changing environments are done in a distributed manner, by a group of software components that work independently and that cooperate to agree and execute the preferred course of action.

These efforts are part of the MEAD (Middleware for Embedded Adaptive Dependability) project [9] which is currently under development at Carnegie Mellon University. While we currently focus on CORBA systems, which seemed the ideal starting point for this investigation given our previous experiences,[1] our approach is intrinsically independent of the middleware platform and can be applied to other systems as well.

3.1. A Tunable, Distributed Infrastructure

To ensure that our overall system architecture enables both the continuous monitoring and the simultaneous tuning of various fault-tolerance parameters, we have four distinct design goals for our software architecture:

- *Tunability and homogeneity:* having one infrastructure that supports multiple knobs and a range of different fault-tolerant techniques;
- *Quantifiability:* using precise metrics to evaluate the trade-offs among various properties of the system and to develop benchmarks for evaluating these metrics;
- *Transparency:* enabling support for replication-unaware and legacy applications;
- *Ease of use:* providing simple knobs that are intuitively easy to adjust.

We assume a distributed asynchronous system, subject to hardware and software crash faults, transient communication faults, performance and timing faults. The architecture of our system is illustrated in Figure 2. At the core of our approach is the *replicator*, a software module that can be used to provide fault-tolerance transparently to an application. The replicator module is implemented as a stack of sub-modules with three layers. The top layer is the interface to the CORBA application; it intercepts the system calls in order to understand the operations of the application. The middle layer contains all the mechanisms for transparently replicating processes and managing the groups of replicas. The bottom layer is the interface to the group communication package and is an abstraction layer to render the replicator portable to various communication platforms.

The unique feature of the replicator is that its behavior is tunable and that it can adapt dynamically to changing conditions in the environment. Given all the design choices for building dependable systems, the middle layer of the replicator can choose, from among different implementations, those that are best suited to meet the system's requirements.

Library Interposition This allows the replicator to perform tasks transparently to the application [7]. The replicator is a shared library that intercepts and redefines the standard system calls to convey the application's messages over a reliable group communication system. As the replicator

[1] MEAD was born out of the lessons learned in developing the fault-tolerant Eternal system [8]: real-time, resources and adaptation were not considered in Eternal's design.

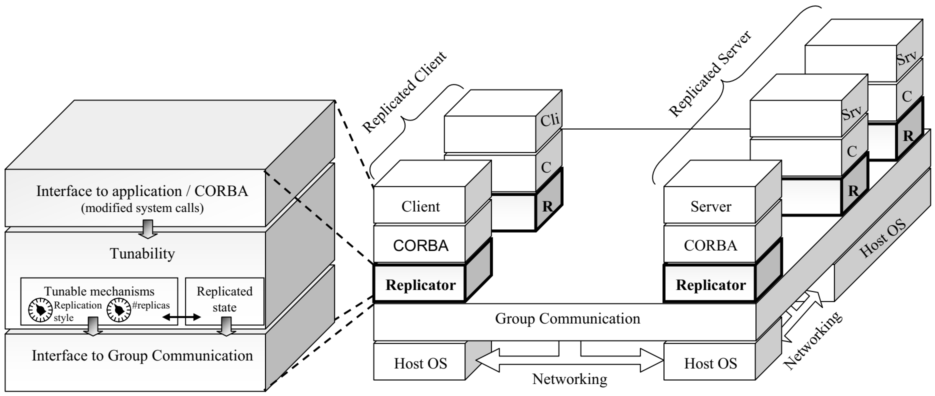

Figure 2. System Architecture.

mimics the TCP/IP semantics, the application continues to believe that it is using regular CORBA TCP/IP connections.

Group Membership and Communication We are currently using the Spread toolkit [1] for group communication. This package provides an API for joining/leaving groups, detecting failures and reliable multicasting.

Tunable Fault-Tolerant Mechanisms We provide fault-tolerant services to both CORBA client and server applications by replicating them in various ways, and by coordinating the client interactions with the server replicas. We implement replication at the process level rather than at the object level because a CORBA process may contain several objects (that share "in-process" state), all of which have to be recovered, as a unit, in the event of a process crash. Maintaining consistent replicas of the entire CORBA application is, therefore, the best way to protect our system against software (process-level) and hardware (node-level) crash faults.

We implement tunability by providing a set of low-level knobs that can adjust the behavior of the replicator, such as the replication style, the number of replicas and the checkpointing style and frequency (see Table 1). We currently support active and passive replication, with the intention of extending our infrastructure to handle other replication styles (e.g., semi-active). Note that versatile dependability does not impose a "one-style-fits-all" strategy; instead, it allows the maximum possible freedom in selecting a different replication style for each CORBA process and in changing it at run-time, should that be necessary.

Replicated State As the replicator is itself a distributed entity, it maintains (using the group communication layer) within itself an identically replicated object with information about the entire system (e.g., group membership, resource availability at all the hosts, performance metrics, environmental conditions). All of the decisions to re-tune the system parameters in order to adapt to changing working conditions are made in a distributed manner by a deterministic algorithm that takes this replicated state as its input. This has the advantage that the decisions are based on data that is already available and agreed upon, and, thus, the distributed adaptation process is very swift. This is accomplished through MEAD's decentralized resource monitoring infrastructure [9].

Adaptation Policies The replicator monitors various system metrics and generates warnings when the operating conditions are about to change. If the contracts for the desired behavior can no longer be honored, the replicator adapts the fault-tolerance to the new working conditions (including modes within the application, if they happen to exist). This adaptation is performed automatically, according to a set of policies that can be either pre-defined or introduced at run time; these policies correspond to the high-level knobs described in Section 2. For example, if the re-enforcement of a previous contract is not feasible, versatile dependability can offer alternative (possibly degraded) behavioral contracts that the application might still wish to have; manual intervention might be warranted in some extreme cases. As soon as all of the instances of the replicator have agreed to follow the new policy, they can start adapting their behavior accordingly.

3.2. Test Bed and Performance

We have deployed a prototype of our system on a test-bed of seven Intel x86 machines. Each machine is a Pentium III running at 900 megahertz with 512MB RAM of memory and running RedHat Linux 9.0. We employ the Spread (v3.17.01) group communication system [1] and the TAO real-time ORB [3] (v 1.4). In our experiments, we use a

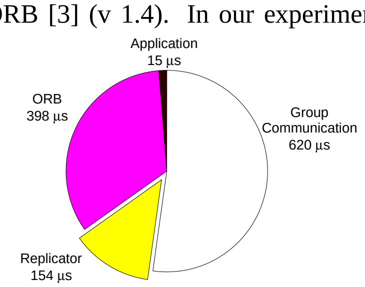

Figure 3. Break-down of the average latency.

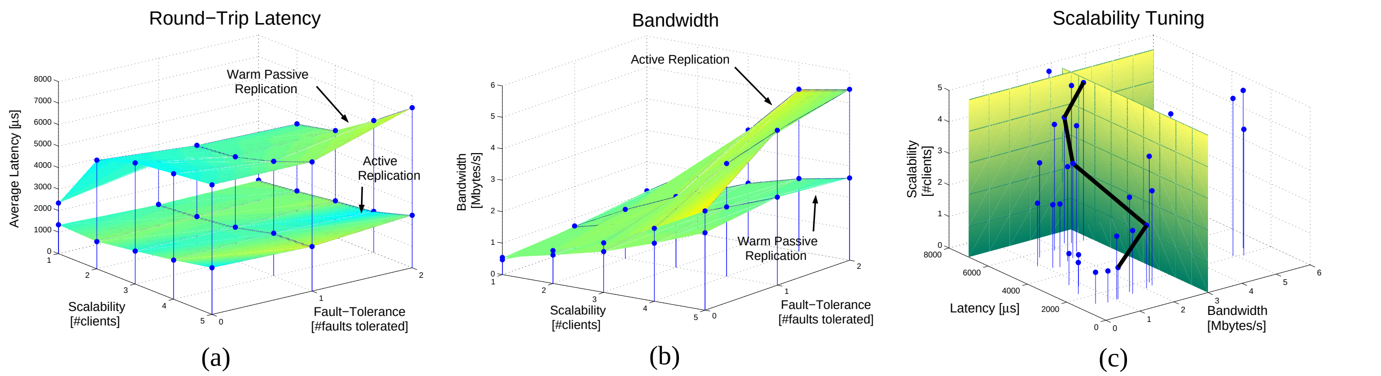

Figure 4. Tuning the scalability of the system under fault-tolerance, latency and bandwidth constraints.

CORBA client-server test application that processes a cycle of 10,000 requests.

Figure 3 shows a break-down of the average round-trip time of a request transmitted through MEAD, as measured at the client (in a configuration with one client and one server replica). We notice that the transmission delay through the group communication layer is the dominant contributor to the overall latency. The application processing time is very small because we are using a micro-benchmark; for a real application, the time to process the request would be significantly higher. The replicator introduces only 154 μs overhead on average, a fairly small figure compared to the latencies of the group communication system and the ORB.

4. Case Study: Tuning Scalability

The first step in tuning the scalability is to gather enough data about the system's behavior in order to construct a policy for implementing a high-level knob (see Section 3.1). We examine the average round-trip latency of requests, under different system loads and redundancy levels (because we were limited to eight computers, we ran experiments with up to five clients and three server replicas). In Figure 4-(a), we can see that the active replication incurs a much lower latency than warm passive replication, which makes the round-trip delays increase almost linearly with the number of clients. With five clients, passive replication is roughly three times slower than active replication.

The roles are reversed in terms of resource usage. In Figure 4-(b), we notice that, although in both styles the bandwidth consumption increases with the number of clients, the growth is steeper for active replication. Indeed, for five clients, active replication requires about twice the bandwidth of passive replication. Thus, when considering the scalability of the system, we must pay attention to the trade-off between latency and bandwidth usage. While this is not intuitively surprising, our quantitative data will let us determine the best settings for a given number of clients.

Implementing a "Scalability" Knob We would like to implement a knob that tunes the scalability of the system under bandwidth, latency, and fault-tolerance constraints. In other words, given a number of clients N_{cli}, we want to decide the best possible configuration for the servers (e.g., the replication style and the number of replicas). Let us consider a system with the following requirements:

Table 2. Policy for Scalability Tuning.

N_{cli}	1	2	3	4	5
Configuration[a]	A (3)	A (3)	P (3)	P (3)	P (2)
Latency [μs]	1245.8	1457.2	4966	6141.1	6006.2
Bandwidth [MB/s]	1.074	2.032	1.887	2.315	2.799
Faults Tolerated	2	2	2	2	1
Cost	0.268	0.443	0.669	0.825	0.895

[a] Active/Passive (number of replicas); e.g., A(3) = 3 active replicas.

1. The average latency shall not exceed 7000 μs;
2. The bandwidth usage shall not exceed 3 MB/s;
3. The configuration should have the best fault-tolerance possible (given requirements 1–2);
4. Among all the configurations i that satisfy the previous requirements, the one with the lowest:

$$Cost_i = p\frac{Latency_i}{7000\mu s} + (1-p)\frac{Bandwidth_i}{3MB/s}$$

should be chosen, where $Latency_i$ is the measured latency of i, $Bandwidth_i$ is the measured bandwidth and p is the weight assigned to each of these metrics.[2]

This situation is illustrated in Figure 4-(c). The hard limits imposed by requirements 1 and 2 are represented by the vertical planes that set the useful configurations apart from the other ones. For each number of clients N_{cli}, we select from this set those configurations that have the highest number of server replicas to satisfy the third requirement. If, at this point, we still have more than one candidate configuration, we compute the cost to choose the replication style (the number of replicas has been decided during the previous steps). The resulting policy is represented by the thick

[2] The cost function is a heuristic rule of thumb (not derived from a rigorous analysis), that we use to break the ties after satisfying the first 3 requirements; we anticipate that other developers could define different cost functions. Here, we use $p = 0.5$ to weight latency and bandwidth equally.

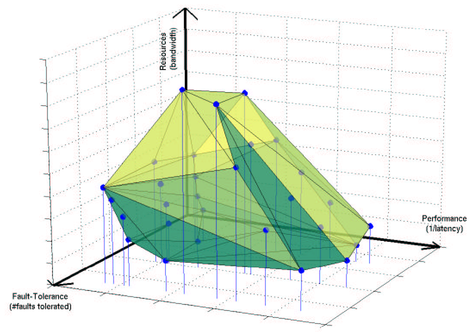

Figure 5. Versatile dependability of the system.

line from Figure 4-(c), and its characteristics are summarized in Table 2.

Note that, while for up to four clients, the system is able to tolerate two crash failures, for five clients only one failure is tolerated because no configuration with three replicas could meet the requirements in this case. This emphasizes the trade-off between fault-tolerance and scalability under the requirements 1–4, which impose hard limits for the performance and resource usage of the system. Furthermore, since in both the active and passive replication styles, at least one of the metrics considered (i.e., bandwidth and latency) increases linearly, it is likely that, for a higher load, we cannot satisfy the requirements. In this case, the system notifies the operators that the tuning policy can no longer be honored and that a new policy must be defined in order to accept any more clients.

Dependability Space Coverage Scalability is only one possible trade-off that versatile dependability can tune; we could similarly implement other high-level knobs such as availability, survivability, etc. In Figure 5 (which is the concrete instance of the abstraction of Figure 1, based on our experiments), we can see the fault-tolerance, performance and resource usage of each configuration of our system (normalized to their maximum values), as well as the region covered in the dependability space.[3] By moving inside this region, versatile dependability can quantify many trade-offs and it can implement various tuning policies.

5. Related Work

Among the first attempts to reconcile soft real-time and fault-tolerance, the Delta-4 XPA project [2] used semi-active replication (*the leader-follower model*) where all the replicas are active but only one designated copy (the leader) transmits output responses. In some conditions, this approach can combine the low synchronization requirements of passive replication with the low error-recovery delays of active replication. The ROAFTS project [4] implements a number of traditional fault-tolerant schemes in their rugged forms and operates them under the control of a centralized network supervision and reconfiguration (NSR) manager.

An offline approach to provisioning fault-tolerance was adopted by the MARS project [6] and its successor, the Time-Triggered Architecture (TTA) [5], which employ a static schedule (created at design time) with enough slack for the system to be able to recover when faults occur. This approach does not provide a generic solution because it delegates the responsibility for reconciling fault-tolerance and real-time requirements to the application designer.

6. Conclusions

Tunable software architectures are becoming important for distributed systems that must continue to run, despite loss/addition of resources, faults and other dynamic conditions. Versatile dependability is designed to facilitate the resource-aware tuning of multiple trade-offs between an application's fault-tolerance and QoS requirements. This architecture provides abstract high-level knobs for tuning system-level properties such as scalability and low-level knobs for selecting implementation choices. As a case study, we detail the implementation of such a scalability knob based on our empirical observations, and present the expanded trade-off space covered by our current implementation of versatile dependability.

[3]Figure 5 represents the same data set as Figure 4; these different visual representations emphasize the effect of dynamic changes on the effective fault-tolerance and QoS experienced by the system.

References

[1] Y. Amir, C. Danilov, and J. Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *International Conference on Dependable Systems and Networks*, 2000.

[2] P. A. Barrett, P. G. Bond, and A. M. Hilborne. The Delta-4 extra performance architecture (XPA). In *Fault-Tolerant Computing Symposium*, pages 481–488, Newcastle upon Tyne, U.K., June 1990.

[3] Douglas C. Schmidt et al. The design of the TAO real-time Object Request Broker. *Computer Communications*, 21(4), 1998.

[4] K. H. Kim. ROAFTS: A middleware architecture for real-time objectoriented adaptive fault tolerance support. In *Proceedings of IEEE High Assurance Systems Engineering (HASE) Symposium*, pages 50–57, 1998.

[5] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 2003.

[6] H. Kopetz and W. Merker. The architecture of MARS. In *Proceedings of FTCS*, page 50, 1985.

[7] J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers, San Francisco, CA, 2000.

[8] P. Narasimhan. *Transparent Fault-Tolerance for CORBA*. PhD thesis, University of California, Santa Barbara, 1999.

[9] P. Narasimhan, T. Dumitraş, S. Pertet, C. Reverte, J. Slember, and D. Srivastava. MEAD: Real-time, fault-tolerant CORBA. *Submitted to Concurrency and Computation: Practice and Experience*, 2003.