An Architecture for Versatile Dependability

Tudor Dumitras 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 × Performance × Resources} design space. From this data, we can extract the interdependencies among the three conflicting properties, and we can

*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.
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 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,[4] 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

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Resources</th>
<th>Size of Requests, Size of Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low-level Knobs</td>
<td>Replication</td>
<td>Replication</td>
</tr>
<tr>
<td>Frequency</td>
<td>Size of State, Frequency</td>
<td></td>
</tr>
<tr>
<td>Replication Style, #Replicas, Checkpointing Style, #Replicas, Checkpointing Frequency</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application Parameters</td>
<td>Frequency of Requests, Size of Resources, Size of Requests and Responses, Frequency</td>
<td></td>
</tr>
<tr>
<td>Scalability, Availability, Real-Time Guarantees</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

1MEAD 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.
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 [2].

**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 reinforcement 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
implement a knob that tunes the scalability of the system

We would like to mine the best settings for a given number of clients. Indeed, for five clients, passive replication requires about twice the bandwidth of active 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:

1. The average latency shall not exceed 7000 \( \mu s \);
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.

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 line.

**Table 2. Policy for Scalability Tuning.**

<table>
<thead>
<tr>
<th>( N_{cli} )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration</td>
<td>A(3)</td>
<td>A(3)</td>
<td>P(3)</td>
<td>P(3)</td>
<td>P(3)</td>
</tr>
<tr>
<td>Latency [( \mu s )]</td>
<td>1245.8</td>
<td>1457.2</td>
<td>4966</td>
<td>6141.1</td>
<td>6006.2</td>
</tr>
<tr>
<td>Bandwidth [MB/s]</td>
<td>1.074</td>
<td>2.032</td>
<td>1.887</td>
<td>2.315</td>
<td>2.799</td>
</tr>
<tr>
<td>Faults Tolerated</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Cost</td>
<td>0.268</td>
<td>0.443</td>
<td>0.669</td>
<td>0.825</td>
<td>0.895</td>
</tr>
</tbody>
</table>

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

References


