Architecting and Implementing Versatile Dependability

Tudor Dumitraş, Deepti Srivastava, and Priya Narasimhan*

Carnegie Mellon University, Pittsburgh PA 15213, USA
tdumitra@ece.cmu.edu, dsrivast@ece.cmu.edu, priya@cs.cmu.edu

Abstract. Distributed applications must often consider and select the appropriate trade-offs among three important aspects – fault-tolerance, performance and resources. We introduce a novel concept, called versatile dependability, that provides a framework for analyzing and reasoning about these trade-offs in dependable software architectures. We present the architecture of a middleware framework that implements versatile dependability by providing the appropriate "knobs" to tune and re-calibrate the trade-offs. Our framework can adjust the properties and the behavior of the system at development-time, at deployment-time, and throughout the application's life-cycle. This renders the versatile dependability approach useful both to applications that require static fault-tolerance configurations supporting the loss/addition of resources and changing workloads, as well as to applications that evolve in terms of their dependability requirements. Through a couple of specific examples, one on adapting the replication style at runtime and the other on tuning the system scalability under given constraints, we demonstrate concretely how versatile dependability can provide an extended coverage of the design space of dependable distributed systems.

1 Introduction

Oftentimes, the requirements of dependable systems 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. Conversely, distributed fault-tolerance techniques, such as replication, can adversely impact the performance and scalability. 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, many 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.

* This work has been partially supported by the NSF CAREER grant CCR-0238381, the DARPA PCES contract F33615-03-C-4110, and also in part by the General Motors Collaborative Research Laboratory at Carnegie Mellon University.

As an alternative, we propose *versatile dependability*[1], 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.

Our versatile dependability framework is an enhancement to current middleware systems such as CORBA or Java. While these middleware do have fault-tolerance support (through the Fault-Tolerant CORBA [2] and the Continuous Availability APIs for Java [3] standards), they lack the support for run-time adaptability. Furthermore, tuning these off-the-shelf middleware is an awkward task for their users because, in most cases, the adjustment process requires detailed knowledge of the system's implementation and because the internal tuning mechanisms are hard to control in an effective manner and can produce undesirable side-effects.

For example, the Fault-Tolerant CORBA standard [2] lists a set of "fault-tolerance properties" (*e.g.*, the replication style, the minimum number of replicas, the checkpointing intervals, the fault monitoring intervals and their timeouts), without providing any guidance as to how they ought to be set or how they map into externally-observable properties, such as scalability. We call these internal fault-tolerance properties the *low-level knobs*. The versatile dependability approach advocates the implementation of *high-level knobs*, corresponding to the external properties of the system, that encode the knowledge about the essential trade-offs and that provide the necessary insights on how to configure the system appropriately. Hence, the users of our COTS middleware do not need to quantify or understand the intricate relationships between internal and external properties, while enjoying the full benefits of an increased flexibility.

This paper makes four main contributions in describing:

- 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);
- How to implement the tuning knobs of versatile dependability, including two examples: dynamically adapting the replication style at runtime and adjusting the system scalability under specified constraints (Section 4);
- Why versatile dependability is relevant for several classes of applications, and what are the biggest challenges for extending this research direction (Section 5).

[1] An earlier version of this chapter, containing the first mention of versatile dependability, was published as [1].

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. In our research, we strive to achieve a high degree of flexibility by evaluating the wide variety of choices for implementing dependable systems, and by quantifying the effect of these choices on the three axes of our {Fault-Tolerance × Performance × Resources} design space. The purpose of this paper is to quantify some of the trade-offs among these three properties and to demonstrate how we can implement the most effective tuning knobs that allow system users and administrators, as well as application designers, to adjust these trade-offs appropriately.

Our 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 [4];
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.

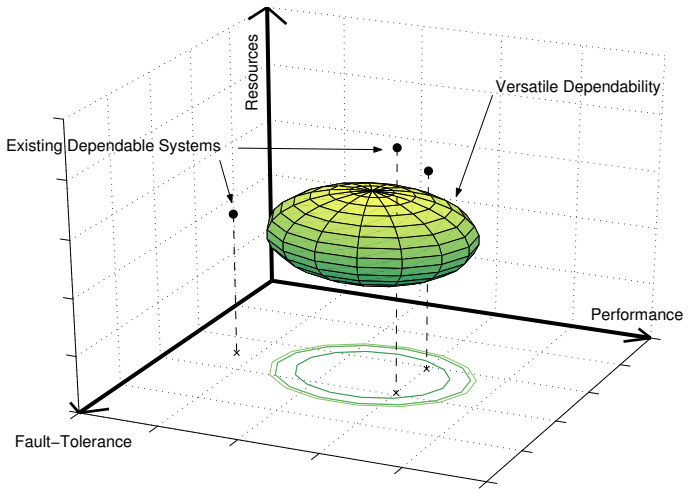

Fig. 1. Design space of dependable systems.

Table 1. Mapping from high-level to low-level knobs.

High-level Knobs	Scalability	Availability	Real-Time Guarantees
Low-level Knobs	Replication Style, Replication Degree	Replication Style, Checkpointing Frequency[a]	Replication Style, Replication Degree, Checkpointing Frequency
Application Parameters	Request Frequency, Request and Response Size, Resources	State Size, Resources	Request Frequency, Request and Response Size, State Size, Resources

[a] This knob is relevant only for passive replication (see Section 3.1)

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, and we define adaptation policies that effectively map the high-level settings to the actual variables of our tunable mechanisms. This approach complements a formal analysis of the system's correctness and performance and it shows how the system can be tuned and configured in its working environment. Table 1 shows three examples of mapping from high-level knobs to low-level knobs; in a complex system there can be many more such knobs and many other parameters that influence those knobs. In this paper, we consider a representative set of these knobs to illustrate the tuning process.

3 The Architecture of our Framework

Our framework is based on the Fault-Tolerant CORBA specification [2], which has only primitive support for tunable fault-tolerance. The tuning and adaptation to changing environments are enacted 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. In order to add a minimal overhead to the systems that we are continuously monitoring and tuning, we try to keep our system as simple as possible and to limit its functionality to the core

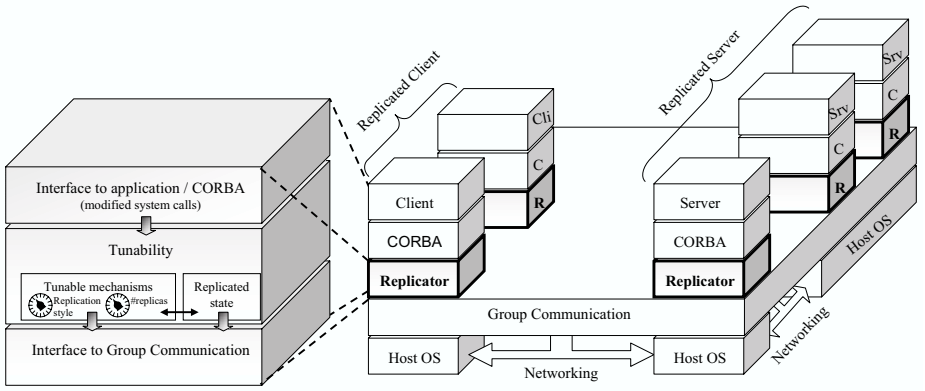

Fig. 2. System Architecture.

mechanisms needed to add and adjust fault-tolerance. We believe that this is important, especially since footprint and scalability are major concerns in some critical middleware applications.

This research forms a fundamental part of the MEAD (Middleware for Embedded Adaptive Dependability) project [4] 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,[2] our approach is intrinsically independent of the specific 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 system 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.

[2] MEAD was born out of the lessons that we learned in architecting and implementing the Eternal system [5]; however, Eternal was primarily designed to support fault-tolerant CORBA – real-time, resource-awareness and tunability were not considered in its design.

The taxonomy of low-level and high-level knobs helps us address the flexibility and ease of use requirements of versatile dependability: the knobs preserve the tunability of the system's behavior (by not hard coding the trade-off settings in the design choices) and they translate the internal variables of the framework into external properties that make sense for the system operators. The transparency and quantifiability requirements of versatile dependability are achieved through the architecture of our framework, which is discussed below.

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 a middleware application. The replicator intercepts the system calls of the CORBA application (on both the client and server sides), redirects the CORBA messages between hosts to a reliable group communication service, and manages groups of client and server replicas. Note that the application and the ORB need not be aware of all these tasks; in fact, we have successfully used the replicator to obtain fault-tolerant versions of legacy, un-replicated applications.

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, as well as the knobs needed to tune the system. 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. In the following paragraphs, we describe some of the techniques used by the replicator.

Library Interposition. This technique allows the replicator to perform tasks transparently to the application and to CORBA itself [6]. 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. Using linker-related environment variables (*e.g.*, `LD_PRELOAD`), we can insert the replicator ahead of all the other shared libraries in the CORBA application process' address space. At runtime, symbol definitions of interest to us (primarily socket and network level routines) resolve to the replicator rather than the default operating system libraries. This is accomplished with *no* change to the application, the ORB, or the operating system, thereby achieving transparency. The calls redefined inside the replicator are interposed between the application and the system libraries, such that, at runtime, the application (unknowingly) calls the

functions from the replicator, instead of the standard ones. Because the replicator mimics the TCP/IP programming interface, the application continues to believe that it is using regular CORBA GIOP connections. For example, if a client is trying to send a message to a server, we can intercept it and broadcast it (using group communication) to several replicas of that server in order to increase the dependability of the service.

Group Membership and Communication. We are currently using the Spread toolkit [7] for group membership and communication. This package provides an API (based on the extended virtual synchrony model [8]) for joining/leaving groups, detecting failures and reliable multicasting. Spread can provide five types of guarantees for message delivery: best effort (no guarantees), reliable delivery, FIFO ordering (by sender), causal ordering and total ordering. These guarantees enable us to ensure the consistency between the different replicas of the application. The price we have to pay for this consistency is that our system inherits the performance overhead of maintaining virtual synchrony between the nodes and the behavior of the replicator is closely related to the performance of the underlying group communication protocol.

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 loss of state or processing in the event of software (process-level) and hardware (node-level) crash faults.

Currently, the replicator supports the two canonical replication styles: active replication and passive replication:[3]

- *Active replication*, also called the "state-machine approach" [10], is a technique where all the replicas are running and processing requests simultaneously on different nodes. The client has two choices for determining the correct response:
  - it can accept the first response received, if the server replicas are trusted not to behave maliciously (which is the case in this paper);
  - it can do majority voting on all the responses it receives, if Byzantine failures may occur in the system [11].
- *Passive replication*, also called the "the primary-backup approach" [12], mandates that only one replica, called the *primary*, executes the application, while one or several *backups* are waiting to take over when the primary fails. Depending on how and when the state of the primary is transferred to the backups, this replication style has two flavors:

[3] In the future, we plan to include support for other replication styles [9] as well.

- *cold passive* replication, where a backup is launched (by a watchdog) only when the primary crashes, retrieving the state from a log saved on shared permanent storage, and
- *warm passive* replication, where the backups are in a stand-by mode, periodically receiving state updates from the primary. When the primary crashes, a new primary is chosen from among the running backups, using some deterministic algorithm.

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). 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.*, current view of the group membership, resource availability at all the hosts, performance metrics, environmental conditions). This object is needed for certain steps of the replication process (such as failover) and for making consistent decisions when adapting to the conditions in the environment. This is accomplished through MEAD's decentralized resource monitoring infrastructure and through the Fault-Tolerance Advisor [4], whose task is to identify the most appropriate configurations (including the replication style and degree) for the current state of the system.

Adaptation Policies. There are various reasons why a system may need to adapt its fault-tolerance properties. For example, an application may be multimodal and hence require different fault-tolerance in different modes, or runtime profiling of an application may show different resource availability at different times, and hence fault-tolerance policies would need to be adapted to this. These scenarios require different approaches and hence different adaptation algorithms.

Our system can perform static as well as runtime profiling to adapt the fault-tolerance of the system. It can monitor 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 adjusts the fault-tolerant mechanisms 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.

Application of Adaptation Policies. The decision to act on an adaptation policy must be applied consistently at all the nodes of the distributed system. This can be accomplished in two ways: (i) applying the adaptation without any further communication, based on the replicated state, and (ii) sending a "switch" message through a totally ordered multicast channel to initiate the change. With the first strategy, all the decisions to re-tune the system parameters are made in a distributed manner by a deterministic algorithm that takes the replicated state as input. If each local change is the outcome of events that are consistently delivered[4] at all the nodes by the resource monitoring system, then no further communication is needed; the decisions are based on data that is already available and agreed upon, and virtual synchrony ensures that the adaptation will be applied correctly. This has the advantage that the distributed adaptation process is very swift. With the second strategy, the system sends a "switch" message to all the replicators in a group; reception of this message triggers the adaptation process. This is equivalent to running Consensus to decide when to apply the change, and the "switch" message acts as a checkpoint in the totally ordered stream of messages indicating a time when all the replicas have received the same set of incoming messages and they are in the same state (we give a more detailed example of this strategy in Section 4.1). This approach introduces the delay of a totally ordered multicast between the time when an adaptation decision is made and the time when it is applied.

There are cases when the first strategy cannot be applied. For example, if the Fault-Tolerance Advisor runs as a separate process from the replicator, the decision to change will be communicated through an IPC or a shared memory mechanism. Since our system uses group communication to enforce consistency, using a side-channel (such as IPC or shared memory) may lead to unrecognized causality between the stream of requests and the adaptation decision and, therefore, the change could be applied when the replicas are in inconsistent states.[5] Integrating the replicator, the resource monitoring and the adaptation policy parsing in a single execution thread would remove this shortcoming, but it would increase the overhead of processing the requests. This shows that there is a trade-off between the overhead of the replicator in the average case and the ability to apply the adaptation policies very fast.

High and Low Level Knobs. Using all the mechanisms described above, we can implement the high and low level knobs mandated by versatile dependability. The group communication package allows us to implement a low-level knob that specifies the type of delivery guarantee the messages in the stream of requests have. Depending on the nature of the application, different types of messages may be used to achieve the target performance and dependability (for example, a stateless server requires only reliable message delivery, while a stateful server

[4] In the virtual synchrony model [13, 8], consistent delivery means that the same events are delivered in the same order, but without any timeliness guarantees.

[5] This does not happen when the requests do not update the state or when the replicas are stateless.

needs totally ordered messages if the requests contain state updates). Our replication mechanisms let us tune a number of parameters, such as the replication style, the number of replicas and the checkpointing frequency. The aggressiveness of resource monitoring and the strategy for applying adaptation policies define other low-level knobs that can be adjusted to control the overhead and the speed of the adaptation process. Finally, the high-level knobs are implemented on top of all these low-level knobs, using the adaptation policies.

4 Implementation of Tuning Knobs

Our versatile dependability framework includes both knobs that can be used off-line, to configure the system for particular requirements and workloads, and knobs that adapt to conditions in the working environment at runtime. Below, we estimate empirically the performance and the overhead of our framework (Section 4), we show how to implement a low-level knob that allows us to switch between an active and a passive replication style at runtime (Section 4.1), and we show how to construct a high-level knob to tune the system scalability (Section 4.2).

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 (v. 3.17.1) group communication system [7] and the TAO real-time ORB [14] (v. 1.4). In our experiments, we use a CORBA client-server test application that processes a cycle of 10,000 requests.

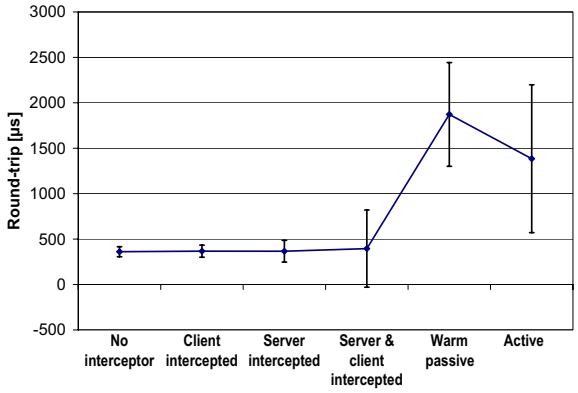

Fig. 3. Overhead of the replicator for a remote client–server application.

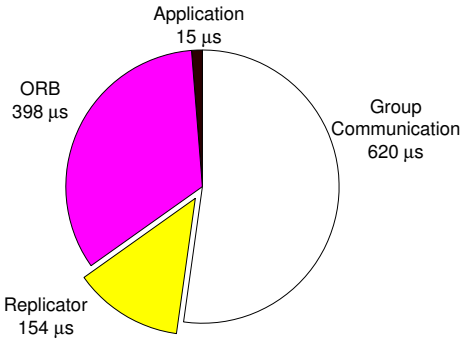

Fig. 4. Break-down of the average round-trip time.

Performance and Overhead of the Replicator. In Figure 3, we examine the raw overhead introduced by the replicator and the replication mechanism. We compare here the latencies of the baseline application (without the replicator), of an operating mode where the system calls are intercepted, but not modified (with just the client, just the server, and both of them intercepted), and of the active and warm passive replication styles (with one client and an unreplicated server to keep the results comparable). The vertical error bars from the figure indicate the jitter measured in the corresponding experiment. We can see that the replicator itself introduces little overhead, but the replication mechanisms lead to increased latency and jitter.

Figure 4 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 an unreplicated server). We notice that the transmission delay through the group communication layer is the dominant contributor to the overall latency (in this paper, by latency we mean round-trip time). 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.

4.1 Runtime Adaptive Replication

The active and passive replication styles represent different trade-offs between timeliness, recovery and resource usage. In general, active replication is faster in responding to requests and in recovering from faults because checkpointing and rollback are not needed, while passive replication uses more efficiently the resources available, such as bandwidth and CPU cycles. Adaptive systems should be able to modify replication styles on the fly, at run-time, in response to workload changes and application requirements. We implement a low-level knob to switch between replication styles through three steps (see also the pseudocode in Figure 5):

1. One or more replicas initiate the transition process by sending a "switch" message to the entire replica group (duplicate messages are discarded);
2. Each replica, on receiving the "switch" message, starts enqueuing application messages and broadcasts all the information needed by the other replicas to update their local state and to perform the switch;
3. Each replica, on receiving all the information needed to ensure a consistent state, updates its internal state and assumes its role in the new replication style.

The second step is different depending on the direction of the switch: when switching from warm passive to active replication, the backups must synchronize their states with the primary before they can start processing requests. In the case of a crash of the primary, the backups can restore a consistent state by replaying the messages received since the last checkpoint prior to the crash. When switching from active to warm passive replication, a new primary must be selected and the other replicas become backups after finishing to service their current requests.

The "switch" messages are sent through a totally ordered, reliable multicast channel using our group communication layer (see Section 3.1), which makes our algorithm tolerant to the crash of any replica. Since fault notifications are ordered consistently with respect to the "switch" and the other messages, the remaining non-faulty instances of the replicator can always determine at which point in the algorithm the crash has occurred and continue the work from that point until the replication style switch is complete. The protocol described in Figure 5 can tolerate the crash failure of either the primary or of any of the backups.

Our adaptive replication style takes the middle ground between the fast, resource-hungry active replication and the slower, resource-efficient passive replication. Figure 6 shows how we can adapt the replication style in response to the load of the system. Since active replication can handle higher request arrival rates than passive replication, in this example we switch whenever the request rate increases above a certain threshold. This simple adaptation policy selects the replication style that is appropriate for the measured request arrival rate at the server.

The observed delays required to complete the switch are comparable to the average response time, and they are negligible at high loads, such as the ones that trigger the adaptation. It is interesting to note that the request arrival rate observed at the server is 4.1% higher in the case of adaptive replication than when using static passive replication with the same workload. This is because active replication can respond faster under such high loads; clients waiting for the replies receive them faster and can send new requests sooner than in the previous case (there is no need for quiescence and checkpointing). This speed-up effect allows the servers to regulate the load imposed by the clients and to increase the throughput of the replicated service.

The adaptive replication knob provides the ability to change the replication style whenever required, either off-line, before the application is launched, or

```
I INITIATE adaptation:
     send switch message

II PREPARE to switch:
   /* Case 1: switch Warm Passive --> Active */
     If (this replica == current Primary)
             prepare to send one more checkpoint before switching
     If (replica == current Backup)
             prepare to wait for one more checkpoint after the switch
   /* Case 2: switch Active --> Warm Passive */
     Choose a new primary
     Prepare to handle outstanding messages, if any, after the switch

III SWITCH to new replication style:
   /* Case 1: switch Warm Passive --> Active */
     New replication style = Active
     If (this replica == previous Primary)
             send one more checkpoint
     If (this replica == previous Backup)
             accept one more checkpoint
             If (no checkpoints received &&
                 detect crash of previous Primary)
                     process all outstanding requests
                     in message queue (rollback)
             else
                     continue
   /* Case 2: switch Active --> Warm Passive */
     New replication style = Active
     If (this replica == new backup)
             If (any outstanding requests in message queue)
                      process those requests and then
                      become completely passive
             else
                      continue as backup
```

Fig. 5. Algorithm to switch between replication styles.

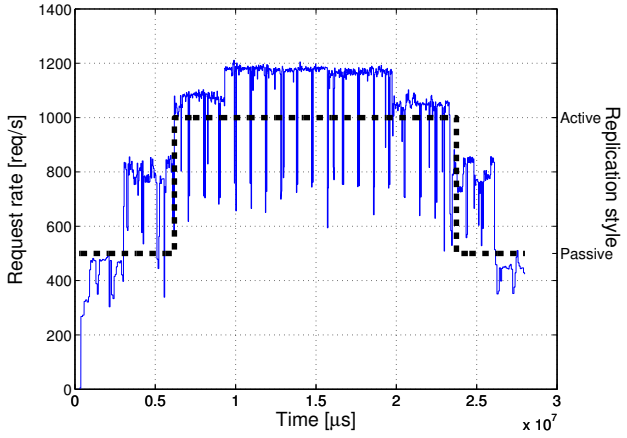

Fig. 6. Low-level knob: adaptive replication.

online, during its execution. This flexibility allows us to tune with precision the behavior of dependable systems in the space between active and passive replication, by defining the appropriate adaptation policies. This is essential when the middleware infrastructure needs to support a graceful degradation to operation modes with reduced functionality, (*e.g.*, when taking the system in a safe mode when the loss of redundancy threatens the reliability and safety of the system). However, adaptive replication is most useful for implementing high-level knobs that correspond to external system properties, as described in the next section.

4.2 Tuning System Scalability

In this section, we show we can tune the *scalability* of the system (*i.e.*, the number of clients it can service) under specified resource and performance constraints. The first step in implementing a scalability knob 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 7-(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 7-(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

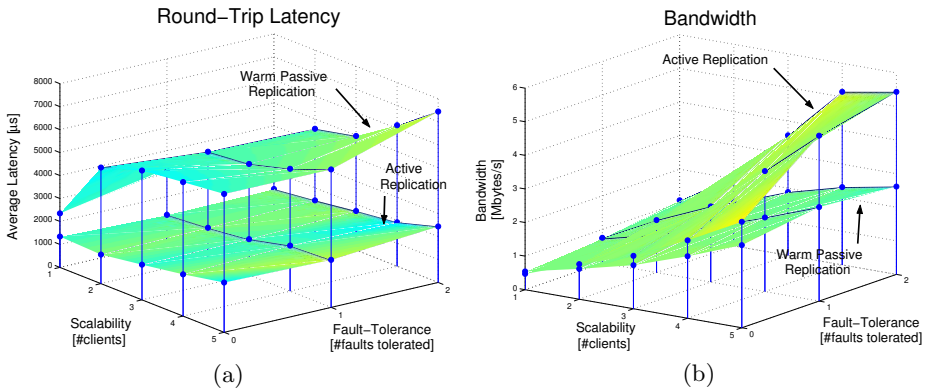

Fig. 7. Trade-off between latency and bandwidth usage.

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 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.[6]

This situation is illustrated in Figure 8. The hard limits imposed by requirements 1 and 2 are represented by the vertical planes that set the useful

[6] 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.

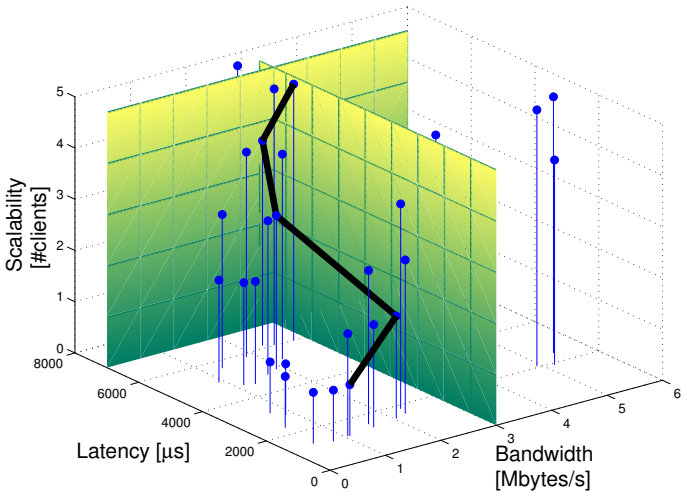

Fig. 8. High level knob: scalability.

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 from Figure 8, 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

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.

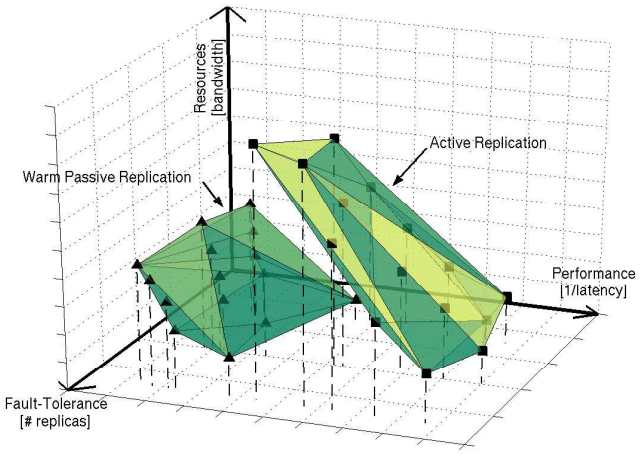

Fig. 9. Active and passive replication in the dependability design space.

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.

5 Discussion

Scalability is only one possible high-level knob that versatile dependability can tune; we could similarly implement other high-level knobs such as availability, reliability, sustained throughput, etc. In fact, each one of the requirements specified in Section 4.2 probably corresponds to a high-level knob that can be tuned independently. Achieving a separation of concerns between these knobs, by reducing the influence they have on each other, is therefore an important research challenge for the future of versatile dependability.

However, in its current stage versatile dependability rises to the challenge of enabling adaptive systems with a tunable range of reliability and performance guarantees. Figure 9 displays the trade-off between the active and passive replication styles in the dependability design space (which was introduced in Figure 1). The data set displayed here is the same one from Figure 7, where the fault-tolerance, performance and resource usage of each configuration are normalized to their maximum values. We can see that each of the two replication styles corresponds to a larger region in this space and includes multiple possible configurations of the system. The two regions are non-overlapping; however, by using low and high-level knobs such as the ones described above, we can position the system in any configuration desired.

Versatile dependability is essential for long-running applications that cannot be stopped (*e.g.*, during a space flight), but that have several modes of operation

with different resource and performance requirements (*e.g.*, simulation/ training and mission modes). The high performance provided by active replication can be used when gathering data and performing actuation must be done within narrow time limits, when there are limited windows of opportunity and data is critical, because of the faster response and recovery times. The more conservative passive replication is needed when the resources are scarce and cannot be wasted by running several active replicas in parallel. When both these conditions are present (*e.g.*, in a network of sensors), the infrastructure must be able to tune the replication style to run in a resource-conservative mode most of the time, and to switch to the high-performance mode only during the limited window of opportunity. The ability to express the tuning problem in terms of external properties (the high-level knobs), rather than internal parameters of the system (the low-level knobs), facilitates the configuration and management of complex distributed systems because it does not require a detailed knowledge of the system's implementation and the internal fault-tolerant mechanisms used.

6 Related Work

Among the first attempts to reconcile soft real-time and fault-tolerance, the Delta-4 XPA project [15] 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 [16] 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.

Traditionally, research on adaptive software systems has focused on either system architectures to support the adaptation process [17, 18], or on domain-specific strategies for adaptation under given constraints encountered in practical situations [19, 20]. The former approach does not make use of any domain knowledge about the application and, thus, only enables hooks for adaptation while leaving the actual implementation details to domain experts; the latter approach usually focuses on one particular (often domain-specific) instance of the problem and does not build a generic framework around the proposed solution.

For instance, the AQuA framework [19] proposes a technique to support graceful QoS adaptation by requiring applications to specify the criticality of their timeliness and consistency requirements in probabilistic terms. This probabilistic QoS model can be implemented through replication and a combination of virtual synchrony and lazy propagation of updates that effectively provides a tunable range of consistency guarantees. Based on the client's request and the measured conditions in the environment (*e.g.*, current network latencies and replica staleness), the framework detects whether the client's QoS specification can be met with the required probability. In this case, AQuA automatically selects the subset of replicas to service the invocation using a greedy algorithm.

Note that, in our terminology, AQuA's tunable QoS guarantees are analogous to a high-level knob.

However, in some cases, the QoS requirements and the environmental conditions can change so drastically that a switch to a completely different algorithm is necessary. Cactus [17] proposes a generic software architecture for adaptive systems based on fine-grained software modules that implement abstract QoS properties. The adaptation framework uses fitness functions associated with each module to determine the best one for the current requirements and execution environment. The adaptive action is performed after all the distributed components have agreed to select the corresponding software modules in a consistent way. This adaptation mechanism is similar to a low-level knob from our framework, such as the one described in Section 4.1.

It has also been noted that hybrid replication strategies can be conceived, and these can be combined with caching in order to give more flexibility to the application designer [21]. For example, some of the replicas can be active and some can be passive in order to increase the scalability of the system while keeping low fail-over delays. There are possibly 50–100 such hybrid strategies which give a much finer control of the operational parameters of the system. An analysis of all these combinations, emphasizing the most useful ones of them, would result in a better coverage of the presently very sparsely populated space of replication strategies.

For example, the DARX framework [22] is aimed at providing adaptive fault-tolerance for multi-agent software platforms. This infrastructure associates a replication policy with each agent, and the replication style and degree are adjusted according to the importance of each agent with respect to the rest of the application. This derives from a fundamental assumption that the importance of an agent evolves over time and so do its fault-tolerance requirements.

An offline approach to selecting the appropriate trade-off between fault-tolerance and real-time guarantees was adopted by the MARS project [23] and its successor, the Time-Triggered Architecture (TTA) [24] which are based on time-triggered protocols with strong temporal predictability. Fault-tolerance is achieved in the TTA by using a static schedule (created at design time) that allows 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 who establishes the static schedule.

7 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. We propose the concept of "knobs" as a convenient architectural feature that helps designers reason about the system trade-offs and that

expresses the tuning process in terms of externally-observable properties of the system. The architecture described in this paper provides abstract high-level knobs for tuning system-level properties such as scalability and low-level knobs for selecting implementation choices, such as the replication style. We detail the implementation of such knobs based on empirical observations, and present the expanded trade-off space covered by our current implementation of versatile dependability.

References

1. Dumitraş, T., Narasimhan, P.: An architecture for versatile dependability. In: Workshop on Architecting Dependable Systems, International Conference on Dependable Systems and Networks, Florence, Italy (2004)
2. Object Management Group: Fault Tolerant CORBA. OMG Technical Committee Document formal/2001-09-29 (2001)
3. Java Community Process: J2EE APIs for Continuous Availability. Java Specification Request, JSR117 (2003)
4. Narasimhan, P., Dumitraş, T., Paulos, A., Pertet, S., Reverte, C., Slember, J., Srivastava, D.: MEAD: Support for real-time, fault-tolerant CORBA. Concurrency and Computation: Practice and Experience **17** (2005)
5. Narasimhan, P.: Transparent Fault-Tolerance for CORBA. PhD thesis, University of California, Santa Barbara (1999)
6. Levine, J.R.: Linkers and Loaders. Morgan Kaufmann Publishers, San Francisco, CA (2000)
7. Amir, Y., Danilov, C., Stanton, J.: A low latency, loss tolerant architecture and protocol for wide area group communication. In: International Conference on Dependable Systems and Networks, New York, NY (2000) 327–336
8. Moser, L.E., Amir, Y., Melliar-Smith, P.M., Agarwal, D.A.: Extended virtual synchrony. In: The 14th IEEE International Conference on Distributed Computing Systems (ICDCS). (1994) 56–65
9. M. Wiesmann et al.: Understanding replication in databases and distributed systems. In: International Conference on Distributed Computing Systems, Taipei, Taiwan, R.O.C. (2000) 264–274
10. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Computing Surveys **22** (1990) 299–319
11. Lamport, L., Shostak, R., Pease, M.: The Byzantine Generals problem. ACM Transactions on Programming Languages and Systems **4** (1982) 382–401
12. Budhiraja, N., Schneider, F., Toueg, S., Marzullo, K.: The primary-backup approach. In Mullender, S., ed.: Distributed Systems. ACM Press - Addison Wesley (1993) 199–216
13. Birman, K., Joseph, T.: Exploiting virtual synchrony in distributed systems. In: Symposium on Operating Systems Principles (SOSP). (1987) 123–138
14. Douglas C. Schmidt et al.: The design of the TAO real-time Object Request Broker. Computer Communications **21** (1998)
15. Barrett, P.A., Bond, P.G., Hilborne, A.M.: The Delta-4 extra performance architecture (XPA). In: Fault-Tolerant Computing Symposium, Newcastle upon Tyne, U.K. (1990) 481–488

16. Kim, K.H.: ROAFTS: A middleware architecture for real-time objectoriented adaptive fault tolerance support. In: Proceedings of IEEE High Assurance Systems Engineering (HASE) Symposium, Washington, DC (1998) 50–57
17. Chen, W.K., Hiltunen, M., Schlichting, R.: Constructing adaptive software in distributed systems. In: International Conference on Distributed Computing Systems, Phoenix, AZ (2001) 635–643
18. Nett, E., Gergeleit, M., Mock, M.: Guaranteeing real-time behaviour in adaptive distributed systems. In: Symposium on Large Scale Systems: Theory and Applications, University of Patras, Greece (1998)
19. Cukier, M., Ren, J., Sabnis, C., Sanders, W.H., Bakken, D.E., Berman, M.E., Karr, D.A., Schantz, R.: AQuA: An adaptive architecture that provides dependable distributed objects. In: Proceedings of the IEEE 17th Symposium on Reliable Distributed Systems, West Lafayette, IN (1998) 245–253
20. González, O., Shrikumar, H., Stankovic, J., Ramamritham, K.: Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling. In: IEEE Real-Time Systems Symposium, San Francisco, CA (1997) 79–89
21. Bakken, D., Bjune, G., Ahmad, M.: Towards hybrid replication and caching strategies. In: Digest of FastAbstracts presented at the International Conference on Dependable Systems and Networks, New York (2000)
22. Marin, O., Bertier, M., Sens, P.: Darx - a framework for the fault-tolerant support of agent software. In: International Symposium on Software Reliability Engineering, Denver, USA (2003)
23. Kopetz, H., Merker, W.: The architecture of MARS. In: Fault-Tolerant Computing Symposium (FTCS-15), Ann Arbor, MI (1985) 247–259
24. Kopetz, H., Bauer, G.: The time-triggered architecture. Proceedings of the IEEE **91** (2003) 112–126