Static Analysis Meets Distributed Fault-Tolerance: Enabling State-Machine Replication *with* Nondeterminism*

Joseph G. Slember and Priya Narasimhan
Electrical & Computer Engineering Department
Carnegie Mellon University, Pittsburgh, PA 15213

Abstract

Midas is an inter-disciplinary approach to supporting state-machine replication for nondeterministic distributed applications. The approach exploits compile-time static analysis to identify both first-hand and second-hand sources of nondeterminism. Subsequent runtime compensation occurs through either the transfer of nondeterministic checkpoints or the re-execution of inserted code, and restores consistency among replicas before each new client request. The approach avoids the need for lock-step synchronization and leverages application-level insight to address only the nondeterminism that matters. Our preliminary evaluation demonstrates Midas' feasibility and current performance overheads.

1. Introduction

State-machine replication [11] has long been used for providing fault-tolerance for servers; here, every server replica receives, performs, and responds to, every client request. One advantage of state-machine replication is its failure-masking, i.e., if a replica fails while processing a client's request, another actively functioning replica is either processing or will process the same request, hiding the failure from the client. Thus, state-machine replication is often used for rapid recovery.

State-machine replication requires deterministic application behavior. This means that any two server replicas, when starting from the same initial state and executing the same ordered sequence of operations, should reach the same final state and produce the same output.

Nondeterminism can arise from various sources, such as local timers, multithreading, system calls, etc. When nondeterminism is present in servers that use state-machine replication for fault-tolerance, the server replicas can diverge in state. The resulting inconsistency defeats the purpose of replication as a fault-tolerance strategy. However, many real-world applications inevitably contain nondeterminism. Thus, state-machine replication is precluded for these applications, even if their availability/recovery requirements are demanding.

Many research efforts [2, 3, 4, 5, 6, 7, 8, 9, 12] have focused on addressing nondeterminism for state-machine and other types of replication. A common strategy is to use transparent, lock-step synchronization. The idea is that every time a server replica executes a nondeterministic call, the replica is effectively paused until the result of the call is synchronized across all of the running replicas. This ensures consistent state across all of the server replicas throughout their execution. A transparent approach results because the layer that intercepts every nondeterministic call to perform the synchronization is often at the virtual-machine, middleware or OS level, and the application is unaware of the existence of this layer.

We have previously developed such an approach [10], and learned some lessons in the process. Transparency, while attractive to application programmers, is not really ideal for handling nondeterminism because (i) not every nondeterministic call actually materializes into replica divergence (we provide examples later), and (ii) a transparent layer cannot identify second-hand nondeterminism that arises when the results of (first-hand) nondeterministic calls "taint" otherwise deterministic code. In addition, lock-step synchronization has a performance cost, by requiring all of the replicas to virtually cease operation until they reach consensus on a result.

With these insights, we developed a new approach called Midas. There are two primary questions that drive our design and implementation of Midas. Can we leverage application-level knowledge in a manner that facilitates the accurate handling of (first- and second-hand) nondeterminism? Can we provide effectively consistent state-machine replication by asynchronously handling all of the detected sources of nondeterminism, so that replicas are free to operate independently and use nondeterministic features, without needing to forcibly synchronize with each other?

2. Midas in a Nutshell

Midas is a deliberately non-transparent approach to handling nondeterminism in distributed, replicated applica-

* Partially supported by the NSF CAREER grant CCR-0238381 and the ARO grant number DAAD19-02-1-0389

tions. By exploiting techniques from the field of static analysis, we are able to extract and leverage application-level information about the origin, the propagation and the impact of nondeterminism. The static analysis, together with an automated instrumentation framework, allows us to insert "compensation" code for the detected, relevant sources of nondeterminism. This compensation (described later) effectively supports state-machine replication by allowing replicas to asynchronously render themselves consistent with each other, at runtime, prior to handling every new client request.

Midas imposes some requirements in its approach. First, compiler-based techniques such as static analysis require access to source code. However, they have the advantage of being able to extract information from source-code that would be lost to a transparent approach. Secondly, we currently require totally-ordered reliable multicast to communicate every message to each group of server replicas. Thus, all of the replicas of a server will receive the same set of messages in the same order, even if they produce different results on processing these messages. We do not limit the kinds of nondeterminism that the application can exhibit, and do not require special middleware, virtual-machine or OS support.

We aim to relax the requirement of determinism for distributed applications that wish to use state-machine replication, and show how an inter-disciplinary approach (applying static analysis to distributed fault-tolerance) can achieve that goal.

3. Design and Implementation

3.1. Kinds of Nondeterminism

Control-flow nondeterminism is due to any path of execution within the application that is not deterministic, and primarily arises from multithreading, exceptions and signals. *Interaction nondeterminism* is due to the application's interaction with the system/environment via system/library calls, e.g., those dealing with file-system I/O and memory interaction.

We also distinguish between actual and superficial nondeterminism. For example, if a `gettimeofday` call is executed, and its results are stored in a state variable, that variable constitutes actual nondeterminism. If the results of the call are not stored but discarded (say, after printing them to screen), then, that instance of `gettimeofday` does not need to be consistent across server replicas; this is referred to as superficial nondeterminism. A transparent approach would necessarily address every occurrence, superficial or actual, of a nondeterministic call. On the other hand, static analysis of the application can focus our attention on the interesting (i.e., the actual) nondeterminism.

First-hand, or pure, nondeterminism refers to any execution or state that is the direct/root source of nondeterminism, e.g., an instance of `gettimeofday`. Because multithreading can introduce nondeterminism, shared state among threads is also considered as first-hand nondeterminism. Multithreading can arise when a server requires multiple threads to process a single request. However, even if a server requires only one thread to process a single client's request, the simultaneous processing of multiple clients can lead to multithreading.

Second-hand, or contaminated, nondeterminism refers to any execution or state that is "touched" by pure nondeterminism or other second-hand nondeterminism. Contamination occurs if a nondeterministic value (say, variable `a`) propagates due to dependencies (e.g., `b = f(a)`), thereby rendering other variables or state (in this case, `b`) nondeterministic, even if the latter were deterministic left to themselves. In multi-tier applications (where one server plays the role of a client for another server tier), requests and replies between server tiers can serve as "carriers" of nondeterminism.

The tracking of contamination requires application-level information about first-hand nondeterministic sources and how they influence other application state through dependencies, requests and replies. Static analysis can help to pinpoint first-hand nondeterminism as well as track second-hand nondeterminism.

3.2. Static-Analysis Framework

Midas' program-analysis framework comprises the front-end of a compiler coupled with a source-code regenerator. The custom framework, written from scratch, converts C/C++ source-code into an annotated abstract syntax tree (AST), performs several analytical passes over the tree, automatically generates and inserts code snippets and, finally, outputs source-code.

Our framework identifies first-hand sources of control-flow and interaction nondeterminism within the application. For interaction nondeterminism, Midas analyzes the application's source-code, seeking out instances of items in our "nondeterminism-dictionary", currently consisting of (i) 262 system calls, including `read`, `write`, `gettimeofday`, etc., and (ii) 163 library functions within the C/C++ standard I/O, memory and machine-dependent OS libraries.

Apart from seeking out these system calls and library routines, Midas performs a comprehensive search for any additional sources of first-hand nondeterminism that might not exist in our dictionary. To this end, our framework extracts all function calls from the application, and processes this list in four steps. (1) All of the application-defined functions (i.e., neither system calls nor library routines) are carefully removed from this list. Some application-level defined functions might be added

later for consideration if control-flow analysis reveals their potential for nondeterminism. (2) All of the matches between this list and our dictionary are discarded since they are known to be nondeterministic. (3) All of the functions in this list that are dependent on (i.e., contaminated second-hand by) functions in our dictionary are added to our dictionary and also removed from the list; an example is `fread`, which invokes the nondeterministic `read` call, resulting in second-hand nondeterminism. (4) What is left of the list at this point are functions whose potential for nondeterminism we must ascertain manually, by examining the source-code for those functions. If these functions are found to be nondeterministic by inspection, we add them to our dictionary.

To identify control-flow nondeterminism, Midas extracts all shared state between threads automatically[1]. All of the reads and writes made by each thread to this shared state are also flagged as first-hand nondeterminism. Instead of forcing the deterministic acquisition of mutexes and the identical execution of threads across server replicas, we assume that all interleaving/executions of threads are valid; we then compensate for any resulting replica divergence after the fact.

Midas then performs control-flow and data-flow analyses on the application source-code (this includes the joint analysis of the client and the server code, to cover the distributed contamination of nondeterminism). The analyses produce a list of inter-dependencies of state variables within the application, depending on the control path that is chosen. This control-path-specific dependency list allows us to identify all second-hand nondeterminism within the application; for example, if all writes to an inter-thread state variable `x` constitute first-hand nondeterminism, then, the analyses determine what other state within the application is contaminated by `x`. At the end of this phase, Midas has identified every piece of both first- and second-hand nondeterminism within each server of the distributed application.

3.3. Compensation Techniques

Midas performs two kinds of annotations to the application source-code to track, and compensate for, the nondeterminism at runtime. (1) The first set of annotations consists of data structures for holding nondeterministic information at runtime. For instance, Midas inserts thread-specific arrays to track each thread's order of execution and the thread's modification to any inter-thread, shared state. (2) The second set of annotations consists of code-snippets that can recreate the second-hand nondeterminism at a server, if provided the first-hand nondeterministic variables as input. For instance, if a variable `x` is written to by a thread and three other state variables are subsequently contaminated by `x`, Midas can create and insert a new function that takes `x` as input, and re-executes code to recreate the values of the three contaminated nondeterministic variables. We explain how (1) and (2) compensate for nondeterminism, without requiring lock-step synchronization across replicas.

Midas employs two kinds of compensation techniques: checkpointing and re-execution. In both cases, the replicas do not need to block or wait on each other before executing requests. All of the replicas of a server are rendered consistent, in an asynchronous manner, before processing each client request.

We use a multi-tier example, client $C \rightleftarrows$ server $S_1 \rightleftarrows$ server $S_2 \rightleftarrows$ ..., to illustrate our compensation techniques. A $\rightarrow$ denotes a downstream request while a $\leftarrow$ denotes an upstream reply. We focus on the caller S_1 and the callee S_2 in the following discussion on inter-tier compensation.

Checkpoint-to-compensate (`transfer-contam`): In this case, the information from the annotations in (1) is used. Within each S_2 replica, both the first- and second-hand nondeterminism are locally tracked and stored (we call this a nondeterministic checkpoint). Each S_2 replica returns its response to S_1, piggybacking its nondeterministic checkpoint. Due to state-machine replication, every S_2 replica transmits its response to S_1. On its end, S_1 chooses the first-received response from the S_2 replicas and processes it, discarding the remaining responses. Because of the underlying totally ordered protocol, if S_1 is replicated, each S_1 replica will choose the same S_2 replica. S_1 notes its choice of the S_2 replica (the chosen S_2 replica can differ from one request to the next) and initiates an asynchronous callback to each of the S_2 replicas, passing along its choice of the S_2 replica and that replica's nondeterministic checkpoint. The callback has no effect at the chosen S_2 replica; the other S_2 replicas overwrite their own nondeterministic checkpoint with the one received in the callback.

Reexcute-to-compensate (`reexec-contam`): Here, we use the information from the annotations in (1) and (2). Each S_2 replica piggybacks only its first-hand nondeterminism in its response to S_1. As with `transfer-contam`, S_1 chooses an S_2 replica, and piggybacks that replica's first-hand nondeterminism in its callback to the S_2 replicas. The chosen S_2 replica is unaffected; the other S_2 replicas overwrite their own first-hand nondeterminism with the one received in the callback, and then recreate the second-hand nondeterminism by re-executing the code in (2).

Additional Intricacies. After processing the callback, all of the server/callee replicas are consistent

1 If dynamic pointers are used to access state, we need to enhance Midas with off-the-shelf packages to assist in the analysis.

once more in their first- and second-hand nondeterminism, and ready to process the next request. In a multi-tier application, inter-tier compensation can occur concurrently with the end-to-end operation. In the example, S_1 could issue and complete a compensation callback to S_2, while C is still processing the response returned from S_1.

Midas employs the notion of forward and backward callbacks to handle nondeterminism contamination that occurs due to requests (i.e., $C \rightarrow S_1 \rightarrow S_2 \rightarrow ....$) and replies (i.e., $C \leftarrow S_1 \leftarrow S_2 \leftarrow ....$), respectively. For instance, in the midst of processing a request from C, an S_1 server replica might issue a downstream request to S_2, wait for S_2's response to arrive, process the incoming response, and then respond, in its turn, to C. Forward nondeterminism at S_1 is any nondeterminism due to the processing that occurs at S_1 after it receives C's request, but before it issues its own request to S_2. Similarly, backward nondeterminism at S_1 ensues during any processing after it receives S_2's response, but before it responds to C. Both forward and backward nondeterminism need to be addressed. Using Midas, the downstream tier (S_2) issues a forward callback to S_1 to compensate for the forward nondeterminism, while the upstream tier (C) issues a backward callback to compensate for the backward nondeterminism.

There are several intricate details that we omit here for lack of space. One such detail is the level of concurrency and dependency analysis that Midas performs on the application source-code to determine the causal relationships between forward requests and backward responses at each tier, as well as the ensuing forward and backward nondeterministic state.

We emphasize that, while the annotations in (1) and (2) do modify the original application source-code, they do not alter the functionality or the semantics expected of the servers, and they do not introduce any nondeterminism of their own. The annotations consist mostly of data structures for tracking and holding nondeterministic information at runtime. Even the re-execution code that Midas adds is deterministic, although it takes a nondeterministic input. Thus, we allow every server replica to continue to be nondeterministic, just as the application programmer had intended, but exploit Midas to restore consistency asynchronously before each new client request is processed in the system.

The use of the `transfer-contam` vs. `reexec-contam` techniques depends on the relative costs of transferring a nondeterministic checkpoint (consisting of both first- and second-hand nondeterminism) vs. re-executing to regenerate the second-hand nondeterminism. The main difference in the two techniques is the types of overhead incurred. The overhead of `reexec-contam`, arises predominantly from computational costs, while that of `transfer-contam` arises from increased communication. Application programmers can choose which technique to use based on their needs and application characteristics [1]. If communication overhead is not a significant issue, the `transfer-contam` is preferable, while `reexec-contam` might be a better option if computational time is readily available.

4. Preliminary Evaluation

We evaluated Midas' implementation and compensation techniques using variations of a basic multi-tier, multi-client, micro-benchmark application on Emulab. Each server tier performs the same amount of processing, and each client has identical functionality. Every server replica and every client is located on a different Pentium III, 850MHz machine with 256MB RAM running TimeSys Linux 2.4 over a 100 Mbps LAN. The application is multithreaded with shared state across threads and uses nondeterministic system calls (e.g. `random()`); we can also vary the amount of forward and backward nondeterminism. Our goal with this microbenchmark is to show the performance and feasibility of our approach with respect to the number of clients and the number of replicated tiers. We varied our experimental configurations to change (i) the number of clients to 2 and 4, (ii) the number of tiers to 2 and 4, (iii) the forward nondeterminism to 5% and 60% of the total state within the tier, (iv) the backward nondeterminism to 5% and 60% of the total state within the tier, and (v) the compensation techniques, `reexec-contam` or `transfer-contam`.

We implement a `transfer-ckpt` technique, similar to `transfer-contam`, that transfers the entire state (deterministic and nondeterministic) of the server replica. In the baseline `vanilla` case, replicas are allowed to remain nondeterministic and no compensation is involved. The metric that we use for evaluation is round-trip time as measured at a client. For each configuration, we compute the average round-trip time across all the clients for 300 end-to-end invocations/client.

The "total state" of a tier is represented by two arrays (one forward and one backward) of 10,000 `long`s each. x% forward nondeterminism and y% backward nondeterminism mean that x% of the forward array is nondeterministic on the forward request-path and y% of the backward array is nondeterministic on the backward reply-path. The backward state depends on the forward state if the latter is accessed by the incoming/backward reply.

We have two major sources of nondeterminism: multithreading with shared state and nondeterministic system calls. By changing the amount of overlapping state across threads, we vary the amount of first-hand nondeterminism. Changing the amount of state modified by a first-hand nondeterministic system call can vary the amount

of second-hand nondeterminism. These two sources of nondeterminism are split equally for the purposes of introducing varying amounts of nondeterminism in our micro-benchmark's experimental configurations. Therefore, if 60% of the total state is nondeterministic, 30% of state is first-hand nondeterministic and shared across the threads, and 30% of state is due to second-hand nondeterminism.

Results. Figures 1(b) and 1(a) show Midas' performance for a fixed 4 replicas/tier and a varying number of clients. The workload across the tiers doubles when the number of clients doubles, as seen by the linear increase in round-trip time for the 4-tier case in Figure 1(b).

Our evaluation is performed for low (5%) and significant (60%) amounts of both forward and backward nondeterminism within the application. Clearly, the lower the amount of nondeterminism within the application, the less the compensation work to be done and the lower the overheads, as seen in the 5% nondeterminism case in Figure 1(a). Even in this case, the `transfer-ckpt` case stands out, particularly for the 4-tier situation, because of the significant amount of deterministic state that constitutes each checkpoint; the nondeterministic portion of the checkpoint does not contribute significantly to the overheads, as seen in `transfer-contam` in Figure 1(a). Note also that, regardless of the number of tiers and the % of nondeterminism, more clients imply higher latencies because more work results from all of the accompanying interleaving and callbacks.

Figure 1(b) shows that Midas can handle even highly nondeterministic applications, albeit with expectedly larger overheads for all of the compensation techniques because of the additional work in coping with the increased nondeterminism. `transfer-ckpt` still tops the chart because of the amount of state that constitutes an entire checkpoint. For an application with lower cost for state-transfer and higher cost for re-execution, we might expect `reexec-contam` to exhibit higher overheads.

5. Midas for a Transparent Approach

We view Midas' operation as two separate phases. The first phase is the initial program analysis that Midas performs on the application source-code to determine nondeterministic attributes. Midas then implements compensation techniques to address the detected nondeterminism. While it is clear that the synergy of the program analysis and the subsequent compensation effectively accomplish our goals, Midas' need to access and modify application source-code might be viewed as disadvantageous, if not outright impossible in some cases.

Even if a transparent nondeterminism-compensation approach is to be used, thereby avoiding source-code modifications, Midas' program-analysis phase can be useful. The main purpose of Midas' program analysis is to differentiate between the actual nondeterminism that causes replica divergence and the superficial nondeterminism that does not matter. Many transparent approaches will inevitably over-compensate by addressing even any superficial nondeterminism in the application. However, the results of Midas' analysis of the application source-code could be fed into a transparent approach. In other words, even a transparent approach to handling nondeterminism might be improved with application-level knowledge. This improvement can be seen in the form of increased efficiency in several ways.

For instance, a significant amount of nondeterminism is often due to multithreading. Although the threads might share data, Midas' analysis can determine whether or not the threads interfere with each other in a way that the replicas diverge in state. Armed with this information, a transparent approach can simply ignore thread interactions that will not cause replica divergence. Without this application-level information, a transparent approach would likely attempt to intercept and compensate for *all* thread interactions, regardless of their impact on replica consistency. Thus, even a transparent approach stands to benefit from Midas' program analysis ahead of deployment, although the approach might employ other compensation mechanisms at runtime.

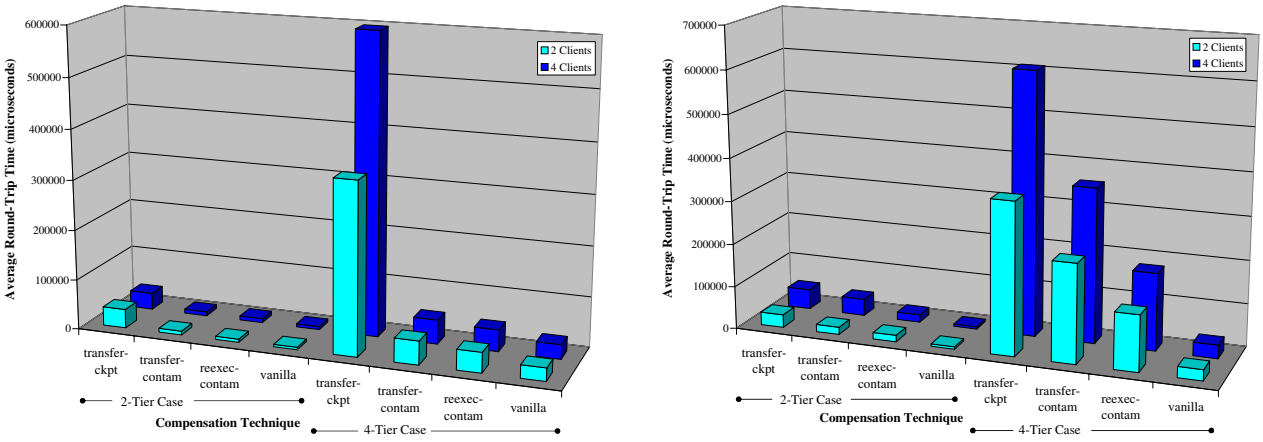
6. Looking Forward

Our work in this paper was not an exercise in optimization, but a demonstration of the feasibility of supporting state-machine replication with nondeterminism in multi-tier, multi-client distributed applications, where nondeterminism might propagate in a rampant way. There are many performance enhancements that we can make to Midas to reduce its runtime compensation overheads. We could vary other application-level characteristics – request size, state size, processing time, inter-request latency – to complete our evaluation. Also, much of our technique is based on static analysis. We intend to incorporate dynamic-analysis techniques to gain any application-level information that is beyond the scope of static analysis.

6.1. Other Replication Styles

Passive, or primary-backup, replication is often hailed as the solution to alleviate the difficulties posed by state-machine replication for nondeterministic applications. With passive replication, a designated primary/leader replica processes all of the requests and replies, and synchronizes itself with its backup replicas by sending them periodic checkpoints of its state.

Without application-level insight, even passive replication cannot handle nondeterminism in multi-tier, multi-client, distributed applications. Given the possi-



(a) **Low (5%) forward & backward nondeterminism**

(b) **Significant (60%) forward & backward nondeterminism**

Figure 1. Overhead for compensation techniques for varying number of clients.

ble propagation of nondeterminism through inter-tier dependencies, checkpoints cannot necessarily be taken independently at the primary replica of any tier without sufficient consideration of the remaining tiers in the system. Also, without the due analysis of inter-tier communication and cross-invocation dependencies, only one end-to-end operation can be performed at any time in the system; all of the tiers would need to be blocked until an ongoing end-to-end operation completes.

We intend to investigate how Midas can benefit even passive replication schemes through its program analysis. The objective would be to exploit application-level information to support passive replication in multi-tier, multi-client nondeterministic applications without compromising replica consistency or sacrificing concurrency.

7. Conclusion

Midas supports the state-machine replication of nondeterministic distributed applications. The approach involves a synergistic combination of compile-time dependency, concurrency and nondeterminism analyses, along with different performance-sensitive techniques to compensate for the nondeterminism at runtime. The compensation involves asynchronous callbacks that let replicas continue to execute nondeterministic calls, but that reconcile them prior to processing every client request, without requiring lock-step synchronization. Our preliminary evaluation demonstrates Midas' feasibility and current performance overheads.

References

[1] J. G. Slember and P. Narasimhan. Living with nondeterminism in replicated middleware systems, *ACM/IFIP Conference on Middleware*, Melbourne, Australia, Nov 2006.

[2] L. Alvisi and J. Napper. A transparent fault tolerant Java Virtual Machine, *IEEE Conference on Dependable Systems and Networks*, San Francisco, CA, June 2003, pp. 425–434.

[3] R. Friedman and A. Kama. Transparent fault-tolerant Java Virtual Machine, *IEEE Symposium on Reliable Distributed Systems*, Florence, Italy, Oct 2003, pp. 319–328.

[4] L. Alvisi, E. Elnozahy, Y. M. Wang and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems, *ACM Computing Surveys*, vol. 34, no. 3, Sept 2002, pp. 375–408.

[5] M. Patino-Martinez, R. Jimenez-Peris, B. Kemme and G. Alonso. MIDDLE-R: Consistent database replication at the middleware level, *ACM Transactions on Computer Systems.*, vol. 23, no. 4, Nov 2005, pp. 375–423.

[6] C. Basile, Z. Kalbarczyk and R. Iyer. A preemptive deterministic scheduling algorithm for multithreaded replicas, *IEEE Conference on Dependable Systems and Networks*, San Francisco, CA, June 2003, pp. 149–158.

[7] S. Bestaoui. One solution for the nondeterminism problem in the SCEPTRE 2 fault tolerance technique, *Euromicro Workshop on Real-Time Systems*, Odense, Denmark, June 1995, pp. 352–358.

[8] T. C. Bressoud, TFT: A software system for application-transparent fault tolerance, *International Symposium on Fault-Tolerant Computing*, Munich, Germany, June 1998, pp. 128–137.

[9] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault-tolerance, *ACM Transactions on Computer Systems*, vol. 14, no. 1, Feb 1996, pp. 80–107.

[10] P. Narasimhan, L. E. Moser and P. M. Melliar-Smith. Enforcing determinism for the consistent replication of multithreaded CORBA applications, *IEEE Symposium on Reliable Distributed Systems*, Lausanne, Switzerland, Oct 1999, pp. 263–273.

[11] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial, *ACM Computing Surveys* vol. 22, no. 4, Dec 1990, pp. 299–319.

[12] T. Wolf. *Replication of Non-Deterministic Objects*, PhD thesis, EPFL, Switzerland, Nov 1988.