MoDist: Transparent Model Checking of Unmodified Distributed Systems

Junfeng Yang[°*], Tisheng Chen[‡], Ming Wu[‡], Zhilei Xu[‡], Xuezheng Liu[‡]
Haoxiang Lin[‡], Mao Yang[‡], Fan Long[†], Lintao Zhang[‡*], Lidong Zhou[‡*]
[°] Columbia University, [‡] Microsoft Research Asia
[*] Microsoft Research Silicon Valley, [†] Tsinghua University

Abstract

MoDist is the first model checker designed for *transparently* checking *unmodified* distributed systems running on *unmodified* operating systems. It achieves this transparency via a novel architecture: a thin interposition layer exposes all actions in a distributed system and a centralized, OS-independent model checking engine explores these actions systematically. We made MoDist practical through three techniques: an execution engine to simulate consistent, deterministic executions and failures; a *virtual clock* mechanism to avoid false positives and false negatives; and a state exploration framework to incorporate heuristics for efficient error detection.

We implemented MoDist on Windows and applied it to three well-tested distributed systems: Berkeley DB, a widely used open source database; MPS, a deployed Paxos implementation; and PacificA, a primary-backup replication protocol implementation. MoDist found 35 bugs in total. Most importantly, it found *protocol-level* bugs (i.e., flaws in the core distributed protocols) in every system checked: 10 in total, including 2 in Berkeley DB, 2 in MPS, and 6 in PacificA.

1 Introduction

Despite their growing popularity and importance, distributed systems remain difficult to get right. These systems have to cope with a practically infinite number of network conditions and failures, resulting in complex protocols and even more complex implementations. This complexity often leads to corner-case errors that are difficult to test, and, once detected in the field, impossible to reproduce.

Model checking has been shown effective at detecting subtle bugs in real distributed system implementations [19, 27]. These tools systematically enumerate the possible execution paths of a distributed system by starting from an initial state and repeatedly performing all possible actions to this state and its successors. This *state-space exploration* makes rare actions such as network failures appear as often as common ones, thereby quickly driving the target system (i.e., the system we check) into corner cases where subtle bugs surface.

To make model checking effective, it is crucial to expose the actions a distributed system can perform and do so at an appropriate level. Previous model checkers for distributed systems tended to place this burden on users, who have to either write (or rewrite) their systems in a restricted language that explicitly annotates event handlers [19], or heavily modify their system to shoehorn it into a model checker [27].

This paper presents MoDist, a system that checks *unmodified* distributed systems running on *unmodified* operating systems. It simulates a variety of network conditions and failures such as message reordering, network partitions, and machine crashes. The effort required to start checking a distributed system is simply to provide a simple configuration file specifying how to start the distributed system. MoDist spawns this system in the native environment the system runs within, infers what actions the system can do by *transparently* interposing between the application and the operating system (OS), and systematically explores these actions with a centralized, OS-independent model checking engine. We have carefully engineered MoDist to ensure the executions MoDist explores and the failures it injects are consistent and deterministic: inconsistency creates false positives that are painful to diagnose; non-determinism makes it hard to reproduce detected errors.

Real distributed systems tend to rely on timeouts for failure detection (e.g., leases [14]); many of these timeouts hide in branch statements (e.g., "`if(now > t + timeout)`"). To find bugs in the rarely tested timeout handling code, MoDist provides a *virtual clock* mechanism to explore timeouts systematically using a novel static symbolic analysis technique. Compared to the state-of-the-art symbolic analysis techniques [3, 4, 13, 31], our method reduces analysis complexity using the following two insights: (1) programmers use time values in *simple* ways (e.g., arithmetic operations) and (2) programmers check timeouts *soon* after they query the current time (e.g., by calling `gettimeofday()`).

We implemented MoDist on Windows. We applied it to three well-tested distributed systems: Berkeley DB, a widely used open-source database; MPS, a Paxos implementation that has managed production data centers with more than 100K machines for over two years; and PacificA, a primary-backup replication protocol implementation. MoDist found 35 bugs in total. In particular, it found *protocol-level* bugs (i.e., flaws in the core protocols) in every system checked: 10 in total, including 2 in Berkeley DB, 2 in MPS, and 6 in PacificA. We measured the speed of MoDist and found that (1) MoDist incurs reasonable overhead (up to 56.5%) as a checking tool and (2) it can *speed up* a checked execution (up to 159 times faster) using its virtual clock.

MoDist provides a customizable framework for incorporating various state-space exploration strategies. Using this framework, we implemented dynamic partial order reduction (DPOR) [9], random exploration, depth-first exploration, and their variations. Among these, DPOR is a strategy well-known in the model checking community for avoiding redundancy in exploration. To evaluate these strategies, we measured their protocol-level coverage (i.e., unique protocol states explored). The results show that, while DPOR achieves good coverage for a small bounded state space, it scales poorly as the state space grows; a more balanced variation of DPOR, with a set of randomly selected paths as starting points, achieves the best coverage.

This paper is organized as follows. We present an overview of MoDist (§2), then describe its implementation (§3) and evaluation (§4). Next we discuss related work (§5) and conclude (§6).

2 Overview

A typical distributed system that MoDist checks has multiple processes,* each running multiple threads. These processes communicate with each other by sending and receiving messages through socket connections. MoDist can re-order messages and inject failures to simulate an *asynchronous and unreliable* network. The processes may write data to disk, and MoDist will generate different possible crash scenarios by permuting these disk writes.

The remainder of this section gives an overview of MoDist, covering its architecture (§2.1), its checking process(§ 2.2), the checks it enables (§ 2.3), and its user interface (§2.4).

2.1 Architecture

Figure 1 illustrates the architecture of MoDist applied to a 4-node distributed system. The master node runs multiple threads (the curved lines in the figure) and might send or receive messages (the solid boxes). For each process in the target system, MoDist inserts an interposition frontend between the process and its native operating system to intercept and control non-deterministic decisions involving thread and network operations.

MoDist further employs a backend that runs in a different address space and communicates with the frontends via RPC. This design minimizes MoDist's perturbation of the target system, allowing us to build a generic backend that runs on a POSIX-compliant operating system, and makes it possible to build the frontends for MoDist on different operating systems. The backend consists of five components: a dependency tracker, a failure simulator, a virtual clock manager, a model checking engine, and a global assertion checker.

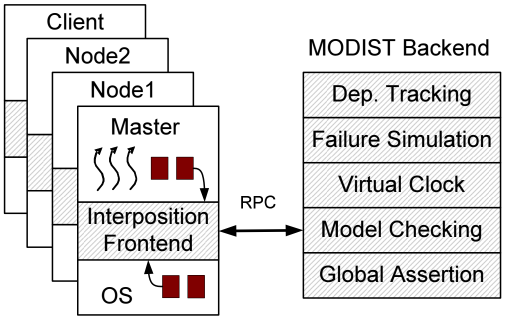

Figure 1: MoDist *architecture.* All MoDist components are shaded. The target system consists of one master, two replication nodes, and one client. MoDist's frontend interposes between each process in the target system and the operating system to intercept and control non-deterministic actions, such as message interleaving and thread interleaving. MoDist's backend runs in a separate address space to schedule these actions.

Interposition. MoDist's interposition frontend is a thin layer that exposes what actions a distributed system can do and lets MoDist's backend deterministically schedule them. Specifically, it does so in two steps: (1) when the target system is about to execute an action, the frontend pauses it and reports it to the backend; and (2) upon the backend's command, the frontend either resumes or fails the paused action, turning the target system into a "puppet" of the backend.

We place the interposition layer at the OS-application boundary to avoid modifying either the target system or the underlying operating system. In addition, despite variations in OS-application interfaces, they provide similar functions, allowing us to build a generic backend.

Since the interposition layer runs inside the target system, we explicitly design it to be simple and mostly stateless, and leave the logic and the state in the backend, thereby reducing the perturbation of the target system.

Dependency Tracking. MoDist's dependency tracker oversees how actions interfere with each other. It uses these dependencies to compute the set of *enabled* actions, i.e., the actions, if executed, that will not block in the OS. For example, a `recv()` is enabled if there is a message to receive, and disabled otherwise. The model checking engine (described below) only schedules enabled actions, because scheduling a disabled action will deadlock the target system (analogous to a cooperative thread scheduler scheduling a blocked thread).

Failure Simulation. MoDist's failure simulator or-

*In this paper we use node and process interchangeably.

```
# command          working dir  inject failure?
  master.exe        ./master/         1
  node.exe          ./node1/          1
  node.exe          ./node2/          1
  client.exe test1  ./client/         0
```

Figure 2: A configuration file that spawns the distributed system in Figure 1. We used this file to check PacificA.

chestrates many rare events that may occur in a distributed system, including message reordering, message loss, network partition, and machine crashes; these events can expose bugs in the often-untested failure handling code. The failure simulator lets MoDist inject these failures as needed, consistently to avoid false positives, and deterministically to let users reliably reproduce errors (cf. §2.2).

Virtual Clock. MoDist's virtual clock manager has two main functions: (1) to discover timers in the target system and fire them as requested by MoDist's model checking engine to trigger more bugs, and (2) to ensure that all processes in the target system observe a consistent clock to avoid false positives. Since the clock is virtual, MoDist can "fast forward" the clock as needed, often making a checked execution faster than a real one.

Model Checking. MoDist's model checking engine acts as an "omnipresent" scheduler of the target system. It systematically explores a distributed system's executions by enumerating the actions, failures, and timers exposed by the other MoDist components. It uses a set of search heuristics and state-space reduction techniques to improve the efficiency of its exploration. We elaborate the model checking process in next section and the search strategies in §3.6.

Global Assertion. MoDist's global assertion mechanism lets users check distributed properties on consistent global snapshots; these properties cannot be checked by observing only the local states at each individual node. Its implementation leverages our previous work [25].

2.2 Checking Process

With all MoDist's components in place, we now describe MoDist's checking process. To begin checking a distributed system, the user only needs to prepare a simple configuration file that specifies how to start the target system. Figure 2 shows a configuration file for the 4-node replication system shown in Figure 1; it is a real configuration that we used to check PacificA. Each line in the configuration tells MoDist how to start a process in the target system. A typical configuration consists of 2 to 10 processes. The "inject failure" flag is useful when users do not want to check failures for a process. For example, `client.exe` is an internal test program that does not handle any failures, so we turned off failure checking for this process.

```
init_state = checkpoint(create_init_state());
q.enqueue(init_state, init_state.actions);

while(!q.empty()) {
  <state, action> = q.dequeue();
  try {
    next_state = checkpoint(action(restore(state)));
    global_assert(next_state); //check user-provided global assertions
    if (next_state has never been seen before)
      q.enqueue(next_state, next_state.actions);
  } catch (Error e) {
    // save trace and report error
    . . .
  }
}
```

Figure 3: Model checking pseudo-code.

With a configuration file, users can readily start checking their systems by running `modist <config>`. MoDist then instruments the executables referred to in the configuration file to interpose between the application and the operating system, and starts its model checking loop to explore the possible *states* and *actions* in the target system: a *state* is an instantaneous snapshot of the target system, while an *action* can be to resume a paused WinAPI function via the interposition layer, to inject a failure via the failure simulator, or to fire a timer via the virtual clock manager.

Figure 3 shows the pseudo-code of MoDist's model checking loop. MoDist first spawns the processes specified in the configuration to create an *initial state*, and adds all $\langle \mathit{initial\ state}, \mathit{action} \rangle$ pairs to a *state queue*, where *action* is an action that the target system can do in the initial state. Next, MoDist takes a $\langle \mathit{state}, \mathit{action} \rangle$ pair off the state queue, restores the system to *state*, and performs *action*. If the action generates an error, MoDist will save a trace and report the error. Otherwise, MoDist invokes the user-provided global assertions on the resultant global state. MoDist further adds new state/action pairs to the state queue based on one of MoDist's search strategies (cf. §3.6 for details.) Then, it takes off another $\langle \mathit{state}, \mathit{action} \rangle$ pair and repeats.

To implement the above process, MoDist needs to checkpoint and restore states. It uses a *stateless* approach [12]: it checkpoints a state by remembering the actions that created the state and restores it by redoing all the actions. Compared to a stateful approach that checkpoints a state by saving all the relevant memory bits, a stateless approach requires little modifications to the target system, as previous work has shown [12, 19, 28, 39].

2.3 Checks

The checks that MODIST performs include generic checks that require no user intervention as well as user-written system-specific checks.

Currently, MODIST detects two classes of generic errors. The first is "fail-stop" errors, which manifest themselves when the target system unexpectedly crashes in the absence of an injected crash from MODIST. These crashes can be segmentation faults due to memory errors or program aborts because MODIST has brought the target system into an erroneous state. MODIST detects these unexpected crashes by catching the corresponding signals. The second is "divergence" errors [12], which manifest themselves when the target system deadlocks or goes into an infinite loop. MODIST catches these errors using timeouts. When MODIST schedules one of the actions of the target system, it waits for a user-specified timeout interval (10 seconds by default) until the target system gets back to it; otherwise, MODIST will flag a divergence error.

Because MODIST checks the target system by executing it, MODIST can easily check the effects of real executions and find errors. Thus, we can always combine MODIST with other dynamic error detection tools (e.g., Purify [16] and Valgrind [29]) to check more generic properties; we leave these checks for future work.

In addition to generic checks, MODIST can perform system-specific checks via user-provided assertions, including local assertions (via the `assert()` statements) inserted into the target system and global assertions that run in the centralized model checking engine. Given these assertions, MODIST will amplify them by driving the target code into many possible states where these assertions may fail. In general, the more assertions users add, the more effective MODIST will be.

2.4 Advanced User Interface

As with most other automatic error detection tools, the more system-specific knowledge MODIST has, the more effective it will be. For users who want to check their system more thoroughly, MODIST provides the following methods for incorporating domain knowledge.

Users can add more program assertions in the code for a more thorough check. In addition to these local assertions, users can enrich the set of checks by specifying *global assertions* in MODIST. These assertions check distributed properties on any consistent global snapshot.

Users can make MODIST more effective by reducing their system's state space. A simple trick is to bound the number of failures MODIST injects per execution. Our previous work [38, 39] showed that tricky bugs are often caused by a small number of failures at critical moments. Obviously, without bounds on the number of failures, a distributed system may keep failing without making any progress. In addition, developers tend to find bugs triggered by convoluted failures uninteresting [38].

Users can provide hints to let MODIST focus on the states (among an infinite number of states) that users consider most interesting. Users can do so in two ways: (1) extend one of MODIST's search algorithms through the well-defined state queue interface, and (2) construct a test case to test some unusual parts of the state space.

3 Implementation

We implemented MODIST on Windows by intercepting calls to WinAPI [36], the Windows Application Programming Interface. We chose WinAPI because it is the predominant programming interface used by almost all Windows applications and libraries, including the default POSIX implementation on Windows. While we built MODIST on Windows, we expect that porting to other operating systems, such as Linux, BSD, and Solaris, should be easy because WinAPI is more complicated than the POSIX API provided by most other operating systems. For example, WinAPI has several times as many functions as POSIX. Moreover, many WinAPI functions operate in both synchronous and asynchronous mode, and the completion notifications of asynchronous IO (AIO) may be delivered through several mechanisms, such as events, *select*, or IO completion ports [36].

When we implemented MODIST we tried to adhere to the following two goals:

1. *Consistent and deterministic execution.* The executions MODIST explores and the failures it injects should be consistent and deterministic to avoid difficult-to-diagnose false positives and non-deterministic errors.
2. *Tailor for distributed systems.* We explicitly designed MODIST to check distributed systems. Having this goal in mind, we customized our implementation for distributed systems and avoided being overly general.

These goals were reflected at many places in our implementation. In the rest of this section, we describe MODIST's implementation in details, highlighting the decisions entailed by these goals.

3.1 Interposition

MODIST's interposition layer transparently intercepts the WinAPI functions in the target system and allows MODIST's backend to control it deterministically. There are two main issues regarding interposition. First, *interposition complexity*: since the interposition layer runs inside the address space of the target system, it should be as simple as possible to avoid perturbing the target system, or introducing inconsistent or non-deterministic executions. Second, *IO abstraction*: as previously mentioned, WinAPI is a wide interface with rich semantics; Windows networking IO is particularly complex. To avoid

Category	# of functions	# of LOC
Network	28	1816
Time	7	161
File System	9	640
Mem	5	126
Thread	33	1433
Shared		1290
Total	82	5466

Table 1: *Interposition complexity.* This table shows the lines of code for WinAPI wrappers, broken down by categories. The "Shared" row refers to the code shared among all API categories. Most wrappers are fairly small (67 lines on average).

excessive complexity in MODIST's backend, the interposition layer should abstract out the semantics irrelevant to checking and abstract the WinAPI networking interface to a simpler form.

Interposition complexity. To reduce the interposition complexity, we implemented the interposition layer using the binary instrumentation toolkit from our previous work [25]. This toolkit takes a list of annotated WinAPI functions we want to hook and automatically generates much of the wrapper code for interposition. Under the hood, it intercepts calls to dynamically linked libraries by overwriting the function addresses in relocation tables (*import tables* in Windows terminology).

Since we check distributed systems, we only need to intercept WinAPIs relevant to these systems. Table 1 shows the categories of WinAPIs we currently hook: (1) networking APIs, such as `WSARecv()` (receiving a message), for exploring network conditions; (2) time APIs, such as `GetSystemTime()`, for discovering timers; (3) file system APIs, such as `WriteFile()` and `FlushFileBuffers()`, for injecting disk failures and simulating crashes, (4) memory APIs, such as `malloc()`, for injecting memory failures; and (5) thread APIs, such as `CreateThread()` and `SetEvent()`, for scheduling threads.

Most WinAPI wrappers are simple: they notify MODIST's backend about the WinAPI calls using an RPC call, wait for the reply from the backend, and, upon receiving the reply, they either call the underlying WinAPIs or inject failures. Table 1 shows the total lines of code in all manually-written wrappers. Each wrapper on average consists of only 67 lines of code.

IO abstraction. Controlling the Windows networking IO interface is complex for three reasons: (1) there are many networking functions; (2) these functions heavily use AIO, whose executions are hidden inside the kernel and not exposed to MODIST; and (3) these functions may produce non-deterministic results due to failures in the network. We addressed these issues using three methods: (1) abstracting similar network operations into one generic operation to narrow the networking IO interface, (2) exposing AIO to MODIST by running it synchronously in a *proxy thread*, and (3) carefully placing error injection points to avoid non-determinism.

To demonstrate our methods, we show in Figure 4 the wrapper for `WSARecv()`, a WinAPI function to synchronously or asynchronously receive data from a socket. For simplicity, we omit error-handling code and assume AIO completion is delivered using events only (events are similar to binary semaphores.)

Our wrapper first checks whether the network connection represented by the socket argument `s` is already broken by MODIST (line 5–8). If so, it simply returns an error to avoid inconsistently returning success on a broken socket. It then handles AIO (line 9–24) by creating a generic network IO structure `net_io` (line 10–14), hijacking the application's IO completion event (line 16–18), spawning a proxy thread (line 21), and issuing the AIO to the OS (line 23). The proxy thread will invoke function `mc::net_io::run()` (line 29–55). This function first notifies MODIST about the IO (line 34). Upon MODIST's reply, it either injects a failure (line 36–40), or waits for the OS to complete the IO (line 40–51). Function `run()` then reports the IO result to MODIST, which in this example is the length of the data received (47–50). Finally, it calls the wrapper to `SetEvent()` to wake up any real threads in the target system that are waiting for the IO to complete.

This wrapper example demonstrates the abstraction we use between MODIST's interposition frontend and the backend. A network IO is split into an `io_issue` and an `io_result` RPC. The first RPC, `io_issue`, expresses the IO intent of the target system to MODIST before it proceeds to a potentially blocking IO, letting MODIST avoid scheduling a disabled (i.e., blocked) IO. Its second purpose is to serve as a failure injection point. The second RPC, `io_result`, lets MODIST update the control information it tracks.

These RPC methods take the message sizes and the network connections as arguments, but not the specific message buffers or sockets, which may change across different executions. This approach ensures that MODIST's backend sees the same RPC calls when it replays the actions to recreate the same state as when it initially created the state. If MODIST detects a non-deterministic replay (e.g., a `WSARecv()` receives fewer bytes than expected), it will retry the IO by default.

There are two additional nice features about our IO abstraction: (1) it allows wrapper code sharing and therefore reduces the interposition complexity (Table 1,

```
1 : // the OS uses lpOverlap to deliver IO completion
2 : int mc_WSARecv(SOCKET s, LPWSABUF buf, DWORD nbuf,
3 :             ..., LPWSAOVERLAPPED lpOverlap, ...) {
4 :   // check if MoDist has broken this connection
5 :   if(mc_socket_is_broken(s)) {
6 :     ::WSASetLastError(WSAENETRESET);
7 :     return SOCKET_ERROR;
8 :   }
9 :   if(overlap) { // Asynchronous mode
10:     mc::net_io *io = ...;
11:     io->orig_lpOverlap = lpOverlap;
12:     io->op = mc::RECV_MESSAGE; // set IO type
13:     io->connection = ...; // Identify connection using
14:             // source <ip, port> and destination <ip, port>
15:
16:     // Hijack application's IO completion notification event
17:     io->orig_event = lpOverlap->hEvent;
18:     lpOverlap->hEvent = io->proxy_event;
19:
20:     // Create a proxy thread and run mc::net_io::run
21:     io->start_proxy_thread();
22:     // Issue asynchronous receive to the OS
23:     return ::WSARecv(s,buf,nbuf,...,io->proxy_lpOverlap,...);
24:   }
25:   // Synchronous mode
26:   ...
27: }
28: // mc::net_io code is shared among all networking IO
29: void mc::net_io::run() {// called by proxy thread
30:   mc::rpc_client *rpc = mc::current_thread_rpc_client();
31:
32:   // This RPC blocks this thead. It returns only when MoDist
33:   // wants to (1) inject a failure, or (2) complete the IO
34:   int ret = rpc->io_issue(this->op, this->connection);
35:
36:   if(ret == mc::FAILURE) {
37:     // MoDist wants to inject a failure
38:     this->orig_lpOverlap->Internal // Fake an IO failure
39:       = STATUS_CONNECTION_RESET;
40:       ... // Ask the OS to cancel the IO
41:   } else { // MoDist wants to complete this IO
42:     // Wait for the OS to actually complete the IO, because the
43:     // data to receive may still be in the real network.
44:     // This wait will not block forever, since MoDist's
45:     // dependency tracker knows there are bytes to receive
46:     ::WaitForSingleObject(this->proxy_event, INFINITE);
47:
48:     // Report the bytes actually sent or received, so MoDist's
49:     // dep. tracker knows how many bytes are in the network.
50:     int msg_size = this->orig_lpOverlap->InternalHigh;
51:     rpc->io_result(this->op, this->connection, msg_size);
52:   }
53:   // deliver IO notification to application. mc_SetEvent is
54:   // a wrapper to WinAPI SetEvent;
55:   mc_SetEvent(this->orig_event);
56: }
```

Figure 4: Simplified `WSARecv()` wrapper.

"Shared" row), (2) it abstracts away the OS-specific features and enables the backend to be OS-agnostic.

3.2 Dependency Tracking

MoDist's dependency tracker monitors how actions might affect each other. The notion of dependency is from [12]: two actions are dependent if one can enable or disable the other or if executing them in a different order leads to a different state. MoDist uses these dependencies to avoid false deadlocks (described below), to simulate failures (§3.3), and to reduce state space (§3.6).

To avoid false deadlocks, MoDist needs to compute the set of enabled actions that will not block in the OS. For determinism, MoDist schedules one action at a time and pauses all other actions (cf. §2.2). If MoDist incorrectly schedules a disabled action (such as a blocking `WSARecv()`), it will deadlock the target system because the scheduled action is blocked in the OS while all other actions are paused by MoDist.

Since the dependency tracker tries to infer whether the OS scheduler would block a thread in a WinAPI call (recall that the interposition layer exposes AIOs as threads), it unsurprisingly resembles an OS scheduler and replicates a small amount of the control data in the OS and the network. To illustrate how it works, consider the `WSARecv()` wrapper in Figure 4. The dependency tracker will track precisely how many bytes are sent and received for each network connection using the `io_result` RPC (line 50). If a thread tries to receive a message (line 34) when none is available, the dependency tracker will mark this thread as disabled and place it on the wait queue of the connection. Later, when a `WSASend()` occurs at the other end of the connection, the dependency tracker will remove this thread from the wait queue and mark it as enabled. When MoDist schedules this thread by replying to its RPC `io_issue`, the thread will not block at line 45 because there is data to receive. In addition to network control data, the dependency tracker also tracks threads, locks, and semaphores.

3.3 Failure Simulation

When requested by the model checking engine, MoDist's failure simulator injects five categories of failures: API failures (e.g., `WriteFile()` returns "disk error"), message reordering,* message loss, network partitions, and machine crashes. Simulating API failures is the easiest: MoDist simply tells the interposition layer to return an error code. Reordering messages is also easy since the model checking engine already explores different orders of actions. To simulate different crash scenarios, we used techniques from our previous work [38, 39] to permute the disk writes that a system issues.

Simulating network failures is more complicated due to the consistency and determinism requirement. We first tried a naïve approach: simply closing sockets to simulate connection failures. This approach did not work well because we frequently experienced inconsistent failures:

*Message reordering is not a failure, but since it is often caused by abnormal network delay, for convenience we consider it as a failure.

the "macro" failures we want to inject (e.g., network partition) map to not one but a set of "micro" failures we can inject through the interposition layer (e.g., a failed `WSARecv()`). For example, to break a TCP connection, we must carefully fail all pending asynchronous IOs associated with the connection at both endpoints. Otherwise, the target system may see an inconsistent connection status and crash, thus generating a false positive.

We also frequently experienced non-deterministic failures because the OS detects failures using non-deterministic timeouts. Consider the following actions:

1. Process P_1 calls WSASend(P_2, message).
2. Process P_2 calls asynchronous WSARecv(P_1).
3. MoDist breaks the connection between P_1 and P_2.

P_2 may or may not receive the message, depending on when P_2's OS times out the broken connection.

Our current approach ensures that failure simulation is consistent and deterministic as follows. We know the exact set of real or proxy threads that are paused by MoDist in `rpc->io_issue()` (Figure 4, line 34). To simulate a network failure, we inject failures to all these threads, and we do so immediately to avoid any non-deterministic kernel timeouts. Note that doing so in the example above will not cause us to miss the scenario where P_2 receives the message before the connection breaks; MoDist will simply explore this scenario in a different execution where it completes P_2's asynchronous `WSARecv()` first (by replying to P_2's `io_issue()` RPC), and then breaks the connection between P_1 and P_2.

3.4 Virtual Clock

MoDist's virtual clock manager injects timeouts when requested by the model checking engine and provides a consistent view of the clock to the target system. A side benefit of virtual clock is that, the target system may run faster because the virtual clock manager can fast forward time. For example, when the target system calls `sleep(1000)`, the virtual clock manager can add 1000 to its current virtual clock and let the target system wake up immediately.

Discovering Timeouts. To detect bugs in rarely tested timeout handling code, we want to discover as many timers as possible. This task is made difficult because system code extensively uses *implicit timers* where the code first gets the current time (e.g., by calling `gettimeofday()`), then checks if a timeout occurs (e.g., using an `if`-statement). Figure 5 shows a real example in Berkeley DB.

Since implicit timers do not use OS APIs to check timeouts, they are difficult to discover by a model checker. Previous work [19, 27, 38] requires users to manually annotate implicit timers.

```
// db-4.7.25.NC/repmgr/repmgr_sel.c
int __repmgr_compute_timeout(ENV *env, timespec * timeout)
{
  db_timespec now, t;
  ...  // Set t to the first due time.
  if (have_timeout) {
    __os_gettime(env, &now, 1); // Query current time.
    if (now >= t) // Timeout check, immediately follows the query.
      *timeout = 0; // Timeout occurs.
    else
      *timeout = t - now; // No timeout.
  }
  ...
}
```

Figure 5: An implicit timer in Berkeley DB (after macro expansion and minor editing).

To discover implicit timers automatically, we developed a *static* symbolic analysis technique. It is based on the following two observations:

1. Programmers use time in *simple* ways. For example, they explicitly label time values (e.g., `db_timespec` in Figure 5), they do simple arithmetic on time values, and they generally do not cast time values to pointers and other unusual types. This observation implies that simple static analysis is sufficient to track how a time value flows.
2. Programmers check timeouts *soon* after they query the current time. The intuition is that programmers want the current time to be "fresh" when they check timeouts. This observation implies that our analysis only needs to track a short flow of a time value (e.g., within three function calls) and may stop when the flow becomes long.

We analyzed how time values are used in Berkeley DB version 4.7.25. We found that Berkeley DB mostly uses "+," "-," and occasionally "*" and "/" (for conversions, e.g., from seconds to milliseconds). In 12 out of 13 implicit timers, the time query and time check are within a few lines.

Our analysis resembles symbolic execution [3, 4, 13, 31]. It has three steps: (1) *statically* analyze the code of the target system and find all system calls that return time values; (2) track how the time values flow to variables; and (3) upon a branch statement involving a tracked time value, use a simple constraint solver to generate symbolic values to make both branches true. To show the idea, we use a source code instrumentation example. In Figure 5, our analysis can track how time flows from "`__os_gettime`" to "`if (now >= t)`," and replace the "`__os_gettime`" line in Figure 5 with

```
mc::rpc_client *rpc = mc::current_thread_rpc_client()
now = rpc->gettime(/*timer=*/t);
```

This RPC call tells the virtual clock manager that a timer fires at `t`; the virtual clock manager can then return a time value smaller than `t` for one execution, and greater than `t` for another execution, to explore both possible execution paths. We implemented our analysis using the Phoenix compiler framework [30].

Since our analysis is static, it avoids the runtime overhead of instrumenting each load and store for tracking symbolic values and thus is much simpler than dynamic symbolic execution tools [3, 4, 13, 31], which often take iterations to become stable [3, 4]. Note our analysis is unsound, as with other symbolic analysis tools, in that it may miss some timers and thus miss bugs. However, it will not introduce false positives because the virtual clock manager ensures the consistency of time.

Ensuring Consistent Clock. A consistent clock is crucial to avoid false positives. For example, the safety of the lease mechanism [14] requires that the lessee timeouts before the lessor; reversing the order may trigger "bugs" that never occur in practice. We actually encountered a painful false positive due to a violation of this safety requirement when checking PacificA.

To maintain consistent time, the virtual clock manager sorts all timers in the target system from earliest to last based on when these timers will fire. When the model checking engine decides to fire a timer, it will systematically choose one of several timers that fall in the range of $[T, T+E]$, where T is the earliest timer and E is a configurable clock error allowed by the target system. This mechanism lets MoDist explore interesting timer behaviors while not deviating too much from real timer-triggered executions.

3.5 Global Assertion

We have implemented global assertions leveraging our previous work D^3S [25]. D^3S enables transparent predicate checking of a running distributed system. It provides a simple programming interface for developers to specify global assertions, interposes both user-level functions and OS system calls in the target system to expose its runtime state as state tuples, and collects such tuples as globally consistent snapshots for evaluating assertions. To use D^3S, developers need to specify the functions being interposed, the state tuples being retrieved from function parameters, and a sequential program that takes a complete state snapshot as input to evaluate the predicate. D^3S compiles such assertions into a state exposing module, which is injected into all processes of the target system, and a checking module, which contains the evaluation programs and outputs checking results for every constructed snapshot.

MoDist incorporates D^3S to enable global assertions, with two noticeable modifications. First, we simplify D^3S by letting each node transmit state tuples synchronously to MoDist's checking process, which verifies assertions *immediately*. Previously, because nodes may transmit state tuples concurrently, D^3S must, before checking assertions, buffer each received tuple until all tuples causally dependent before that tuple have been received. Since MoDist runs one action at a time, it no longer needs to buffer tuples. Second, while D^3S uses a Lamport clock [23] to totally order state tuples into snapshots, MoDist uses a *vector clock* [26] to check more global snapshots.

3.6 State Space Exploration

MoDist maintains a queue of the state/action pairs to be explored. Due to the complexity of a distributed system, it is often infeasible for MoDist to exhaust the state space. Thus, it is key to decide which state/action pairs to add to the queue and the order in which they are explored.

MoDist tags each action with a vector clock and implements a customizable modular framework for exploring the state space so different reduction techniques and heuristics can be incorporated. This is largely inspired by our observation that the effectiveness of various strategies and heuristics is often application-dependent.

The basic state exploration process is simple: MoDist takes the first state/action pair $\langle s, a\rangle$ from the queue, steers the system execution to state s if that is not the current state, applies the action a, reaches a new state s', and examines the new resulting state for errors. It then calls a customizable function `explore`, which takes the entire *path* from the initial state to s and then s', where each state is tagged with its vector clock and each state transition is tagged with the action corresponding to the transition. For s', all enabled actions are provided to the function. The function then produces a list of state/action pairs and indicates whether the list should be added to the front of the queue or the back. MoDist then inserts the list into the queue and repeats the steps.

MoDist has a natural bias towards exploring $\langle s, a\rangle$ pairs where s is the state MoDist is in. This default strategy will save the cost of replaying the trace to reach the state in the selected state/action pair.

Now we show how various state exploration strategies and heuristics can be implemented in the MoDist framework.

Random. Random exploration with a bounded maximum path length explores a random path up to a bounded path length and then starts from the initial state for another random path. The `explore` function works as follows: if the current path has not exceeded the bound, the function will randomly pick an enabled action a' at the new state s' and has $\langle s', a'\rangle$ inserted to the end of the queue (note that the queue is empty). If the current path has reached the bound, the function will randomly choose an enabled action a_0 in the initial state s_0, and has $\langle s_0, a_0 \rangle$ inserted to the end of the queue.

DFS and BFS. For Depth First Search (DFS) and Breadth First Search (BFS), the `explore` function simply inserts $\langle s', a' \rangle$ for every enabled action a' in state s'. For DFS, the new list is inserted at the front of the queue, while for BFS at the back. Clearly, DFS is more attractive since MoDist does not have to replay traces often to recreate states.

DPOR. For dynamic partial order reduction (DPOR), the `explore` function works as follows. Let a be the last action causing the transition from s to s'. The function looks at every state s_p before s on the path and the action a_p taken at that state. If a is enabled at s_p (i.e., if s and s_p are concurrent judged by the vector clocks) and a does not commute with a_p (i.e., the different orders of the two actions could lead to different executions), we record $\langle s_p, a \rangle$ in the list of pairs to explore. Once all states are examined, the function returns the list and has MoDist insert the list in the queue.

By specifying how the list is inserted, the function could choose to use DFS or BFS on top of DPOR. Also, by ordering the pairs in the list differently, MoDist will be instructed to explore the newly added branches in different orders (e.g., top-down or bottom-up). The default is DFS again to avoid the cost of recreating states. We further introduce *Bounded DPOR* to refer to the variation of DPOR with bounds on DFS for a more balanced state-space exploration.

The `explore` function can be constructed to favor certain actions (e.g., crash events) over others, to bound the exploration in various ways (e.g., the path length and the number of certain actions on the path), and to focus on a subset of possible actions.

4 Evaluation

We have applied MoDist to three distributed systems: (1) Berkeley DB, a widely used open-source database (a version with replication); (2) MPS, a closed source Paxos [22] implementation built by a Microsoft product team and has been deployed in commercial data centers for more than two years; and (3) PacificA, a mature implementation of a primary-backup replication protocol we developed. We picked Berkeley DB and MPS because of their wide deployment and importance and PacificA because it provides an interesting case study where the developers apply model checking to their own systems.

System	KLOC	Protocol	Impl.	Total
Berkeley DB	172.1	2	5	7
MPS	53.5	2	11	13
PacificA	12	6	9	15
Total	237.6	10	25	35

Table 2: *Summary of errors found.* The **KLOC** (thousand lines of code) column shows the sizes of the systems we checked. We separate protocol-level bugs (**Protocol**) and implementation-level bugs (**Impl.**), in addition to reporting the total (**Total**). 31 of the 35 bugs have been confirmed by the developers.

Table 2 summarizes the errors we found, all of which are previously unknown bugs. We found a total of 35 errors, 10 of which are *protocol-level bugs* that occur only under rare interleavings of messages and crashes; these bugs reflect flaws in the underlying communication protocols of the systems. *Implementation bugs* are those that can be caused by injecting API failures. All MPS and PacificA bugs were confirmed by the developers. Three out of seven Berkeley DB bugs, including one protocol-level bug, were confirmed by Berkeley DB developers; we are having the rest confirmed. These unconfirmed bugs are likely real bugs because we can reproduce them without MoDist by manually tweaking the executions and killing processes according to the traces from MoDist.

While other tools (e.g., a static analyzer) can also find implementation bugs, MoDist has the advantage of not generating false positives. In addition, it can expose the effects of these bugs, helping prioritize fixing.

In the rest of this section, we describe our error detection methodology, the bugs we found, MoDist's coverage results and runtime overhead, and the lessons we have learned.

4.1 Experimental Methodology

Test driver. Model checking is most effective at checking complicated interactions between a small number of objects. Thus, in all tests we run, we use several processes servicing a bounded number of requests. Since the systems we check came with test cases, we simply use them with minor modifications.

Global assertions. By default, MoDist checks failstop errors. To check the correctness properties of a distributed system, MoDist supports user supplied global assertions (§2). For the replication systems we checked, we added two types of assertions. The first type was global predicates for the safety properties. For example, all replicas agree on the same sequence of commands. The second type of predicates check for liveness. True liveness conditions cannot be checked by execution monitoring so we instead approximate them by checking for progress in the system: we expect the target system to make progress in the absence of failures. In the end, we did not find any bug that violated the safety properties in any of the systems, probably reflecting the relative maturity of these systems. However, we did find bugs that violated liveness global assertions in every system.

Search strategy. MoDist has a set of built-in search strategies; no single strategy works the best. We have combined these strategies in our experiments for discovering bugs effectively. For example, we can first perform random executions (Random) on the system and inject the API failures randomly to get the shallow implementation bugs. We can then use the DPOR strategy with randomly chosen initial paths to explore message orders systematically. We can further add crash and recovery events on top of the message interleaving, starting from a single crash and gradually increasing the number of crashes, to exercise the system's handling of crash and recovery. We can run these experiments concurrently and fine-tune the strategies.

Terminology. Distributed systems use different terminologies to describe the roles the nodes play in the systems. In this paper, we will use *primary* and *secondary* to distinguish the replicas in the systems. They are called *master* and *client* respectively in Berkeley DB documents. In the Paxos literature, a primary is also called a *leader*.

4.2 Berkeley DB: a Replicated Database

Berkeley DB is a widely used open source transactional storage engine. Its latest version supports replication for applications that must be highly available. In a Berkeley DB replication group, the primary supports both reads and writes while secondaries support reads only. New replicas can join the replication group at any time.

We checked the latest Berkeley DB production release: 4.7.25.NC. We use `ex_rep_mgr`, an example application that comes with Berkeley DB as the test driver. This application manages its data using the Berkeley DB Replication Manager. Our test setup has 3 to 5 processes. They first run an election. Once the election completes, the elected primary inserts data into the replicated database, reads it back, and verifies that it matches the data inserted.

Results and Discussions. We found seven bugs in Berkeley DB: four were triggered by injecting API failures, one was a dangling pointer error triggered by the primary waiting for multiple ACK messages simultaneously from the secondaries, and the remaining two were protocol-level bugs, which we describe below.

The first protocol-level bug causes a replica to crash due to an "unexpected" message. The timing diagram of this bug is depicted in Figure 6. Replica *C* is the original primary. Suppose a new election is launched, resulting in replica *A* becoming the new primary. Replica *A* will broadcast a `REP_NEWMASTER` message, which means "I am the new primary." After replica *B* receives this message, it tries to synchronize with the new primary and sends *A* a `REP_UPDATA_REQ` message to get the up-to-date data. Meanwhile, *C* processes `REP_NEWMASTER` by first broadcasting a `REP_DUPMASTER` message, which means "duplicate primary detected," and then degrading itself to a secondary. Broadcasting a `REP_DUPMASTER` message is necessary to ensure that all other replicas know that *C* is not primary anymore. When *A* processes `REP_DUPMASTER`, it has to give up its primary role because it cannot make sure that it is the latest primary. Soon *A* receives the delayed but not-outdated `REP_UPDATA_REQ` message from *B*. Replica *A* panics at once, because such message should only be received by primary. Such panics occur whenever a delayed `REP_UPDATA_REQ` message arrives at a recently degraded primary.

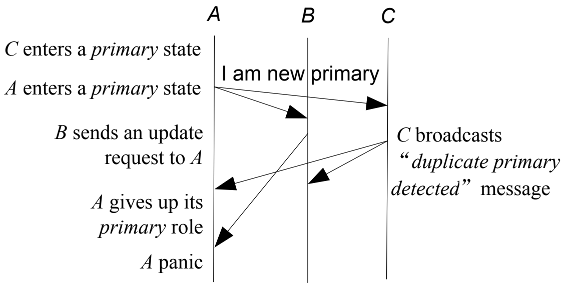
A
B
C
C enters a primary state
A enters a primary state
I am new primary
B sends an update request to A
C broadcasts "duplicate primary detected" message
A gives up its primary role
A panic

Figure 6: Timing Diagram of Message Exchanges in a Berkeley DB Replication Bug.

The second protocol level bug is more severe: it causes permanent failures in leader election due to a primary crash when all secondaries believe they cannot be primaries. Suppose replica *A* is the original primary and is synchronizing data with secondaries *B* and *C*. Normally synchronization works as follows. *A* sends a `REP_PAGE` message with the modified database page to *B* and *C*. Upon receipt of this message, *B* and *C* transit to log recovery state by setting the `REP_F_RECOVER_LOG` flag. *A* then sends a `REP_LOG` message with the updated log records. However, if *A* crashes before it sends `REP_LOG`, *B* and *C* will never be able to elect a new primary because, in Berkeley DB's replication protocol, a replica in log recovery is not allowed to be a primary.

4.3 MPS: Replicated State Machine Library

MPS is a practical implementation of a replicated state machine library. The library has been used for over two years in production clusters of more than 100K machines for maintaining important system metadata consistently and reliably. It consists of 8.5K lines of C++ code for the communication protocol, and 45K for utilities such as networking and storage.

At the core of MPS is a distributed Paxos protocol for consensus [22]. The protocol is executed on a set of machines called *replicas*. The goal of the protocol is to have replicas agree on a sequence of deterministic commands

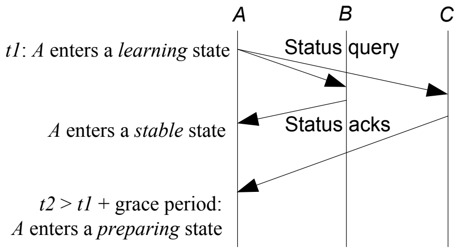

Figure 7: The Timing Diagram of Message Exchange in MPS Bug 1.

and execute the commands in the same sequence order. Because all replicas start with the same initial state and execute the same sequence of commands, consistency among replicas is guaranteed.

The MPS consensus protocol is leader (primary) based. While the protocol ensures safety despite the existence of multiple primaries, a single primary is needed for the protocol to make progress. A replica can act as a primary using a certain *ballot number*. A primary accepts requests from clients and proposes those requests as *decrees*, where decree numbers indicate the positions of the requests in the sequence of commands that is going to be executed by the replicated state machine. A decree is considered *committed* when the primary gets acknowledgment from a quorum (often a majority) of replicas indicating that they have accepted and persistently stored the decree.

If a replica receives a message that indicates that a decree unknown to the replica is committed, then the replica enters a *learning* phase, in which it learns the missing decrees from other replicas.

When an existing primary is considered to have failed, a new primary can be elected. The new primary will use a higher ballot number and carry out a *prepare* phase to learn the decrees that could have been committed and ensure no conflicting decrees are proposed. For each replica, a proposal with a higher ballot number overwrites any previous proposal with lower ballot numbers.

Our test setup consists of 3 replicas, proposing a small number of decrees.

Results and Discussions. We found 13 bugs in MPS, 11 are implementation bugs that crash replicas, and the other two bugs are protocol-level bugs.

The first protocol-level bug reveals a scenario that leads to state transitions that are not expected by the developers (as demonstrated by the assertion that rules out the transition). MPS has a simple set of states and state transitions. A replica is normally in a *stable* state. When it gets indication that its state is falling behind (i.e., missing decrees), it enters a *learning* state. In the learning state, it fetches the decrees from a quorum of replicas. Once it brings its state up to date with what it receives from a quorum of replicas, it checks whether it should become a primary: if the primary lease expires, then the replica will compete to be a primary by entering a *preparing* state; otherwise, it will return to a stable state. There is an assertion in the code (and also in the design document for MPS) that the state transition from stable to preparing is impossible.

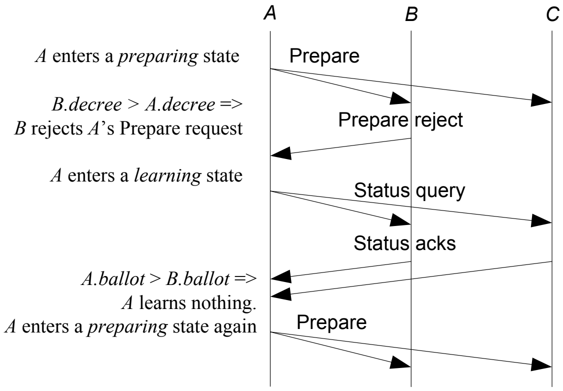

Figure 8: The Timing Diagram of Message Exchange in MPS Bug 2.

Figure 7 shows the MoDist-generated scenario that triggers the assertion failure. The following is a list of steps that lead to the violation. Consider the case where the system consists of three replicas *A*, *B*, and *C*, where any two of them form a quorum. Replica *A* enters the learning state because it realizes that it does not have the information related to some decree numbers. This could be due to the receipt of a message that indicates that the last committed decree number is at least *k*, while *A* knows only up to some decree number less than *k*. *A* then sends a status query to *B* and *C*. *A* receives the response from *B* and learns all the missing decrees. Since *A* and *B* form a quorum, *A* enters the stable state. *C* was the primary. *C*'s response to *A* status query was delayed, and the primary lease becomes expired on *A*. At some later point, *C*'s response arrives. The implementation will handle that message as if *A* were in the learning state. After *A* is done, it notices that the primary lease has expired and transitions into the preparing state, causing the unexpected state transition. As a result, *A* crashes and reboots.

The second protocol-level bug is a violation of a global liveness assertion. It is triggered during primary election under the following scenario: replica *A* has accepted a decree with ballot number 2 and decree number 1, while replica *B* only has ballot number 1, but accepted a decree of decree number 2.

The following series of events lead to this problematic scenario: *B* is a primary with ballot number 1, it proposes a decree with decree number 1 and the decree is accepted by all replicas including *A* and *B*. It then proposes another decree with decree number 2, which is accepted only on *B*. *B* fails before *A* gets the proposal. *A* then becomes a primary with ballot number 2, learns the decree with decree number 1, re-proposes it with a ballot number 2.

Figure 8 shows the timing diagram continuing from this scenario. *B* comes back, receives the prepare request from *A*, and sends a rejection to *A* because *B* thinks *A* is not up-to-date given that *B* has a higher decree number. After getting the rejection, *A* enters a learning state. In the learning state, even if *B* returns the decree with decree number 2, *A* will reject it because it has a lower ballot number. *A* will consider itself up-to-date and enter the preparing state again with a yet higher ballot number. This continues as *A* keeps increasing its ballot number, but unable to have new decrees committed, triggering a liveness violation.

The problem in this scenario is due to the inconsistency of the views on what constitutes a newer state between the *preparing phase* and the *learning phase*: one view uses a higher ballot number, while the other uses a higher decree number. The inconsistency is exposed when one has a higher decree number, but a lower ballot number than the other.

4.4 PacificA: a Primary-Backup Replication Protocol

PacificA [24] is a large-scale storage system for semi-structured data. It implements a Primary-Backup protocol for data replication. We used MoDist to check an implementation of PacificA's replication protocol. This implementation consists of 5K lines of C++ code for the communication protocol and 7K for utilities.

PacificA uses a variety of familiar components including two-phase commit for consistent replica updates, perfect failure detection, replica group reconfiguration to handle node failures, and replica reconciliation for nodes rejoining a replica group.

Our test setup for PacificA has 4 processes: 1 master that maintains global metadata, 2 replica nodes that implement the replication protocol, and 1 client that updates the system and drives the checking process. Figure 2 shows the configuration file.

Results and Discussions. We found 15 bugs in PacificA: 9 are implementation bugs that cause crashes and 6 are protocol-level bugs. We managed to find more protocol-level bugs in PacificA than in other systems for two reasons: (1) since we built the system, we could quickly fix the bugs MoDist found then re-run MoDist to go after other bugs; and (2) we could check more global assertions for PacificA.

The most interesting bug we found in PacificA prevents PacificA from making progress. It is triggered by a node crash followed by a replication group reconfiguration. A primary replica keeps a list of prepared updates (i.e., updates that have been prepared on all replicas, but not yet committed); a secondary replica does not have this data structure. When a primary crashes, a secondary will try to take over and become the new primary. If the crash happens in the middle of a commit operation that leaves some commands prepared but not yet committed, the new primary will try to re-commit all prepared updates by sending the "prepare" messages to the remaining secondary replicas. Unfortunately, PacificA did not put these newly prepared updates into the prepared update list. This prevents all the following updates from getting committed because of a hole in the prepared update list.

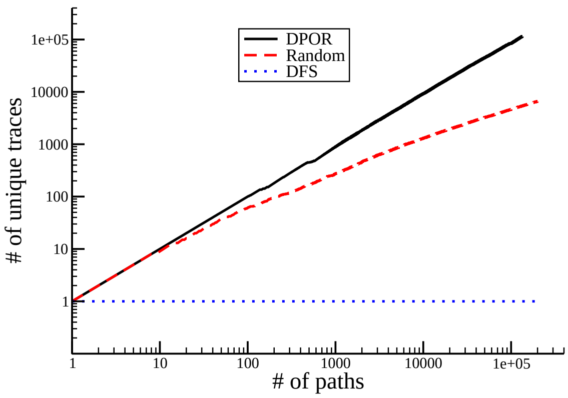

Figure 9: Partial order state coverage of different exploration strategies.

4.5 State Coverage

To evaluate the state-space exploration strategies described in §3.6, we measured *state coverage*: the number of unique states a strategy could explore after running a fixed number of execution paths. We examined the coverage of two types of states:

1. *Partial order traces [12].* Since two paths with the same partial order are equivalent, the number of different partial order traces provides an upper bound on the number of unique behaviors a strategy can explore.
2. *Protocol states*. These states capture the more important protocol behaviors of a distributed system.

We did two experiments, both on MPS: one with a small partial order state space and the other with a nearly unbounded state space. These two state spaces give an idea of how sensitive the strategies are to state-space sizes. No crash was injected during the evaluation.

In the first experiment, we made the state space small using a configuration of two nodes, each receiving up to two messages. Figure 9 shows the number of unique partial order traces with respect to the number of paths explored. (Note that both axes are in log scale.) DPOR

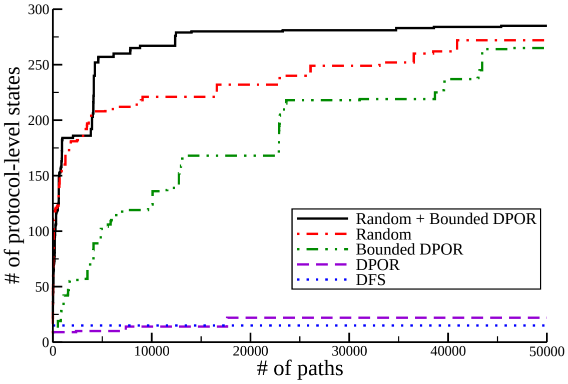

Figure 10: Protocol state coverage of different exploration strategies.

shows a clear advantage: it exhausted all 115,425 traces after 134,627 paths (the small redundancy was due to an approximation in our DPOR implementation.) The Random strategy explored 6,614 unique traces or 5.7% of the entire state space after 200,000 paths. DFS is the worst: all the 200,000 paths were partial order equivalent and corresponded to only one partial order trace.

In the second experiment, we used a nearly unbounded partial order state space with three MPS nodes sending and receiving an unbounded number of messages. We bounded the maximum decree (two decrees) and the maximum path length (40,000 actions) to make the execution paths finite. Since the state space was large, it was unlikely that Random ever explored a partial order trace twice. As a result, DPOR behaved the same as Random. (This result is not shown.)

While partial order state coverage provides an upper bound on the unique behaviors a strategy explores, different partial order traces may still be redundant and map to the same protocol state. Thus, we further measured the protocol state coverage of different exploration strategies. We defined the *protocol state* of MPS as a tuple $\langle state, ballot, decree \rangle$, where the *state* could be `initializing`, `learning`, `stable primary`, or `stable secondary`.[*]

Figure 10 shows the protocol states covered by the first 50,000 paths explored in each strategy, using the MPS configuration from the second experiment. DFS had the worst coverage: it found no new states after exploring the first path. The reason is, when the state space is large, DFS tends to explore a large number of paths that differ only at the final few steps; these paths are often partial-order equivalent. DPOR performed almost equally badly: it found less than 30 protocol states. This result is not surprising for two reasons: (1) different partial order traces might correspond to the same protocol state and (2) DPOR is DFS-based, thus suffers the same problem as DFS when the state space is large.

In Bounded DPOR, protocol-level redundancy is partially conquered by the bounds on backtracks. As shown in Figure 10, the protocol-level state coverage of Bounded DPOR was larger than that of DPOR by an order of magnitude, in the first 50,000 paths.

Surprisingly, the Random strategy yielded better coverage than DFS, DPOR, and even Bounded DPOR. The reason is that Random is more balanced: it explores actions anywhere along a path uniformly, therefore it has a better chance to jump to a new path early on and explores a different area of the state space.

These results prompted us to develop a hybrid *Random + Bounded DPOR* search strategy that works as follows. It starts with a random path and explores the state space with Bounded DPOR. We further bound the total number of backtracks so that the Bounded DPOR exploration ends. Then, a new round of Bounded DPOR exploration starts with a new random path. *Random + Bounded DPOR* inherits both the balance of Random and the thoroughness of DPOR to cover the corner cases. Both the round number of DPOR explorations and the bound of the total number of backtracks are customizable, reflecting a bias towards Random or towards DPOR. As shown in Figure 10, the *Random + Bounded DPOR* strategy with a round number 100 performed the best.

4.6 Performance

In our performance measurements, we focused on three metrics: (1) MoDist's path exploration speed; (2) the speedup due to the virtual clock fast-forward; and (3) the runtime overhead MoDist adds to the target system, including interposition, RPC, and backend scheduling.

We set up our experiments as follows. We ran MoDist with two different search strategies: RANDOM and DPOR. For each search strategy, we let MoDist explore 1K execution paths and recorded the running times. We repeated this experiment 50 times and took the average. We used Berkeley DB and MPS as our benchmarks, using identical configurations as those used for error detection. We ran our experiments on a 64-bit Windows Server 2003 machine with dual Intel Xeon 5130 CPU and 4GB memory. We measured all time values using `QueryPerformanceCounter()`, a high-resolution performance counter.

It appears that we should measure MoDist's overhead by comparing a system's executions with MoDist to those without. However, due to nondeterminism, we cannot compare these two directly: the executions without MoDist may run different program paths than those with MoDist. Moreover, repeated executions of the same testcase without MoDist may differ; we did ob-

[*] We also measured the coverage of *global* protocol states, which consist of protocol states of each node in a consistent global snapshot. The results were similar and not shown.

System	Strategy	Real (s)	Sleep (s)	Speedup	Overhead (absolute and relative)
Berkeley DB	RANDOM	$1,717 \pm 14$	$38,204 \pm 193$	22.3 ± 0.2	$302 \pm 1s$ ($17.7 \pm 0.1\%$)
Berkeley DB	DPOR	$1,658 \pm 24$	$36,402 \pm 5,137$	22.1 ± 3.2	$301 \pm 17s$ ($18.2 \pm 0.9\%$)
MPS	RANDOM	$1,661 \pm 20$	$240,568 \pm 1,405$	145 ± 2	$825 \pm 11s$ ($49.9 \pm 0.2\%$)
MPS	DPOR	$1,853 \pm 116$	$295,435 \pm 45,659$	159 ± 19	$1,048 \pm 108s$ ($56.5 \pm 2.6\%$)

Table 3: MODIST's performance. All numbers are of the form *average* ± *standard deviation*.

serve a large variance in MPS's execution times and final protocol states. Thus, we evaluated MODIST's overhead by running a system with MODIST and measuring the time spent in MODIST's components.

Table 3 shows the performance results. The **Real** column shows the time it took for MODIST to explore 1K paths of Berkeley DB and MPS with RANDOM and DPOR strategies; the exploration speed is roughly two seconds per path and does not change much for the two different search strategies. The **Sleep** column shows the time MODIST saved using its virtual clock when the target systems were asleep; we would have spent this amount of extra time had we run the same executions without MODIST. As shown in the table, the real execution time is much smaller that the sleep time, translated into significant speedups (Column **Speedup**, computed as **Sleep/Real**). The **Overhead** column in this table shows the time spent in MODIST's interposition, RPC, and backend scheduling. For Berkeley DB, MODIST accounts for about 18% of the real execution time. For MPS, MODIST accounts for a higher percentage of execution time (up to 56.5%) because the MPS testcase we used is almost the worst case for MODIST: it only exercises the underlying communication protocol and does no real message processing. Nonetheless, we believe such overhead is reasonable for an error detection tool.

4.7 Lessons

This section discusses the lessons we learned.

Real distributed protocols are buggy. We found many protocol-level bugs and we found them in every system we target, suggesting that real distributed protocols are buggy. Amusingly, these protocols are based on theoretically sound protocols; the bugs are introduced when developers filled in the unspecified parts in the protocols in practice.

Controlling all non-determinism is hard. Systematic checking requires control of non-determinism in the target system. This task is very hard given the non-determinism in the OS and network, the wide API interface, the many possible failures and their combinations, and MODIST's goal of reducing intrusiveness to the target system. We have had bitter experiences debugging non-deterministic errors in Berkeley DB, which uses process id, memory address, and time to generate random numbers, and in MPS, which randomly interferes with the default Windows firewall. Among all, making the Windows socket APIs deterministic was the most difficult; the interface shown in §3 went through several iterations. Our own experiences show that controlling all non-determinism is much harder than merely capturing it as in replay-debugging tools.

Avoid false positives at all cost. False positives may take several days to diagnose. Thus, we want to avoid them, even at the risk of *missing errors.*

Leverage domain knowledge. In a sense, this entire paper boils down to leveraging the domain knowledge of distributed systems to better model-check them. The core idea of model checking is simple: explore all possible executions; a much more difficult task is to implement this idea effectively in an application domain.

When in doubt, reboot. When we checked MPS, we were surprised by how robust it was. MPS uses a defensive programming technique that works particularly well in the context of distributed replication protocols. MPS extensively uses local assertions, reboots when any assertion fails, and relies on the replication protocol to recover from these eager reboots. This recovery mechanism makes MPS robust against a wide range of failures. Of course, rebooting is not without penalty: if a primary reboots, there could be noticeable performance degradation, and the system also becomes less fault tolerant.

5 Related Work

5.1 Model Checking

Model checkers have previously been used to find errors in both the design and the implementation of software [1, 6, 12, 17–19, 27, 28, 34, 38, 39]. Traditional model checkers require users to write an abstract model of the target system, which often incurs large up-front cost when checking large systems. In contrast, MODIST is an implementation-level model checker that checks code directly, thus avoids this cost. Below we compare MODIST to implementation-level model checkers.

Model checkers for distributed system. MODIST is most related to model checkers that check real distributed system implementations. CMC [27] is a stateful model checker that checks C code directly. It has been used to check network protocol implementations [27] and file systems [38]. However, to check a system, CMC requires invasive modifications to run the system inside CMC's address space [39]. MaceMC [19] uses bounded depth first search combined with random walk to find safety and liveness bugs in a number of network protocol implementations written in a domain-specific language. Compared to these two checkers, MODIST directly checks live, unmodified distributed systems running in their native execution environments, thus avoids the invasive modifications required by CMC, and the language restrictions [20] enforced by MaceMC.

CrystalBall [37] detects and avoids errors in deployed distributed systems using an efficient global state collection and exploration technique. While CrystalBall is based on MaceMC and thus checks only systems written in the Mace language [20], its core technique may be portable to MODIST's model checking framework to improve the reliability of general distributed systems.

Other software model checkers. We compare MODIST to other closely related implementation-level model checkers. Our transparent checking approach is motivated by our previous work EXPLODE [39]. However, EXPLODE focuses on storage systems and does not check distributed systems.

To our best knowledge, VeriSoft [12] is the first implementation-level model checker. It systematically explores the interleavings of concurrent C programs, and uses partial order reduction to soundly reduce the number of states it explores. It has been used to check industrial-strength programs [5].

Chess [28] is a stateless model checker for exploring the interleavings of multi-threaded programs. To avoid perturbing the target system, it also interposes on WinAPIs. In addition, Chess uses a *context-bounding* heuristic and a *starvation-free* scheduler to make its checking more efficient. It has been applied to several industry-scale systems and found many bugs.

ISP [35] is an implementation-level model checker for MPI programs. It controls a MPI program by intercepting calls to MPI methods and reduces the state-space it explores using new partial order reduction algorithms.

All three systems focus on checking interleavings of concurrent programs, thus do not address issues on checking real distributed systems, such as providing a transparent, distributed checking architecture and enabling consistent and deterministic failure simulation

5.2 Replay-based debugging

A number of systems [11, 21, 32], including our previous work [15, 25], use deterministic replay to debug distributed system. These approaches attack a different problem: when a bug occurs, how to capture its manifestation so that developers can reproduce the bug. Combined with fault injection, these tools can be used to detect bugs. Like these systems, MODIST also provides reproducibility of errors. Unlike these systems, MODIST aims to *proactively* drive the target system into corner-cases for errors in the testing phase before the system is deployed. MODIST uses the instrumentation library in our previous work [25] to interpose on WinAPIs.

5.3 Other error detection techniques

We view testing as complementary to our approach. Testing is usually less comprehensive than our approach, but works "out of the box." Thus, there is no reason not to use both testing and MODIST together.

There has been much recent work on static bug finding (e.g., [1, 2, 7, 8, 10, 33]). Roughly speaking, because dynamic checking runs code, it is limited to just executed paths, but can more effectively check deeper properties implied by the code (e.g., two replicas are consistent). The protocol-level errors we found would be difficult to find statically. We view static analysis as complementary: easy enough to apply such that there is no reason not to use them together with MODIST.

Recently, symbolic execution [3, 4, 13, 31] has been used to detect errors in real systems. This technique is good at detecting bugs caused by tricky input values, whereas our approach is good at detecting bugs caused by the non-deterministic events in the environment.

6 Conclusions

MODIST represents an important step in achieving the ideal of model checking unmodified distributed system in a transparent and effective way. Its effectiveness has been demonstrated by the subtle bugs it uncovered in well-tested production and deployed systems.

Our experience shows that it requires a combination of art, science, and engineering. It is an art because various heuristics must be developed for finding delicate bugs effectively, taking into account the peculiarity of complex distributed systems; it is a science because a systematic, modular approach with a carefully designed architecture is a key enabler; it involves heavy engineering effort to interpose between the application and the OS, to model and control low-level system behavior, and to handle system-level non-determinism.

Acknowledgement

We thank Huayang Guo and Liying Tang for their help in evaluating MODIST, Jian Tang for his contributions to an earlier version of the system, and our colleagues at Microsoft Research Silicon Valley and the System Research Group at Microsoft Research Asia for their comments and support. We especially thank Stephen A. Edwards for extensive edits and Stelios Sidiroglou-Douskos for detailed comments. We are also grateful to the anonymous reviewers for their valuable feedback and to our shepherd Steven Hand for his guidance.

References

[1] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the Eighth International SPIN Workshop on Model Checking of Software (SPIN '01)*, pages 103–122, May 2001.

[2] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.

[3] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security (CCS '06)*, pages 322–335, Oct.–Nov. 2006.

[4] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 209–224, Dec. 2008.

[5] S. Chandra, P. Godefroid, and C. Palm. Software model checking in practice: an industrial case study. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*, pages 431–441, May 2002.

[6] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)*, pages 439–448, June 2000.

[7] M. Das, S. Lerner, and M. Seigle. Esp: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*, pages 57–68, June 2002.

[8] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI '00)*, Sept. 2000.

[9] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd Annual Symposium on Principles of Programming Languages (POPL '05)*, pages 110–121, Jan. 2005.

[10] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*, pages 234–245, June 2002.

[11] D. Geels, G. Altekarz, P. Maniatis, T. Roscoey, and I. Stoicaz. Friday: Global comprehension for distributed replay. In *Proceedings of the Fourth Symposium on Networked Systems Design and Implementation (NSDI '07)*, Apr. 2007.

[12] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th Annual Symposium on Principles of Programming Languages (POPL '97)*, pages 174–186, Jan. 1997.

[13] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '05)*, pages 213–223, June 2005.

[14] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP '89)*, pages 202–210, Dec. 1989.

[15] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 193–208, Dec. 2008.

[16] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '92)*, pages 125–138, Dec. 1992.

[17] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5): 279–295, 1997.

[18] G. J. Holzmann. From code to models. In *Proceedings of the Second International Conference on Applications of Concurrency to System Design (ACSD '01)*, June 2001.

[19] C. Killian, J. W. Anderson, R. Jhala, , and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the Fourth Symposium on Networked Systems Design and Implementation (NSDI '07)*, pages 243–256, April 2007.

[20] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI '07)*, pages 179–188, June 2007.

[21] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed Java applications. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing (IPDPS '00)*, pages 219–228, May 2000.

[22] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[23] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, 1978.

[24] W. Lin, M. Yang, L. Zhang, and L. Zhou. Pacifica: Replication in log-based distributed storage systems. Technical report.

[25] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3s: Debugging deployed distributed systems. In *Proceedings of the Fifth Symposium on Networked Systems Design and Implementation (NSDI '08)*, pages 423–437, Apr. 2008.

[26] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. 1988.

[27] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 75–88, Dec. 2002.

[28] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 267–280, Dec. 2008.

[29] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI '07)*, pages 89–100, June 2007.

[30] Phoenix. `http://research.microsoft.com/Phoenix/`.

[31] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, pages 263–272, Sept. 2005.

[32] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the USENIX Annual Technical Conference (USENIX '04)*, pages 29–44, June 2004.

[33] The Coverity Software Analysis Toolset. `http://coverity.com`.

[34] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[35] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, , and R. Thakur. Formal verification of practical mpi programs. In *Principles and Practices of Parallel Programming (PPoPP)*, pages 261–260, Feb. 2009.

[36] Windows API. `http://msdn.microsoft.com/`.

[37] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the Sixth Symposium on Networked Systems Design and Implementation (NSDI '09)*, Apr. 2009.

[38] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 273–288, Dec. 2004.

[39] J. Yang, C. Sar, and D. Engler. Explode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 131–146, Nov. 2006.