To appear in Middleware'09, http://middleware2009.cs.uiuc.edu/

DSF: A Common Platform For Distributed Systems Research and Development

Chunqiang Tang

IBM T.J. Watson Research Center
ctang@us.ibm.com

Abstract. This paper presents Distributed Systems Foundation (DSF), a common platform for distributed systems research and development. It can run a distributed algorithm written in Java under multiple execution modes—simulation, massive multi-tenancy, and real deployment. DSF provides a set of novel features to facilitate testing and debugging, including chaotic timing test and time travel debugging with mutable replay. Unlike existing research prototypes that offer advanced debugging features by hacking programming tools, DSF is written entirely in Java, without modifications to any external tools such as JVM, Java runtime library, compiler, linker, system library, OS, or hypervisor. This simplicity stems from our goal of making DSF not only a research prototype but more importantly a production tool. Experiments show that DSF is efficient and easy to use. DSF's massive multi-tenancy mode can run 4,000 OS-level threads in a single JVM to concurrently execute (as opposed to simulate) 1,000 DHT nodes in real-time.

Keywords: distributed systems, simulation, debugging, mutable replay, massive multi-tenancy, chaotic timing test.

1 Introduction

Nowadays, almost every application becomes distributed for one reason or another. Their functions are diverse and their working environments are heterogeneous, ranging from small embedded devices for home healthcare monitoring, to large mainframe servers for extremely reliable transaction processing. Despite their prevalence, it still remains challenging to build robust and high-performance distributed systems, simply because of their very nature: concurrent and asynchronous execution in potentially volatile and failure-prone environments. For example, the "simple" Paxos algorithm was invented in 1990, while its robust implementation remained a challenging problem that was worth publishing in a top research conference in 2007 [1].

In the past, several complimentary methods have been proposed to facilitate the development of distributed systems:

- **Simulation** [6, 10, 12]**:** Provide a framework that can execute the same code of a distributed algorithm in both simulation and real deployment. The simulation mode eases the tasks of testing and debugging.
- **Deterministic replay** [3, 7, 8, 15, 17]**:** Log application activities during normal execution and, if a bug shows up, replay the execution flow to precisely repeat the bug. This helps capture elusive, unreproducible bugs.
- **Fault injection** [16]**:** Provide a framework that automatically exercises a distributed application with various failure scenarios, which helps trigger bugs that do not show up under normal operations.

- **Model checking** [11, 13, 20]**:** Write assertions that an application's distributed states must satisfy, and then use a runtime or offline tool to discover violations of the assertions. This helps find a bug soon after its inception.
- **Code generation** [14]**:** Describe a distributed algorithm in a high-level specification language, and then use a tool to translate the specification into a real implementation. This method may help reduce both the development effort and the chance of introducing bugs.

These ideas have been well known for some time, and each of them provides certain benefits in lowering the difficulty of developing distributed systems. However, these ideas mostly still stay in research labs and their wide adoption in mainstream software development (either open-source or proprietary) is yet to be reported. One main reason of their limited adoption is that they often deviate from the mainstream programming environments and rely on customized programming tools that are not familiar to layman programmers, e.g., an "extended" programming language or a hacked hypervisor, OS, runtime, compiler, linker, etc. Moreover, customized tools also limit the lifetime of a research prototype, due to the lack of long-term support and inability to keep up with the evolution of mainstream programming tools.

1.1 Distributed Systems Foundation (DSF)

The authors develop both research prototypes and commercial software products at IBM. Like many past efforts, we build our own framework, called *Distributed Systems Foundation (DSF)*, to ease the task of developing distributed systems. For practical reasons, the design of DSF strictly follows one rule: *no hacking programming tools.* DSF is written entirely in Java, without hacking any external tools such as JVM, Java runtime library, compiler, linker, system library, OS, or hypervisor. We strongly believe that this simplicity is key to ensuring that DSF has a long life as programming tools evolve, and this simplicity is also crucial to the success of DSF not only as a research prototype but more importantly as a production tool.

Like previous work, DSF provides some well-known features to facilitate testing and debugging, including simulation, deterministic replay, fault injection, and model checking. In addition, DSF offers several novel features not available in previous work:

- **Mutable replay:** After observing an elusive bug, the most popular debugging technique perhaps is to add code to do detailed logging for activities related to the bug, re-compile the program, and then run it again, hoping that the bug will show up again but with detailed information logged this time. All existing deterministic replay methods [3, 7, 8, 15, 17], however, do not support this popular debugging practice, because they cannot replay a modified program even if the modification has no side effects on the application logic, e.g., simple read-only statements for logging or assertion. Moreover, even if the executable of the program does not change, those methods cannot replay the program with a changed configuration, e.g., changing the logging level of *log4j* from "INFO" to "DEBUG" in order to log detailed debug information. By contrast, DSF supports *mutable replay.* For the example above, DSF guarantees that the bug precisely repeats itself in the replay run as in the original run, while the added (or newly enabled) debugging code logs more information to help pinpoint the root cause of the bug. Moreover, after the code is changed to fix the bug (which is almost certain to have side effects on the application logic), DSF can deterministically replay the original execution flow until right before the new code, and then start to execute the new code and test whether the bug still shows up.

- **Chaotic timing test:** Because of the concurrent and asynchronous nature of distributed systems, many elusive, unreproducible bugs are caused by unexpected interaction sequences between distributed components or unexpected timing of these events. In addition to fault injection [16], DSF introduces randomized, chaotic timing to all event executions in order to systematically exercise a distributed system under diverse timing, by drastically varying delays in thread scheduling, timer wake-up, message propagation, and message processing. Because of this feature, DSF's simulation mode overcomes many limitations of existing systems' simulation modes [10, 12], and (in terms of finding bugs) is actually as powerful as a combination of their real deployment mode and simulation mode (see Section 6).
- **Massive multi-tenancy:** In addition to the simulation mode and the real deployment mode, DSF also provides the *massive multi-tenancy mode* that is not available in previous work. With a careful design and implementation, this extremely efficient mode can run thousands of OS-level threads inside a single JVM (i.e., one OS process) to concurrently execute (as opposed to simulate) thousands of instances of a distributed algorithm (e.g., thousands of DHT nodes) in real-time. This not only simplifies testing and debugging, but also makes elusive race condition bugs and subtle performance bugs more evident, due to the severe contention among thousands of threads.

Setting up large-scale distributed testing on many servers is hard, due to both hardware resource constraints and human resource constraints (time and efforts). Moreover, chasing an elusive bug in such a setting can be time consuming, because pieces of the program states related to the bug may be scattered on different servers. One focus in DSF is to test and debug a large-scale setup of a distributed algorithm inside *a single JVM*. The massive multi-tenancy mode and the simulation mode with chaotic timing test are motivated by this need, which allow DSF to trigger most bugs (even those elusive race condition bugs) in a single JVM. Moreover, with all the states of a distributed algorithm readily available in one JVM, these two execution modes allow the user to easily write model checking code to catch violations of global invariants, or to use an interactive debugger to inspect all data structures related to a bug, even if they logically belong to different "distributed" components.

1.2 Contributions

As veteran developers of both research and commercial distributed systems, we constantly feel the pain of low productivity, and DSF grew out of our own needs. DSF takes a pragmatic approach while offering many advanced features. Specifically, we make the following contributions in this paper:

- **Simplicity:** Unlike previous work, DSF offers many advanced features (including simulation, deterministic replay, fault injection, and model checking) without hacking any programming tools. We believe that this simplicity is crucial to the success of DSF not only as a research prototype but more importantly as a production tool.

- **Novel features:** DSF provides several novel features not available in previous work, including mutable replay, chaotic timing test, and massive multi-tenancy. These features help significantly improve development productivity.
- **Implementation:** We build a solid implementation of DSF, demonstrating that our ideas are not only feasible but can also be implemented efficiently. For example, DSF's massive multi-tenancy mode is capable of running 4,000 OS-level threads in a single JVM to concurrently execute 1,000 DHT nodes in real-time.

2 Overview of DSF

This section presents an overview of DSF. We first describe how DSF's API virtualization approach helps improve application portability. We then discuss the challenges in testing and debugging, and how DSF's different execution modes (simulation, massive multi-tenancy, and real deployment) help address these challenges.

2.1 Portability through API Virtualization

Since DSF is written in Java, portability might seem trivial but it is actually not. DSF's implementations of distributed algorithms are intended for broad code reuse across many applications, from small embedded systems to large mainframe servers. Even for tasks as simple as sending a network message, the network APIs vary in different environments.

For servers that can run Java 2 Standard Edition (J2SE), it is natural to implement network communication using the high-performance non-blocking *java.nio* package. For embedded systems that can only run Java 2 Micro Edition (J2ME), *java.nio* is not available and network communication has to use the less efficient *java.net* package. This was exactly the problem faced by people who tried to port FreePastry to J2ME (see https://mailman.rice.edu/pipermail/freepastry-discussion-l/2005-April/000030.html).

Moreover, sophisticated commercial products and open source projects may even mandate the use of certain powerful network packages so that all components handle issues such as security and firewall tunneling in a uniform manner. Examples include Apache FtpServer built atop the Apache MINA network framework, and IBM WebSphere [9] built atop the WebSphere Channel Framework.

Our solution to the portability problem is API virtualization. Figure 1 shows the architecture of DSF, where the DSF APIs provide a programming environment that isolates platform-dependent details. For example, distributed algorithms simply use *TCP.send(Message)* for network communication, and then can run on different platforms, by plugging in different implementations of *TCP.send()*.

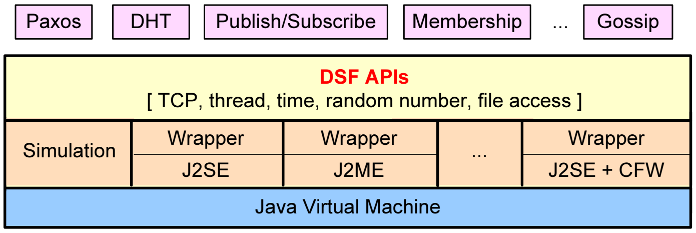

Fig. 1. Architecture of DSF. The DSF APIs are shown in Figure 3.

2.2 Challenges in Testing and Debugging

Distributed systems are notoriously hard to test and debug, which often consume the biggest fraction of the development time, due to the very nature of distributed systems.

- **Hard to set up large-scale testing:** In most organizations, a developer has only a handful of development machines, and can do large-scale testing only for limited time, by carefully coordinating machine usage with many other developers. This inconvenience results in not only low productivity for developers, but also insufficient testing of code.
- **Unexpected failures:** During design and implementation, it is hard to fully anticipate all possible failure scenarios of distributed components, while a single unexpected failure can move the system into an incorrect state.
- **Unexpected event timing:** During design and implementation, it is hard to fully anticipate all possible interaction sequences and event timing among distributed components, while just one delayed thread activity or one message arriving at an unexpected time can move the system into an incorrect state.
- **Hard-to-repeat bugs:** Sometimes, automated testing may take days or even weeks to experience a rare race condition that triggers a bug. Because the race condition is unexpected, it is not uncommon that the log does not contain sufficient information to help pinpoint the root cause of the bug. After the developer adds more debugging code and then restarts the test, it may take a long time for the bug to show up again, or the bug may simply become dormant due to changes of non-deterministic factors in the execution environment.
- **Lack of a global view for checking global consistency:** Often, testing and debugging involves comparing the internal states of distributed components to catch inconsistency. For example, in an overlay network, if node X considers node Y as its neighbor, after some reasonable delay, Y should also consider X as its neighbor. Otherwise, it is a bug of inconsistency. Automated global consistency checking is difficult for a large-scale distributed system, because the states of its components scatter on different servers.

In addition to improving portability, DSF's API virtualization approach (see Figure 1) provides the opportunity of building a powerful testing and debugging framework to address the challenges described above. A distributed algorithm implemented on top of the DSF APIs can run in different execution modes (simulation, massive multi-tenancy, and real deployment), each of which offers certain advantages in testing and debugging. The DSF APIs are minimal and straightforward. They are listed in Figure 3 and will be discussed in Section 3. Below, we start our discussion with the simulation mode.

2.3 Simulation Mode

The simulation mode allows a developer to write the code of a distributed algorithm on a laptop, and then locally simulate its behavior of running on thousands of servers. During simulation, DSF can conduct large-scale failure tests by continuously failing servers and adding new servers. In the simulation mode, the states of all distributed components (e.g., thousands of DHT nodes) are available in a single JVM, which allows the developer to easily write a debug subroutine to check the global consistency of all nodes. Instead of defining a new scripting language (e.g., as in [12]) for checking consistency, we opt for having the developer write the checking subroutine in Java, because it is simple and flexible, and the developer perhaps has been familiar with doing that for local components since the very beginning of programming.

The timing of every event in DSF (e.g., thread delay, network delay, server failure time, and timer wake-up) is randomized, in order to fully exercise the code under diverse timing. The randomization, however, is controlled by a pseudo random number generator and hence the events are precisely repeatable if the generator is initialized with the same seed. After the developer encounters a bug, she can re-run the test in an interactive debugger and step through the code. The bug is repeatable, because all events are deterministically repeatable.

If the bug is triggered by a rare race condition, the simulation may take days or even weeks before the bug shows up. After observing the bug, what the developer wants is time travel back to, e.g., just five minutes before the bug happens, and then start to debug from there. DSF supports this by periodically checkpointing the simulation state into a file. After a bug happens, the developer can add debugging code to log more information, re-compile the program, and then instantly resume the simulation from a recent checkpoint. This saves weeks of time to re-run the simulation from the very beginning.

If the bug is in a colleague's code, the developer can send the checkpoint file to the colleague, who then can deterministically replay and reproduce the bug, even if the checkpoint was taken on one platform (e.g., Windows) while the colleague's re-run is on another platform (e.g., Linux). The checkpoint file contains the execution environment that triggers the bug. No additional efforts are needed to replicate the environment, which sometimes is hard to do.

DSF's implementation of checkpoint is a pure Java solution. It saves in a file the serialized representations of objects in the JVM. By contrast, traditional OS-level solutions checkpoint the process image of JVM, while traditional hypervisor-level solutions checkpoint the image of the entire OS. They are less flexible as they do not support the most popular practice of debugging—adding debugging code to check conditions or log more information. A replay run from an OS image or a JVM process image always executes exactly the same old code, and does not allow changing the Java program's source code or configurations such as logging level.

If the program reads or writes certain files, DSF saves those files' contents and states (e.g., a random access file's current file pointer position) along with the checkpoint, so that a later run resumed from the checkpoint sees exactly the same file contents and states as the original run does, even if those files have been modified since the checkpoint time.

2.4 Massive Multi-tenancy Mode

The simulation mode is helpful in finding and fixing bugs but it cannot discover all bugs. This is because the simulated implementation of the DSF APIs differs from the real implementation of the DSF APIs that is used in production environments. For example, because the two implementations handle DSF's thread pool APIs differently, the simulation mode may not discover certain synchronization bugs.

We address this problem by providing the massive multi-tenancy mode. It uses exactly the same implementation of the DSF APIs as that in the real deployment mode.

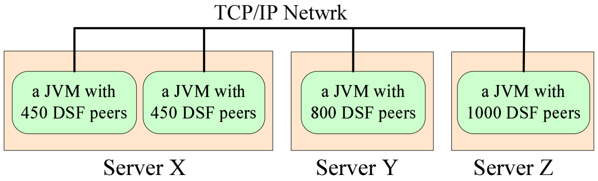

Fig. 2. Distributed multi-tenancy mode.

Inside one JVM, it creates thousands of OS-level threads to do real-time execution (as opposed to simulation) of thousands of distributed components, e.g., thousands of DHT nodes. Each DHT node has its own thread pool and listens on a different TCP port. Although the nodes are in the same JVM, they still communicate through real TCP connections and the traffic goes through the OS kernel, because the multi-tenancy mode uses the real implementation of the DSF APIs.

The massive multi-tenancy mode allows the developer to easily conduct large-scale testing of thousands of nodes on one powerful server. Because the states of all nodes are in one JVM, the multi-tenancy mode and the simulation mode can use the same debug subroutine to check the global consistency of all nodes. Moreover, synchronization bugs, timing bugs, and performance bugs all become more evident in the massive multi-tenancy mode, as thousands of threads compete for CPUs and their execution orders change constantly.

The scalability of the multi-tenancy mode is limited by the physical resources of one server. The distributed multi-tenancy mode in Figure 2 address this problem by connecting multiple servers running the multi-tenancy mode into one large distributed system. This mode differs from distributed simulation, because it uses the real implementation of the DSF APIs. In Figure 2, server *X* runs two JVMs instead of one JVM because sometimes one JVM may not be able to fully utilize the resources of a powerful server, due to JVM's internal bottlenecks.

2.5 Real Deployment Mode

Once the implementation of a distributed algorithm passes the rigorous tests in DSF and reaches a stable stage, it can be packaged into a library (i.e., one JAR file) together with an appropriate real implementation of the DSF APIs, and then widely reused by many different distributed applications. Those applications can simply use the high-level APIs provided by the distributed algorithm (e.g., the DHT APIs for routing and storage), and do not even need to know the existence of DSF or its APIs. That is, a distributed application as a whole need not conform to the DSF APIs.

3 The DSF APIs

This section presents the DSF APIs, which provide accesses to TCP, thread, time, random number, and file in a platform-independent manner. Distributed algorithms implemented on top of the DSF APIs are portable and can run in multiple execution modes, each of which offers certain advantages in testing and debugging. We use the term, *DSF runtime*, to refer to an implementation of the DSF APIs.

In Figure 3, the *Peer* class represents the execution environment seen by the distributed algorithm. This section uses a DHT algorithm as an example, for which

```
class Peer {
    Peer (Config config);                       // Configure the DSF runtime.
    void start ();                              // Boot the DSF runtime.
    void stop ();                               // Emulate a failure.
    Endpoint getLocalEndpoint();                // IP and listening port.
    TCP tcpConnect (Endpoint server);           // Outgoing TCP.
    void submitJob (Runnable job);              // Submit to thread pool.
    void submitFifoJob (String fifoJobQueue, Runnable job);
    TimerHandle submitTimer (long delay, Timer timer);  //Timer fires after ''delay'' ms.
    long localTime ();                          // Like System.currentTimeMillis()
    static Random random ();                    // Deterministic in simulation.
    void registerService (String name, Object service);
    boolean deregisterService (String name, Object service);
    Object lookupService (String name);
    RandomAccessFileIfc getRandomAccessFile (String file, String mode);
}

class TCP {
    void send (Message msg);
    void close ();
    boolean registerTCPClosedCallback (TCPClosedCallback callback);
    boolean deregisterTCPClosedCallback (TCPClosedCallback callback);
}

class Message implements java.io.Serializable {
    Message (String fifoMsgQueue);
    void procMessage (Peer peer, TCP tcp);
}
```

Fig. 3. Summary of the DSF APIs.

one *Peer* object represents one DHT node. The program can create multiple *Peer* objects in one JVM in order to run multiple DHT nodes.

Node Start and Stop. Suppose the *DHTImpl* class contains the implementation of the DHT protocol. To start a new DHT node, the code creates a *Peer* object and a *DHTImpl* object, invokes *Peer.registerService("DHT", dhtImpl)* to register the *DHTImpl* object, and finally invokes *Peer.start()* to boot the DSF runtime. The code of *DHTImpl* invokes the methods of the *Peer* object to interact with the DSF runtime, e.g., sending messages or executing timers to do periodical DHT maintenance. DSF supports code componentization. Another module can invoke *Peer.lookupService("DHT")* to discover the registered DHT service, without being hard-wired with any particular DHT implementation. The class *Config* used in the constructor *Peer(Config)* specifies configurations such as the node's TCP listening port. The node's local IP address and TCP listening port can be obtained from *Peer.getLocalEndpoint()*, where *Endpoint* is a more efficient representation of *java.net.InetSocketAddress*. To emulate the failure of a DHT node, the testing code can invoke *Peer.stop()* to terminate the node.

TCP and Message. An outgoing TCP connection is created by invoking *Peer.tcpConnect(Endpoint server)*, which returns a *TCP* object. *TCP.send(Message)* sends an outgoing message. When the message arrives at the destination, *Message.procMessage(Peer, TCP)* is automatically invoked by the DSF runtime, where the argument *Peer* is the destination's execution environment, and the argument *TCP* is the connection over which the message arrived. Inside *Message.procMessage()*, the code can invoke *Peer.lookupService("DHT")* to retrieve the registered *DHTImpl* object to assist processing. The code can also use *TCP.registerTCPClosedCallback()* to register a callback, which will be invoked by the DSF runtime when the TCP connection breaks. In DSF, there is no *TCP.receive()*, because message processing is automatically invoked by the DSF runtime when a message arrives.

DSF is designed for developing high-performance implementations of distributed algorithms that can be directly used in production systems. Because massive multi-core processors will become prevalent in the near future, DSF strives to minimize synchronization and maximize concurrency. Following this principle, unless specifically required, the DSF runtime does not promise that it will invoke *Message.procMessage()* in sequential order for messages coming from the same TCP connection, because that would preclude concurrent message processing. For example, in a DHT implementation, a node X may send to another node Y multiple DHT lookup messages through one TCP connection. Unless there are specific semantic restrictions, it would be more efficient for Y to concurrently process these DHT lookup messages on multiple processors, instead of processing them in sequential order on one processor.

The argument of the constructor *Message(String fifoMsgQueue)* controls the order of message processing. Only messages from the same TCP connection and tagged with the same *fifoMsgQueue* are processed in sequential order. In addition to the message ordering specified by *fifoMsgQueue*, the "user code" can freely employ any synchronization mechanisms inside *Message.procMessage()* to protect critical regions. (*Note that, in the rest of this paper, we use the term, "user code", to refer to the code of a distributed algorithm written on top of the DSF APIs.*)

Currently, DSF provides no APIs for UDP communication. They can be added easily, but we explicitly discourage the use of UDP in production systems, because of the complications in security and firewall tunneling. This is a hard lesson we learned from our experience of productizing our research prototypes into WebSphere [9].

Timer and Thread Related Jobs. *Peer.submitTimer(long delay, Timer job)* submits a timer to be executed by the DSF runtime after "*delay*" milliseconds. DSF makes best efforts but does not guarantee the accuracy of the timer delay. Expired timers can run concurrently or in any order. DSF does not guarantee that a timer with expiration time x always executes before a timer with expiration time x+1.

Peer.submitJob(Runnable job) submits a job to be executed by the DSF runtime "immediately", which is semantically equivalent to *(new Thread(job)).start()*. Submitted jobs can be executed concurrently or in any order. By contrast, *Peer.submitFifoJob(String fifoJobQueue, Runnable job)* provides the ordering guarantee. Jobs tagged with the same *fifoJobQueue* are executed in sequential order.

To maximize concurrency and to avoid potential deadlocks, the DSF runtime always invokes the user callback code (e.g., timers, jobs, and *Message.procMessage()*) without holding any locks. It is the developer's responsibility to implement proper synchronization, e.g., by using Java's *synchronized* language construct.

If the developer wants to run a distributed algorithm in the simulation mode and benefit from its testing and debugging capabilities, then the user code is not allowed to directly create its own threads or put threads into sleep by invoking *Thread.sleep(long delay)* or *Object.wait()*. Instead, it should use DSF's jobs or timers to implement the same function. Otherwise, the simulation mode cannot provide the feature of "precise replay of buggy runs," because the execution order of threads not under DSF's control is non-deterministic.

This restriction only applies to the part of the code that the developer wants to run in the simulation mode. If the developer merely uses libraries (e.g., DHT) developed in DSF to build an application and has no intention to run the entire application in the simulation mode, then the application need not follow this restriction and can use threads freely. See the related discussion in Section 2.5.

File Access. In the simulation mode, the DSF runtime periodically serializes and checkpoints the entire state of the program in order to support time travel debugging. Files accessed by the program are part of the state that should be saved in the checkpoint. However, Java's file utilities (e.g., *java.io.RandomAccessFile*) are not serializable and hence cannot be checkpointed. To work around this problem, the program should access files through objects returned by *Peer.getRandomAccessFile(String file, String mode)*. Those objects are serializable and provide functions identical to *java.io.RandomAccessFile*. The DSF APIs also provide the equivalences of other file utilities, including *java.io.FileReader*, *java.io.FileWriter*, *java.io.FileInputStream*, and *java.io.FileOutputStream*.

Random Number. Random numbers are widely used in distributed algorithms. For example, introducing randomness into timer delays helps avoid pathological synchronized behaviors of distributed nodes. To provide the feature of "precise replay of buggy runs," all random numbers used by the code must be pseudo random but actually deterministic in the simulation mode. One simple way to achieve this is to replace the statement "*new Random()*" with "*new Random(Peer.random().nextInt())*", i.e., using a controlled pseudo random seed to initialize the random number generator. In the real deployment mode, *Peer.random()* is truly random. In the simulation mode, *Peer.random()* is deterministically controlled by *Config.randomSeed*.

System Time. To run properly in the simulation mode, the code should use *Peer.localTime()* to read system time. In the real deployment mode, *Peer.localTime()* and *System.currentTimeMillis()* are identical. In the simulation mode, *Peer.localTime()* returns the time in the simulated world.

Optional Testing Framework. Using the DSF APIs in Figure 3, the developer can implement a distributed algorithm as well as its testing environment. Optionally, the developer can also use DSF's built-in testing framework to save efforts. The developer only needs to write a global consistency checking subroutine, which takes as inputs a set of *Peer* objects and reports whether their internal states are consistent. Controlled by a configuration file, DSF starts a certain number of *Peers* with the user's algorithm code registered as a plug-in service of each *Peer*. DSF automatically alternates between churn periods and stable periods. During a churn period, DSF randomly fails existing *Peers* and starts new *Peers*. In the simulation mode and the massive multi-tenancy mode, DSF periodically invokes the user's consistency checking subroutine to detect state inconsistency among *Peers*.

4 Implementations of the DSF APIs

This section describes the simulated and real implementations of the DSF APIs.

4.1 Simulated Implementation of the DSF APIs

In the simulation mode, DSF simulates TCP, thread, and time, but provides real file access. The DSF APIs interact with a discrete-event simulation engine that uses one thread to execute events ordered in time. Timers and jobs submitted by *Peer.submitTimer()*, *Peer.submitJob()*, and *Peer.submitFifoJob()* are treated as future events to be executed. DSF also generates events internally, e.g., for the arrival of TCP messages.

Simulated TCP. DSF simulates the high-level semantics of the TCP protocol, including connection establishment, in-order and reliable data transfer, and connection termination. Currently, DSF does not simulate low-level details such as TCP congestion control, but can simulate volatile network delay and varying available bandwidth. A developer can either use DSF's built-in network model, or link her own network model with DSF through a well defined interface.

DSF simulates a TCP port manager that recycles ports as *Peers* start and stop. It allows the use of much more than 65,536 ports in order to simulate a large system. DSF can be configured to pass messages between *Peers* using either *serialization* or *cloning*. Serialization allows DSF to accurately measure network traffic, but is about 7 times slower than cloning.

Peer.stop() simulates a fail-stop failure, which can be either *apparent* or *silent*. To illustrate the difference, consider a distributed system with two processes X and Y, and a TCP connection between them. The failure of X is apparent to Y, if Y's read operation from the TCP connection returns immediately with an error. For example, if process X crashes but the OS that hosts X still functions, then the OS will close all X's TCP connections and Y will immediately notice the TCP read error. The failure of X is silent to Y, if Y's TCP read operation returns no errors. This happens, for example, if the machine that hosts X suddenly lost power, or if X encounters an uncaught exception and hangs. Silent failures are usually more subtle to handle and more difficult to test in deployed systems. Simulating silent failure helps test whether the user code handles them properly through mechanisms such as heartbeat timeout.

Chaotic timing test. To discover race condition bugs that depend on event timing, DSF purposely introduces randomized delays into simulated thread scheduling, timers, message propagation, and message processing. Recall that *Peer.submitJob(job)* is semantically equivalent to *(new Thread(job)).start()*. To simulate the delay before the thread is scheduled to run on a CPU and the time it takes to finish the job, DSF adds a randomized delay to the simulated event that represents the execution of this job. As a result, it is possible that a job submitted later actually gets executed before a job submitted earlier. DSF's built-in thread delay model uses a long tailed distribution, and a user can also link her own thread delay model with DSF through a well defined interface. Randomized delays are added to all other types of events as well. DSF's randomized event timing respects event causality as well as event ordering required by semantics, e.g., message processing order mandated by *fifoMsgQueue*.

Checkpoint and rollback. During simulation, DSF periodically checkpoints the states of all distributed components in order to support time travel debugging. We do not use traditional OS or hypervisor level checkpoint methods [3, 17], because they rely on customized programming tools and do not support the most popular practice of debugging—adding debugging code to check assertions or log more information. Unlike DSF's mutable replay method, the OS or hypervisor level methods cannot resume from a checkpoint to run a modified version of the program that has added or newly enabled debugging code, even if the modification has no side effects on the application logic.

DSF implements checkpointing by serializing Java objects in the program and saving them in a file. A checkpoint is always taken between the executions of two events, so that we need not worry about local variables in the user code. DSF does not serialize all objects in the JVM, e.g., the objects that represent loaded Java classes. Instead, DSF only serializes the *Peer* objects and the object that represents the simulation engine. Due to the deep-copy semantics of Java serialization, all other objects recursively reachable through those objects are also serialized. For the DHT example in Section 3, the *DHTImpl* object that implements the DHT protocol is serialized along with the *Peer* object, because the *Peer* object adds a reference to the *DHTImpl* object when *Peer.registerService("DHT", dhtImpl)* was invoked in the initialization phase.

DSF also checkpoints files accessed by the user code. Because Java's file utilities are not serializable, the DSF APIs provide our own serializable file utilities. A file "/dirX/dirY/fileZ" accessed through our utilities is saved on the real file system as file "/tmp/dsf/*$peer_id*/dirX/dirY/fileZ", where *$peer_id* differentiates peers. During a checkpointing operation, DSF flushes all file buffers to ensure that the files on disk are up to date. It then makes a copy of the entire directory "/tmp/dsf" and saves it along with the checkpoint file. DSF also saves in the checkpoint file the states of the files accessed by the user code, e.g., a random access file's current file pointer position.

Suppose the developer encounters a bug, adds some debugging code, re-compiles the program, and resumes the execution from a checkpoint. During the checkpoint recovery, DSF restores the *Peer* objects and the simulation engine object. It copies the saved files back to the directory "/tmp/dsf", re-opens those files, and sets the file pointers to the exact positions before the checkpoint. DSF then continues to execute the next event in the recovered event queue, guaranteeing the bug precisely repeats itself in the resumed run as in the original run, while the added debugging code logs more information to help pinpoint the root cause of the bug. After the developer fixes the bug, she can re-compile the program and resume the execution from the checkpoint again. This time, the resumed run executes the new code and tests whether the bug still shows up. That is, DSF's mutable replay not only helps understand the root cause of the bug, but can also test whether the bug fix actually works.

4.2 Real Implementation of the DSF APIs

This section presents a real implementation of the DSF APIs based on J2SE. In the real implementation, each *Peer* has its own pool of worker threads that execute jobs submitted by *Peer.submitJob()* or *Peer.submitFifoJob()*. Proper synchronization ensures that FIFO jobs are executed in sequential order. The size of the worker thread pool is configurable. Each *Peer* has a dedicated "timer thread" to keep track of all timers submitted by *Peer.submitTimer()*. Inside a loop, this thread sleeps until the first timer expires, and then transfers the expired timer to the worker thread pool for execution.

DSF uses the non-blocking, high-performance *java.nio* package for network communication. A *Message* is serialized before being sent over the network and then deserialized at the destination. Each *Peer* has a dedicated "network thread" that accepts incoming TCP connections on the *Peer's* TCP listening port. This thread also reads data for all established TCP connections. Once a complete *Message* is read in, a job is submitted to the worker thread pool to execute *Message.procMessage()*. Proper synchronization ensures that the ordering requirements of message processing are enforced. If the "network thread" notices that a TCP connection is broken, it submits a job to invoke the user callback code previously registered through *TCP.registerTCPClosedCallback()*.

For an outgoing message, the thread invoking *TCP.send()* serializes the message and then performs a non-blocking network write operation. Unless the message is large, this write typically can send out the entire message. Otherwise, the "network thread" will be responsible for sending out the rest of the message later when this TCP connection is ready for write again. All operations performed by the "network thread" are non-blocking so that a single thread can handle all network I/O operations.

The implementation of other DSF APIs is straightforward. All threads used by the DSF runtime are created inside the constructor *Peer(Config)*, and activated to run by *Peer.start()*. *Peer.localTime()* is mapped to *System.currentTimeMillis()*. *RandomAccessFileIfc* is mapped to *java.io.RandomAccessFile*. *Peer.random()* returns a true random number generator.

5 Experiments and Experience

This section evaluates the efficiency, scalability, and usability of DSF, and reports our experience of using DSF to find bugs.

5.1 Scalability of the Multi-tenancy Mode

We first demonstrate that both the real implementation and the simulated implementation of the DSF APIs are scalable. All experiments are conducted on an IBM System x3850 server with 16GB memory and four dual-core 3GHz Intel Xeon processors, running Linux 2.6.9. To demonstrate that DSF does not make any changes to the programming tools, we use both IBM JDK 1.5.0 and Sun JDK 1.6.0 in our experiments.

We have implemented multiple distributed algorithms in DSF. This experiment uses a DHT implementation called *BlueDHT*, which adopts the ring topology as that in Chord [18], but uses our own algorithms for maintenance and routing. BlueDHT uses a hard-state, rate-limited, reactive maintenance protocol, as opposed to Chord's soft-state, periodical recovery protocol. BlueDHT has low maintenance overhead and is robust under churn, but its implementation is challenging because of its hard-state protocol. The testing and debugging features of DSF provided great help in developing BlueDHT. The details of BlueDHT are beyond the scope of this paper. Below, we use BlueDHT to evaluate the scalability of DSF.

Figure 4 shows the CPU utilization of the x3850 server during one run of BlueDHT in the massive multi-tenancy mode. This experiment runs 4,000 OS-level threads in a single IBM JVM to concurrently execute (as opposed to simulate) 1,000 DHT nodes in real-time. The lifetime of a node is only about 120 seconds (with randomization to avoid synchronized behaviors of nodes). After a node dies, a new node is booted

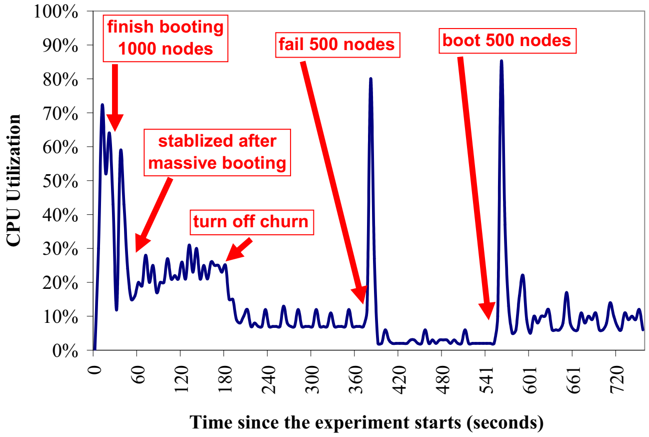

Fig. 4. The massive multi-tenancy mode runs 4,000 threads in a single JVM to concurrently execute (as opposed to simulate) 1,000 DHT nodes in real-time. This experiment was conducted on an IBM System x3850 server with four dual-core 3GHz Intel Xeon processors.

as a replacement. During its lifetime, every node issues a new request roughly every 5 seconds to route a message to the node that is responsible for a random DHT key. The destination node then sends back a confirmation message to the request node.

During the starting phase, a new node is booted roughly every 25 milliseconds. By time 25 seconds, 1,000 nodes have fully booted. The CPU utilization remains high until time 50 seconds, as all nodes are busy with building up their routing tables. DSF's massive multi-tenancy mode is efficient in doing large-scale experiments. With 4,0000 threads running 1,000 DHT nodes in a single JVM, the CPU utilization stays below 30% even under high churn and high traffic, where the node lifetime is only 120 seconds and every node initiates a new DHT lookup every 5 seconds. This excellent performance is due to DSF's scalable runtime as well as BlueDHT's efficient hard-state maintenance protocol.

In addition to automated fault injection, DSF provides an interactive debugging console that allows the developer to manually turn on/off churn and boot/fail nodes. At time 178 seconds, churn is turned off so that nodes do not fail anymore unless triggered manually. At time 376 seconds, a command is issued to fail 500 nodes concurrently, which causes a spike in the CPU utilization. Within 15 seconds from the time the command was issued, BlueDHT fully recovers from the massive failure. With 2,0000 threads running the remaining 500 DHT nodes in a steady state, the CPU utilization is only about 3%. BlueDHT uses a hard-state, rate-limited, reactive protocol. The "hard-state" aspect of BlueDHT (as opposed to Chord's soft-state protocol) is the source of the efficiency in the steady state. The "reactive" aspect of BlueDHT (as opposed to Chord's periodical recovery) is the reason behind the fast recovery. The reactive maintenance operations are "rate limited" so that, even under high churn or massive concurrent failures, the maintenance operations do not generate so much traffic as to cause system collapse.

At time 551 seconds, a command is issued to instantly boot 500 nodes, which causes another spike in the CPU utilization. Within 20 seconds, BlueDHT quickly re-stabilizes from this flash crowd join, and the CPU utilization goes back to a low level. Again, BlueDHT's hard-state, rate-limited, reactive protocol is the reason for the fast stabilization and the low CPU utilization.

This experiment shows that the massive multi-tenancy mode is efficient and scalable even with thousands of threads running inside one JVM. Because the multi-tenancy mode and the real deployment mode use exactly the same real implementation of the DSF APIs, this implies that the real deployment mode also has high performance. This experiment also shows that it is easy to conduct sophisticated large-scale tests in the massive multi-tenancy mode. This 1,000-node experiment (which comprises constant churns, massive concurrent node failures, and flash crowd node joins) takes only about 10 minutes. The developer simply starts one JVM and then issues commands on the debugging console. In the background, the global consistency checking subroutine automatically checks system states and catches bugs. Our experience indicates that the ability to easily test and debug a large-scale setup in a single JVM is the single most important factor that boots development productivity.

5.2 Scalability of the Simulation Mode

The experiment in Figure 5 runs BlueDHT in DSF's simulation mode. The lifetime of DHT nodes is only 120 seconds and every node initiates a new DHT lookup every 5 seconds. Figure 5 reports the "relative simulation time" of BlueDHT of different sizes. Suppose a BlueDHT with n nodes has "relative simulation time" t. It means that DSF takes t seconds wall clock time to simulate all the activities conducted in the simulated world by the n nodes during one second simulated time. The 1,000-node system has t=0.138, which means that the simulation is 1/t=7.2 times faster than execution in the real world. In other words, it takes only about 8 minutes wall clock time to simulate one-hour activities (in the simulated world) of the 1,000-node system. This efficiency improves the developer's productivity by reducing the time spent on testing and debugging. This figure also shows that the simulation mode scales well as the system size increases, which is due to the good scalability of both the DSF runtime and the BlueDHT algorithm.

Figure 6 shows the time needed to create a checkpoint and the size of the checkpoint file. The checkpoint size scales well. As the system size increases from 1,000 nodes to 10,000 nodes, the checkpoint size only increases by a factor of 12. The checkpoint size is small because DSF saves only the algorithm states (e.g., the DHT routing tables) as opposed to the memory image of the JVM process. The time to create a checkpoint is also scalable. As the system size grows from 1,000 to 10,000 nodes, the checkpoint time increases from 1.3 seconds to 16.8 seconds, i.e., a 13 fold increase. This experiment uses the Sun JVM.

5.3 Bugs Found in the Simulation Mode

Next, we report our experience of using DSF to find bugs. Because DSF's simulation mode proactively conducts chaotic event timing tests, it is more powerful than the simulation modes of other tools in terms of triggering bugs (see Section 6 for a comparison). Our experience indicates that, equipped with features such as long-running automated tests, randomized fault injection, chaotic event timing, and global consistency checking, the simulation mode often helps discover more than 95% of the total bugs. The tricky bugs are often related to race conditions caused by unexpected event timing. We give a concrete example below.

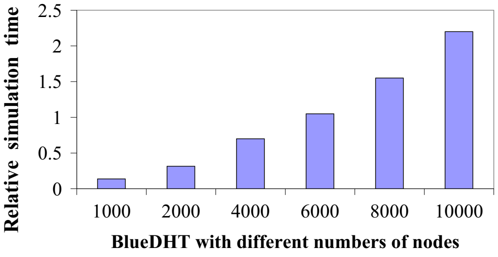

Fig. 5. Relative simulation time of BlueDHT with different numbers of nodes.

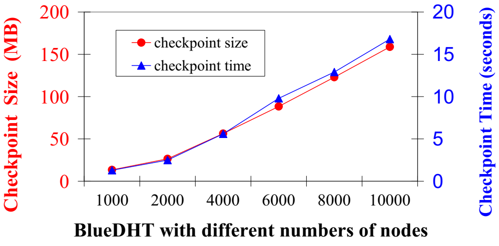

Fig. 6. Time needed to create a checkpoint for BlueDHT and the size of the checkpoint file.

One race condition bug in BlueDHT happens when a node Y reacts to the failure event and the reboot event of another node X out of order. Suppose X and Y are overlay neighbors, and X fails and then immediately reboots with the same IP and TCP listening port. When X fails, Y's DSF runtime detects that the TCP connection to X is broken, and (without holding any locks) invokes Y's callback function *nbrClosed()* previously registered through *TCP.registerTCPClosedCallback()*. Suppose the actual execution of *nbrClosed()* is delayed due to thread scheduling. In the meanwhile, X finishes rebooting and becomes a new neighbor of Y. During the neighbor-establishment process, Y uses the new instance of X to replace the old instance of X in its local data structure, which is a correct behavior.

When Y finally executes *nbrClosed()* after a long delay, it tries to find and remove an existing neighbor with the same IP and port as the failed neighbor X. Y finds the new instance of X, mistakenly considers it as the failed old instance of X, and drops it from the neighbor set without closing the TCP connection to X (because Y considers this connection already closed). The final states are inconsistent: X considers Y as a neighbor, but Y does not considers X as a neighbor.

This bug is a rare race condition in real deployments, because typically X's reboot is slow so that Y processes the failure event of X before processing the reboot event of X. A combination of DSF's testing features helped find the bug. (1) Randomized fault injection triggers one condition of the bug. (2) Randomized network delay triggers one condition of the bug. (3) Randomized thread scheduling delay triggers one condition of the bug. (4) Long-running, automated testing triggers even rare event timing. (5) Finally, global consistency checking automatically catches the bug.

A colleague reported this bug to us and sent us the related checkpoint file. The colleague took the checkpoint on an AIX/Power server, but we were able to resume the execution in our Linux/x86 environment, because the checkpoint file contains serialized Java objects in an platform-independent format. Moreover, there was no need for us to figure out or manually replicate the colleague's setup that triggered the bug. All that information was in the checkpoint file and was taken care of by DSF's replay component automatically. The combined powers of time travel debugging and mutable replay helped us quickly understand the root cause of the bug. We added debugging code, re-compiled the program, and then instantly resumed the execution, without waiting to re-run the simulation from the very beginning. As the bug precisely repeated itself in the resumed run, logs generated by the added debugging code revealed that the bug was triggered by the out-of-order processing of X's failure and rejoin. We then fixed the bug by using epoch numbers to differentiate multiple incarnations of the same node.

5.4 Bugs Found in the Multi-tenancy Mode

The simulation mode randomizes event timing at the event boundary. It cannot trigger race conditions that exist below the event granularity, i.e., inside the subroutine for processing an event. Most of these low-level race condition bugs are caused by incorrect uses of locks. For example, a *ConcurrentModificationException* bug happened in BlueDHT when it processed an incoming message and read a *java.util.collection* data structure without holding a lock.

Synchronization and timing bugs become more evident in the massive multi-tenancy mode, as thousands of threads compete for CPUs and their execution orders change constantly. On the other hand, even under the extreme competition of thousands of threads, it still took about 24 hours of automated testing to trigger the *ConcurrentModificationException* bug described above. This is because the code that accesses the *java.util.collection* data structure without properly holding a lock is very short and hence the probability of concurrent accesses is very low. This indicates that detecting synchronization bugs without the massive multi-tenancy mode would be even more difficult.

Running thousands of threads in one JVM also makes performance bugs (e.g., over synchronization and code inefficiency) more evident, because any performance problems are drastically magnified. For example, an early version of DSF locks the *TCP* object before deserializing an incoming message. This lock unnecessarily prevents other threads of the same *Peer* from using this *TCP* object to send outgoing messages. Because Java deserialization is relatively slow, the locking duration sometimes can be long. DSF has a built-in utility that periodically logs snapshots of all thread stacks. With thousands of threads in one JVM, their blocking patterns easily pop up. We identified this over-synchronization problem and moved the deserialization code outside the lock.

5.5 Bugs Found in Real Deployments

To understand the limitations of testing tools (including DSF), it is interesting to report some bug that slipped through our hands and got into deployed systems. We implemented a distributed performance monitoring algorithm, which collects real-time performance data (e.g., CPU utilization and transaction response time) from a large number of servers to guide resource allocation [9, 19]. This algorithm builds a distributed tree out of an overlay network to collect the data. To ensure the responsiveness of data gathering, a parent node in the tree may bypass a child node to directly collect data from the grandchildren nodes if the child node is temporarily slow, e.g., due to Java garbage collection. Our responsiveness goal is to gather data from most servers in a 1,000-server system within one second most of the time. The algorithm worked well in our environment, but the responsiveness goal was violated from time to time in a deployment environment. By activating DSF's utility that logs the processing time of every event, we first located events that introduced long delays, and then found that the problem was caused by slow DNS lookups—a piece of obsolete debugging code that nobody cared anymore was "accidentally" activated to resolve IP addresses to host names, solely for the purpose of printing user-friendly debugging information. DNS lookups were fast in our development environment but sometimes were slow in the deployment environment. We subsequently disabled all code on the critical path that does DNS lookups. This example shows the difficulty of capturing all bugs in the development environment.

6 Related Work

We first discuss in detail the work that is closest to DSF, and then summarize other related work.

WiDS [12, 13]. The closest work to DSF is WiDS, which provides a set of powerful tools for debugging, e.g., replay-based predicate checking. Like DSF, WiDS also defines a set of APIs, under which different implementations can support simulation and real deployment. DSF and WiDS, however, differ significantly in many aspects. Compared with our three contributions listed in Section 1.2, WiDS (1) is significantly more complicated as it hacks many programming tools, including compiler and linker; (2) does not have DSF's novel features such as mutable replay, chaotic timing test, and massive multi-tenancy; and (3) cannot be used to build high-performance production systems running on multi-core processors, because it uses only a single OS kernel thread to execute all events (i.e., all messages, timers, and user-level thread jobs). The WiDS APIs allows the creation of multiple user-level threads. However, in order to guarantee deterministic replay, all the user-level threads are multiplexed onto a single OS kernel thread.

In its simulation mode, WiDS does not introduce random delays into timers, user-level thread jobs, or message processing. It strictly executes all events in sequential order and always processes an event in "one (simulated) clock tick" [12]. Because "event handling (in the real deployment) can take arbitrarily long," the authors of WiDS [12] noted that "the sequence of events (in the real deployment mode) can differ in unexpected ways (from that in the simulation mode), making it difficult to discover those (race condition) bugs in the simulation environment." Because of this limitation, WiDS Checker [13] has to collect event traces from a system running in the real deployment mode, and then feeds the traces into simulation in order to find race condition bugs.

By contrast, DSF's simulation mode purposely introduces random delays into event timing in order to proactively trigger race condition bugs. One limitation of DSF's simulation mode is that it randomizes event timing at the event boundary, and hence cannot trigger race conditions that exist below the event granularity, i.e., inside the subroutine for processing an event. However, even WiDS' real deployment mode cannot trigger race condition below the event granularity, because it uses a single OS kernel thread to process all events. Therefore, in terms of finding bugs, DSF's simulation mode alone is as powerful as the combination of WiDS' real deployment mode and simulation mode. (This observation is not unique to WiDS. It applies to other previous work [10] as well.) Moreover, all the bugs found by WiDS' real deployment mode can actually be found more easily by DSF's simulation mode, because DSF's chaotic timing tests proactively trigger (as opposed to just passively observe) race condition bugs. Specifically, it would be difficult for WiDS' real deployment mode or simulation mode to trigger the race condition bug described in Section 5.3, because the failure event and the rejoin event are rarely processed out of order.

Other related work. There is an enormous body of work related to development framework for distributed systems, including simulation [6, 10, 12], deterministic replay [3, 7, 8, 15, 17], fault injection [16], model checking [11], and code generation [14]. Below, we only review some representative ones.

Jones and Dunagan [10] developed peer-to-peer systems that use the same code base for simulation and real deployment. Like WiDS, their systems use a single-threaded event-processing model. PlanetSim [6] also supports both simulation and real deployment. These systems, however, do not provide advanced debugging features such as deterministic replay.

Checkpoint and rollback is a well studied topic [5]. Our serialization-based checkpoint method is novel in that it supports mutable replay, i.e., the resumed run can execute a modified program with added debugging code while still getting deterministic replay. ReVirt [3] logs all non-deterministic events in hypervisor to help replay the execution of a guest OS. Flashback [17] modifies the OS kernel to support rollback.

MACEDON [14] allows the user to specify overlay network algorithms in a domain-specific language, and then automatically generates the corresponding C++ code. TLA [11] is a formal specification language for concurrent systems, and the correctness of an algorithm described in TLA can be verified mechanically.

DejaVu [2] modifies JVM to support deterministic replay of multi-threaded Java programs. ConTest [4] uses source code instrumentation to introduce chaotic timing into every shared memory accesses and synchronization operation. Because of its heavy instrumentation, our evaluation shows that ConTest can cause application slowdown by a factor of 100 or more. DSF introduces chaotic timing only at event boundary, which is more efficient but may miss some bugs. ConTest is not designed for testing distributed systems and does not handle bugs caused by concurrent and asynchronous interactions between distributed components.

7 Conclusions

We have presented DSF, a common platform for distributed systems research and development. It can run a distributed algorithm written in Java under multiple execution modes—simulation, massive multi-tenancy, and real deployment. DSF grew out of our own needs in developing research prototypes and commercial software products. It takes a pragmatic approach while offering many advanced features.

Compared with a large body of related work, we made several contributions in this paper. First, we presented a simple yet powerful design that does not hack any programming tools. This simplicity stems from our goal of making DSF not only a research prototype but more importantly a production tool. Second, DSF provides a set of novel testing and debugging features that are not available in previous work, including mutable replay, chaotic timing test, and massive multi-tenancy mode. Finally, we demonstrate through a robust and efficient implementation that our ideas are practical. For example, DSF's massive multi-tenancy mode can run 4,000 OS-level threads in a single JVM to concurrently execute 1,000 DHT nodes in real-time.

The design of DSF takes a pragmatic approach, which naturally has many limitations and leaves room for future improvements. So far, our focus is to provide rich features in the simulation mode and the massive multi-tenancy mode, which allow the developer to easily test and debug a large-scale setup of a distributed algorithm inside a single JVM. This perhaps is the single most important factor that boosts development productivity. Next, we may move on to focus on the real deployment mode. For example, currently DSF's real deployment mode provides no global consistency checking. We may follow the approach in Stardust [20] to store events in a database, and then use SQL to check global consistency.

Acknowledgments

We thank the anonymous reviewers and our shepherd Antony Rowstron for their valuable feedback.

References

1. T. Chandra, R. Griesemer, and J. Redstone. Paxos Made Live—An Engineering Perspective. In *PODC*, 2007.
2. J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, 1998.
3. G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *OSDI*, 2002.
4. O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.
5. E. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.
6. P. García, C. Pairot, R. Mondéjar, J. Pujol, H. Tejedor, and R. Rallo. Planetsim: A new overlay network simulation framework. *Software Engineering and Middleware*, 2005.
7. D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay Debugging for Distributed Applications. In *USENIX*, 2006.
8. Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, F. Kaashoek, and Z. Zhang. R2: An Application-Level Kernel for Record and Replay. In *OSDI*, 2008.
9. IBM WebSphere Extended Deployment.
   http://www-306.ibm.com/software/webservers/appserv/extend/.
10. M. Jones and J. Dunagan. Engineering Realities of Building a Working Peer-to-Peer System. Technical report, MSR Technical Report MSR-TR-2004-54, 2004.
11. L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.
12. S. Lin, A. Pan, Z. Zhang, R. Guo, and Z. Guo. WiDS: an Integrated Toolkit for Distributed System Development. In *HotOS*, 2005.
13. X. Liu, W. Lin, A. Pan, and Z. Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *NSDI*, 2007.
14. A. Rodriguez, C. Killian, S. Bhat, D. Kostic, and A. Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *NSDI*, 2004.
15. Y. Saito. Jockey: a user-space library for record-replay debugging. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, 2005.
16. Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Varton, R. Dancey, A. Robinson, and T. Lin. FIAT–Fault injection based automated testing environment. In *Proc. 18th Int. Symp. Fault-Tolerant Comput*, pages 102–107, 1988.
17. S. M. Srinivasan, S. K, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX*, 2004.
18. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.
19. C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici. A Scalable Application Placement Algorithm for Enterprise Data Centers. In *WWW*, 2007.
20. E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: tracking activity in a distributed storage system. In *SIGMETRICS*, 2006.