DSF: A Common Platform For Distributed Systems Research and Development

Chunqiang (CQ) Tang

IBM Research

December 4, 2009

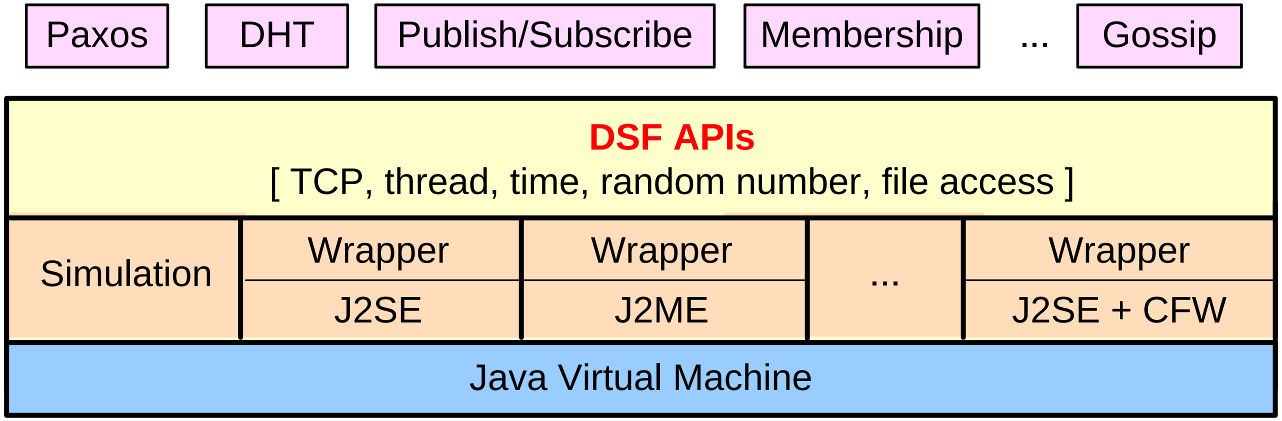
Motivation: a Personal but Common Experience

- I develop production-quality distributed systems at IBM, including
  - Peer-to-Peer Middleware in WebSphere product
  - Cluster performance management in Tivoli product
  - Cloud stuff most recently
- I constantly feel the pain of low productivity due to
  - The difficulty of testing and debugging distributed algorithms
  - The lack of readily-reusable implementations of common distributed algorithms
- After several years of struggling, I finally decided to build a framework called DSF to help myself and hopefully also help others
  - Many people have gone down the same path before, but hopefully I can make a difference this time, for good reasons

Distributed Systems Foundation (DSF)

- DSF is a framework for distributed systems research and development, much like what *ns-2* does for networking research
  - But unlike ns-2, DSF is for building production-quality distributed systems rather than just for simulation
- DSF provides
  - a framework to implement distributed algorithms so that different research results can be compared
  - a set of advanced testing and debugging features to significantly improve development productivity
  - highly-reusable implementations of commonly used distributed algorithms to save repeated development efforts

Overview of DSF



- The DSF APIs provide a programming environment that isolates platform-dependent details
- It improves portability, e.g., different security frameworks can be used without changing the user code
- It also allows a distributed algorithm to run in different execution modes
  - Simulation
  - Real deployment
  - Massive multi-tenancy

Why DSF is Different?

- My goal is to trigger and fix 99% of the bugs (including elusive race condition bugs) while testing all “distributed” components (e.g., 1000 DHT nodes) inside a single JVM
  - My development productivity drops by more than 50% when moving from 1 JVM to just 2 JVMs, not to mention 1000 JVMs
  - It is difficult to chase bugs across servers due to scattered states
- Simulation is widely used, but existing simulation frameworks cannot trigger many bugs that happen in reality
  - From WiDS: “the sequence of events differ in unexpected ways, making it difficult to discover those bugs in the simulation environment”
- DSF provides novel features in simulation to make it much more powerful
  - Chaotic timing test, time travel debugging and mutable replay, fault injection, etc.
- DSF provides the massive multi-tenancy mode
  - Uses thousands of OS kernel threads to actually run thousands of distributed components (e.g., 1000 DHT nodes) in a single JVM

Chaotic Timing Test in Simulation

- Many elusive race condition bugs are caused by unexpected event timing
- It is hard to trigger those bugs even in the real deployment mode
  - They occur rarely but can corrupt everything if they happen
- DSF systematically randomizes all event timings in the simulation mode
  - Server failure, thread scheduling, network delay, message processing, etc.
  - E.g., if the user code says, “run timer job A 5 seconds later; run timer job B 6 seconds later”, DSF sometimes will intentionally run them out of order, just like what may happen in real systems
- How about coverage?
  - DSF does not try to understand the user code in order to generate event sequences that have 100% coverage
  - The hope is that long-running randomized tests will give good coverage

Time Travel Debugging and Mutable Replay in Simulation (1/3)

- You may have this experience
  - Suppose a long-running randomized test takes a whole week to trigger a bug caused by a rare race condition
  - Now you know the bug but you have no sufficient printouts to understand the bug
  - Following the most popular practice, you add more debugging code (e.g., printf and assert), recompile the program, and run it again
  - The bug may show up one week later and this time you have sufficient printouts --- lucky you, despite of the anxiety of one week waiting
  - If you are not lucky, the bug may not even show up in one month
    - uhm…I will just live with it and hope it won't happen in production systems

Time Travel Debugging and Mutable Replay in Simulation (2/3)

- But I hope to offer this new experience
  - Suppose a long-running randomized test takes a whole week to trigger a bug caused by a rare race condition
  - You add more debugging code and recompile the program
  - You time travel back to just 1 minute before the bug happens, but then run the modified program instead of the original program
  - Within 1 minute, the bug precisely repeats itself as in the original run, but the new debugging code prints out everything you want to see
  - You fix the bug in 5 minutes and spend the rest of the week on vacation
- With prior work, deterministic replay is possible, e.g., by using a customized OS or hypervisor, but you cannot add any debugging code
- I want to offer deterministic but mutable replay

Time Travel Debugging and Mutable Replay in Simulation (3/3)

- How it is implemented
  - DSF makes periodical checkpoints, by serializing the objects that represent the distributed algorithm and saving them in a checkpoint file
  - At any time, you may add more debugging code, recompile your program, and then ask DSF to resume the execution from a checkpoint
  - DSF de-serializes objects from the checkpoint to initialize the modified program, and then starts to run it
  - Now the bug precisely repeats itself because all randomized timing tests in DSF are pseudo-random but actually deterministic
  - Files accessed by the user code are also automatically saved in the checkpoint so that the user code sees the same contents in the resumed run
- Unlike prior work, DSF does not checkpoint the JVM process image, or the whole OS image, because that would preclude mutable replay

Massive Multi-tenancy Mode

- Even with fault injection and chaotic timing test, simulation still cannot discover all bugs, because the simulated impl. of the DSF APIs differ from the real one
- The multi-tenancy mode and the real deployment mode use exactly the same implementation of the DSF APIs
- The multi-tenancy mode may use thousands of threads to run thousands of distributed components (e.g., 1000 DHT nodes) in one JVM
  - The user code cannot tell and does not care the difference, i.e., whether the components run on 1000 different servers or in a single JVM
  - All TCP communication still goes through the OS kernel
- The contention of thousands of threads in one JVM also makes race condition bugs and performance bugs more evident
- Since the global states are available in one JVM, the multi-tenancy and simulation modes can use the same Java code for checking global consistency

Massive Multi-tenancy: use 4,000 threads in one JVM to run 1,000 BlueDHT nodes

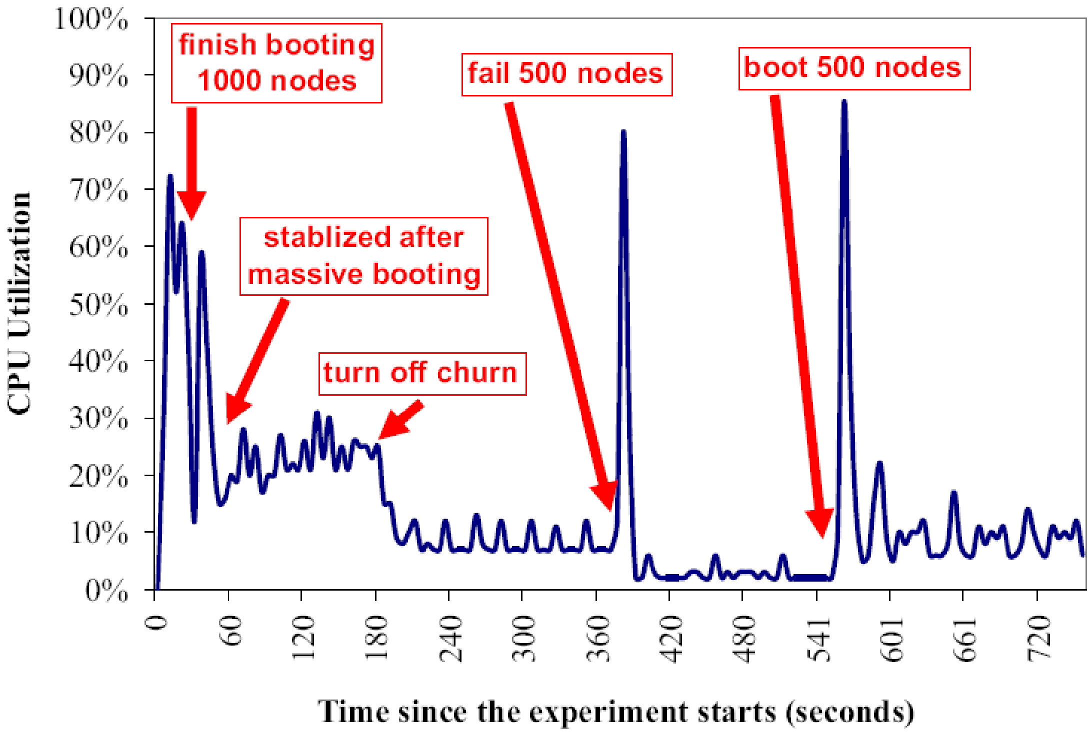


Fig. 4. The massive multi-tenancy mode runs 4,000 threads in a single JVM to concurrently execute (as opposed to simulate) 1,000 DHT nodes in real-time. This experiment was conducted on an IBM System x3850 server with four dual-core 3GHz Intel Xeon processors.

Simulation is Efficient and Scalable

- For a system with 1,000 BlueDHT nodes, it takes only 8 minutes to simulate one-hour activities in the real world

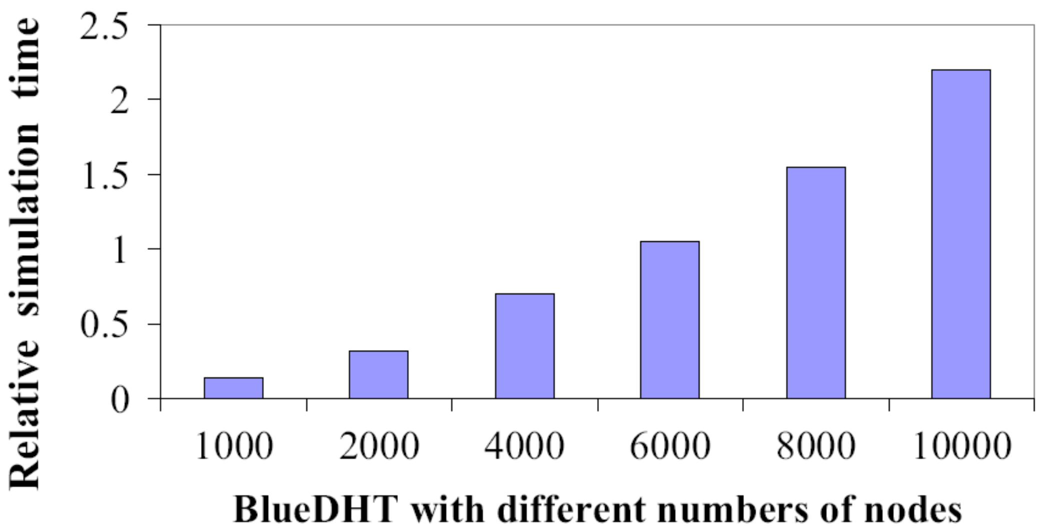


Fig. 5. Relative simulation time of BlueDHT with different numbers of nodes.

Checkpoint is Fast and Scalable

- For a system with 1,000 BlueDHT nodes, it takes 1.3 seconds to create a checkpoint, and the checkpoint size is only 13 MB
  - This efficiency is because DSF do not checkpoint the JVM process image

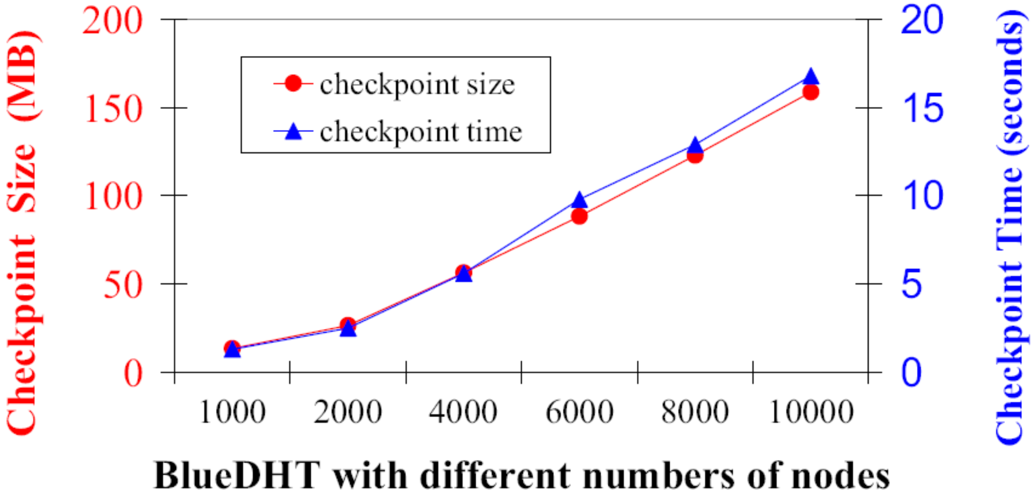


Fig. 6. Time needed to create a checkpoint for BlueDHT and the size of the checkpoint file.

Using DSF to Find Bug

- One real experience: a bug caused by out-of-order processing of a node's departure and re-join events
  - In an overlay network, suppose a node X fails and then reboots quickly
  - X's neighbor Y will process two events: X-fail and X-rejoin
  - However, due to network and thread scheduling delay, Y may process X-rejoin first and then X-fail. Therefore, Y considers X not a neighbor.
  - But X considers Y a neighbor because its rejoin protocol finishes successfully
- It is hard to trigger this bug in the real deployment mode, because X-fail and X-rejoin are rarely processed out of order
  - It is rare but can happen, e.g., due to long delay caused by Java garbage collection
- How DSF helped
  - Chaotic timing test in the simulation mode triggered the bug
  - Global consistency checking captured the bug automatically
  - Time travel debugging and mutable replay allows me to understand the bug instantly

The DSF API is almost as simple as java.util.TreeMap

```
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);
}
```

Conclusion and Status

- The goal is to trigger and fix 99% of the bugs in a single JVM
  - Chaotic timing test and mutable replay are powerful tools
  - Massive multi-tenancy mode can use thousands of threads to actually execute thousands of nodes in a single JVM
- DSF is simple. It is written purely in Java and does not modify or depend on any external tools.
- DSF was released in IBM recently, and some other IBM researchers just started to implement and evaluate their algorithms in it
- I would like to encourage broad reuse to the extent possible, and will see how far it can go