Deterministic Replay of Distributed Java Applications

Ravi Konuru
rkonuru@us.ibm.com

Harini Srinivasan
harini@us.ibm.com

Jong-Deok Choi
jdchoi@us.ibm.com

IBM Thomas J. Watson Research Center
30 Saw Mill River Road, Hawthorne, NY 10532

Abstract

Execution behavior of a Java application can be non-deterministic due to concurrent threads of execution, thread scheduling, and variable network delays. This non-determinism in Java makes the understanding and debugging of multi-threaded distributed Java applications a difficult and a laborious process. It is well accepted that providing deterministic replay of application execution is a key step towards programmer productivity and program understanding. Towards this goal, we developed a replay framework based on logical thread schedules *and* logical intervals*. An application of this framework was previously published in the context of a system called* DejaVu *that provides deterministic replay of multi-threaded Java programs on a single Java Virtual Machine(JVM). In contrast, this paper focuses on* distributed DejaVu *that provides deterministic replay of distributed Java applications running on multiple JVMs. We describe the issues and present the design, implementation and preliminary performance results of distributed* DejaVu *that supports both multi-threaded and distributed Java applications.*

1. Introduction

The relative simplicity of the Java programming language and its platform API has made Java attractive as an application development platform. Certain features of Java, such as multiple threads and network events, however, introduce non-determinism in application's execution behavior. Non-deterministic execution is a well known characteristic of concurrent systems and makes program understanding and debugging a difficult and a laborious process. For example, repeated execution of a program is common while debugging a program. Non-determinism may result in a bug to appear in one execution instance of the program and not appear in another execution instance of the same program. Further, the performance can be different from one execution of a program to another execution of the same program.

Providing deterministic replay of application execution is a key step towards programmer productivity and program understanding [5, 8, 4]. Towards this goal, we developed a replay framework based on *logical thread schedules* and *logical intervals*. An application of this framework was previously published in the context of a system called DejaVu that provides deterministic replay of multi-threaded Java programs on a single Java Virtual Machine(JVM)[2]. No modifications are necessary for standalone Java applications to take advantage of this replay facility. In contrast, this paper describes deterministic replay for distributed Java applications running on multiple JVMs. Our techniques for handling distributed events seamlessly integrate with our earlier work on replay for multi-threaded applications on a single JVM. The result of the integration is an efficient deterministic replay tool for multithreaded and distributed Java applications. We have implemented the deterministic replay techniques for distributed Java applications as extensions to the Sun Microsystems' JVM. We refer to the extended JVM as DJVM.

There are three major cases to consider for a distributed Java application, in terms of how much control the distributed DejaVu system can have over an application: 1) *closed world case*, where all the JVMs running the application are DJVMs; 2) *open world case*, where only one of the JVMs running the application is a DJVM; and 3) *mixed world case*, where some, but not all the JVMs running the application are DJVMs.

For a distributed Java application, DJVM needs to replay execution behavior as defined by Java network communication API. At the core, this API is centered around communication end points called sockets. Three socket types are supported: 1) a point-to-point stream or TCP socket that supports reliable, streaming delivery of bytes; 2) a point-to-point datagram or packet based UDP socket on which message packets can be lost or received out of order; and 3) a multicast (point-to-multiple-points) socket on which a datagram may be sent to multiple destination sockets. With respect to replay, multicast sockets are just a special case of UDP sockets. Behaviors of TCP and UDP sockets differ and therefore need different solutions for execution replay.

A DJVM runs in two modes: (1) *Record mode*, wherein, the tool records the *logical thread schedule information* and the *network interaction information* of the execution while the Java program runs; and (2) *Replay mode*, wherein, the tool reproduces the execution behavior of the program by enforcing the recorded logical thread schedule and the network interactions. DJVM uses a portable approach that is independent of the underlying thread scheduler. DJVM is, to our knowledge, the first tool that addresses the issues in handling all the non-deterministic operations in the context of deterministic replay of distributed and multithreaded Java applications. The approach is general and can be applied to distributed and multithreaded applications written in a language with features similar to Java.

The rest of the paper is organized as follows: Section 2 describes our replay framework and methodology and summarizes the application of these concepts for multithreaded Java applications. More detailed description can be found in [2]. Section 3 sets the context for describing distributed replay by providing a general idea on how the framework can be applied for replaying network activity. Section 4 explains the replay techniques for closed worlds for TCP and UDP sockets. Section 5 describes the techniques for TCP and UDP sockets for open and mixed world cases. Section 6 presents the DJVM implementation and some performance results. Section 7 compares our approach to previous approaches, and Section 8 concludes the paper.

2. Replay Framework

Replaying a multithreaded program on a uniprocessor system can be achieved by first capturing the thread schedule information during one execution of the program, and then enforcing the exact same schedule when replaying the execution [8]. A thread schedule of a program is essentially a sequence of time intervals (time slices). Each interval in this sequence contains execution events of a single thread. Thus, interval boundaries correspond to thread switch points.

2.1. Logical Thread Schedule

We refer to the thread schedule information obtained from a thread scheduler as the *physical thread schedule* information, and each time interval in a physical thread schedule as a *physical schedule interval*. Capturing the physical thread schedule information is not always possible, in particular, with commercial operating systems. Rather than relying on the underlying physical thread scheduler (either an operating system or a user-level thread scheduler) for physical thread scheduling information, we capture the *logical thread schedule* information [2] that can be computed without any help from the thread scheduler.

An execution behavior of a thread schedule can be different from that of another thread schedule, if the order of shared variable accesses is different in the two thread schedules. Thus, it is possible to classify physical thread schedules with the same order of shared variable accesses into equivalence classes. We collectively refer to all the physical thread schedules in an equivalence class as a *logical thread schedule*.

Synchronization events can affect the order of shared variable accesses. Examples of such synchronization operations in Java are synchronized methods/blocks and `wait`. We collectively refer to the events, such as shared variable accesses and synchronization events, whose execution order can affect the execution behavior of the application as *critical events*. A logical thread schedule is a sequence of intervals of critical events, wherein each interval corresponds to the critical and non-critical events executing consecutively in a specific thread.

2.2. Logical Schedule Intervals

The logical thread schedule of an execution instance on a uniprocessor system is an ordered set of critical event intervals, called *logical schedule intervals*. Each logical schedule interval, LSI_i, is a set of maximally consecutive critical events of a thread, and can be represented by its first and last critical events as: $LSI_i = \langle FirstCEvent_i, LastCEvent_i \rangle$.

The approach to capture logical thread schedule information is based on a global counter (*i.e.*, time stamp) shared by all the threads and one local counter exclusively accessed by each thread. The global counter ticks at each execution of a critical event to uniquely identify each critical event. Therefore, $FirstCEvent_i$ and $LastCEvent_i$ can be represented by their corresponding global counter values. Note that the global counter is global within a particular DJVM, not across the network (over multiple DJVMs). A local counter also ticks at each execution of a critical event. The difference between the global counter and a thread's local counter is used to identify the logical schedule interval on-the-fly [2].

The general idea of identifying and logging schedule interval information, and not logging the exhaustive information on each critical event is crucial for the efficiency of our replay mechanism. In the log file generated by the system, we have found it typical for a schedule interval to consist of thousands of critical events, all of which can be efficiently encoded by two, not thousands of counter values.

Each critical event is uniquely associated with a global counter value, which determines the order of critical events. Updating the global counter for a critical event and executing the critical event, therefore, are performed in one atomic operation for shared-variable accesses. We have implemented an application transparent, light-weight *GC-critical section* (for Global Counter critical section) code within the Java Virtual Machine that is used to to implement a single atomic action of critical events. It is used when the critical event is a general event, e.g. a shared variable access. Synchronization events with blocking semantics, such as `monitorenter` and `wait`, can cause deadlocks if they cannot proceed in a GC-critical section. Therefore, we handle these events differently by executing them outside a GC-critical section. (Detailed description on these can be found in [2].)

Updating the global counter and executing the event both in one single atomic operation is only needed during the record phase. For a thread to execute a schedule interval $LSI_i = \langle FirstCEvent_i, LastCEvent_i \rangle$, during the replay phase, the thread waits until the global counter value becomes the same as $FirstCEvent_i$ without executing any critical events. When the global counter value equals $FirstCEvent_i$, the thread executes each critical event and also increments the global counter value until the value becomes the same as $LastCEvent_i$.

3. Distributed DejaVu

In this section, we give a general idea of how the framework can be applied to DJVM in an extensible manner to handle both multi-threaded and distributed Java applications in closed, open and mixed world environments. In each of these environments, we ensure deterministic replay of the distributed Java application by identifying network events as critical events (we describe the details for each network event in the subsequent sections). These network events can potentially change the observable execution behavior of the distributed Java application.

Performing critical events with blocking semantics, such as `connect`, `accept` and `read`, in a GC-critical section can lead to deadlocks. We avoid deadlocks due to these blocking network events by executing them out of a GC-critical section. We provide more details on avoiding deadlocks due to blocking network events in the subsequent sections.

Execution order of critical events up to the first network event will be preserved by the DJVM even without the support for network events. The support in DJVM for network events ensures that the network events happen in the same execution order as in the record mode. With network support in DJVM, we can conclude by induction that DJVM can deterministically replay all critical events, network or non-network. In the following sections, we describe the different solutions in DJVM for replay of network events.

The closed world case is by far the more complex. The replay techniques for TCP and UDP sockets in this world is described in Section 4. We then give an overview in Section 5 of how replay can be supported for both stream and datagram sockets in open and mixed worlds.

4. Closed World Case

Replay for TCP sockets is described in Section 4.1, and that for UDP sockets is described in Section 4.2.

4.1. Supporting Stream Sockets

We first outline the Java APIs for stream sockets followed by issues in replaying distributed applications that use stream sockets, and our techniques to record and replay for deterministic replay.

4.1.1. Java Stream Socket API

In Java, stream sockets are created by `Socket` and `ServerSocket` classes. A client constructs a `Socket` object to connect to a server. In the process of executing the `Socket()` constructor, the client will execute the `connect()` call. The construction is blocked until a connection is established by a server. A server constructs a `ServerSocket` object to specify the port to listen on. It then invokes the `accept()` method of the object to accept a connection request. The `accept()` blocks until a connection is established. It then creates and returns a `Socket` object. The server can close the socket connection via `close()` method of the `ServerSocket` object.

Once a socket is created, `getInputStream()` and `getOutputStream()` of the `Socket` object return `InputStream` and `OutputStream` objects to be used for reading (via `read()` method call) and writing (via `write()` method call) stream data over the socket stream.

Other socket APIs include: method to listen for connections on a stream socket (`listen()`), method to bind a socket to a local port (`bind()`), and method to determine the number of bytes that can be read without blocking (`available()`).

4.1.2. Issues in Replaying Stream Socket Events

Each stream socket call (`accept`, `bind`, `create`, `listen`, `connect`, `close`, `available`, `read`, `write`) is mapped into a native method call in a JVM implementation. We identify each of these native calls as a network event. For convenience, in the rest of the paper, we use the names of the Java calls for the corresponding native calls when referring to the critical events. For replaying stream socket network events, the following issues are relevant.

Variable network delays: Since network delays can vary for different executions of the same distributed Java application, socket connections can be non-deterministic. Therefore, the first step for deterministic replay of network events is deterministic re-establishment of socket connections among threads. The relevant socket calls that are affected by this issue are `accept()` and `connect()`.

Figure 1 illustrates this issue with an example. The server application in the figure has three threads `t1, t2, t3` waiting to accept connections from clients. (This would be a typical scenario in distributed Java applications). Client1, Client2 and Client3 execute the `connect()` call, making connection requests to the server. The solid and dashed arrows indicate the connections between the server threads and the clients during two different executions of the distributed Java application.

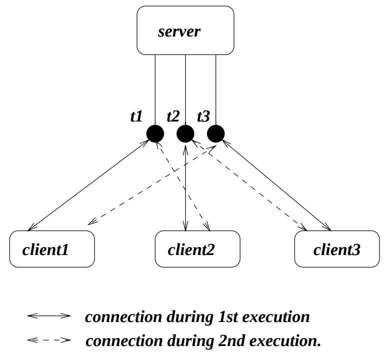
server
t1
t2
t3
client1
client2
client3
connection during 1st execution
connection during 2nd execution.

Figure 1. Figure illustrating network-delays issue

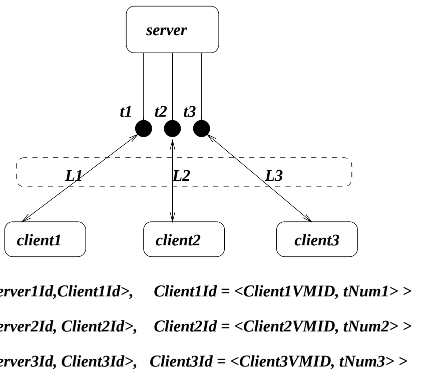
server
t1
t2
t3
L1
L2
L3
client1
client2
client3
L1: <Server1Id,Client1Id>, Client1Id = <Client1VMID, tNum1> >
L2: <Server2Id, Client2Id>, Client2Id = <Client2VMID, tNum2> >
L3: <Server3Id, Client3Id>, Client3Id = <Client3VMID, tNum3> >

Figure 2. Figure illustrating mechanism for deterministic replay of connections

Variable message sizes: The stream-oriented nature of the connections can result in variable length messages read by the receiver of the messages. In other words, the `read()` method calls can return less than or equal to the number of bytes requested. A simple re-execution of the `read()` method during the replay phase can result in a different number of bytes being read than the number read in the record phase.

Network queries: Operations related to the status and attributes of a connection need to be replayed. For instance, if the particular port number was allocated to a socket during the record phase, the application should *see* the same port number during the replay phase. The relevant socket calls affected by this issue are `available()` and `bind()`.

Blocking calls: Socket calls such as `accept`, `connect`, `read`, and `available` are blocking calls. Hence, if these calls are placed within the *GC-critical section*, they can cause the entire DJVM to be blocked until the call completes, and can result in deadlocks and inefficient and heavily perturbed execution behaviour.

4.1.3. Record and Replay Mechanism for Stream Sockets

Each DJVM is assigned a unique JVM identity (DJVM-id) during the record phase. This identity is logged in the record phase and reused in the replay phase. The DJVM-id allows us to identify the sender of a message or connection request. A network event on a DJVM is identified by a *networkEventId* defined as the tuple $\langle threadNum, eventNum \rangle$, where *threadNum* is the thread number of the specific thread executing the network event and *eventNum* is a number that identifies the network event within that thread. The *eventNum* is used to order network events within a specific thread. In addition, we also use the *connectionId* to identify a connection request at a `connect` network event. The *connectionId* is the tuple, $\langle dJVMId, threadNum \rangle$, where *dJVMId* is the identity of the DJVM at which the `connect` event is being generated, and *threadNum* is the thread number of the client thread generating the connection request. Since threads are created in the same order in the record and replay phases, our implementation guarantees that a thread has the same *threadNum* value in both the record and replay phases. In addition, since events are sequentially ordered within a thread, the *eventNum* of a particular network event executed by a particular thread is guaranteed to be the same in the record and replay phases.

In the rest of this section, we discuss our techniques for record and replay phases that handle the issues previously outlined. In the following, we refer to a client's DJVM as DJVM-client and a server's DJVM as DJVM-server. Further, we use the name *NetworkLogFile* to denote the per DJVM log file where information required for replaying network events is recorded.

Replaying `accept` and `connect`: Since these calls are a source of non-determinism, these are made DJVM critical events. Although this guarantees the execution order within a DJVM, it alone is not sufficient for correctness because of non-determinism introduced by network delays. So additional data regarding the connection is stored at the server DJVM. Further, as mentioned earlier, these calls are blocking calls and executing these calls within a GC-critical section reduces application parallelism and introduces potential for deadlocks. We therefore allow the operating system level network operations to proceed and then mark the network operations as critical events. This marking strategy allows threads performing operations on different sockets to proceed in parallel with minimal perturbation.

In the record phase, at the `connect`, DJVM-client sends the socket-connection request to the server, possibly accepted by a peer thread on the server. When the socket connection is finally established, the client thread on DJVM-client sends the *connectionId* for the `connect` over the established socket as the first data (meta data). Note that the *connectionId* is sent to the server via a low level (native) socket `write` call and is done before returning from the `Socket()` constructor call. This ensures that the *connectionId* is indeed the first data sent over this connection. Finally, just before the `connect` call returns, DJVM-client performs the GC-critical section for atomically updating the global counter.

In the replay phase, DJVM-client executes the `connect` and sends the *connectionId* of the `connect` to the server as the first meta data, just as in the record phase. Since `connect` is a critical event, DJVM-client ensures that the `connect` call returns only when the `globalCounter` for this critical event is reached.

On the server side, during the record phase, at an `accept`, the DJVM-server accepts the connection and receives the *connectionId* sent by the client as the first meta data at the corresponding `connect`. The DJVM-server also logs the information about the connection established into the NetworkLogFile. For each successful accept call, the log contains an entry, called a *ServerSocketEntry*, which is the tuple, $\langle serverId,\ clientId \rangle$, where *serverId* is the *networkEventId* of the corresponding `accept` event and *clientId* is the *connectionId* sent by the DJVM-client. Given the information stored in a tuple, it is possible for two different threads to have identical ServerSocketEntry tuples in their part of the NetworkLogFile. However, this lack of unique entries is not a problem. The core, single DJVM ensures the replay of invocation order (not completion) of accepts across threads since `accept` is a synchronized call. Thus for example, if during record phase, a thread *t1* invoked the accept method on a socket before thread *t2*, the thread *t1* will invoke the accept method before t2 during replay. Since the client threads also execute their connects in the original order, the connection gets re-established between the same two threads as during original execution. Further, note that an exception thrown by a network event in the record phase is logged and re-thrown in the replay phase.

```
enterFDCriticalSection( socket )
write
enterGCCriticalSection
leaveGCCriticalSection
leaveFDCriticalSection(socket)

----

done = false;
while(!done) \{
  enterFDCriticalSection( socket )
  if (ReplayPhase) {
     n = read(.. recordedValue .. )
     if ( n == recordedValue)
         done = true;
     if (n < recordedValue)
        recordedValue = recordedValue - n;
     if (n > recordedValue)
        error;
  }
  if (RecordPhase) {
     n = read(..)
     recordedValue = n
     done = true;
  }
  enterGCcriticalSection
  leaveGCcriticalSection
  leaveFDCriticalSection( socket )
}
```

Figure 3. Efficient replay of read, write

To replay `accept` events, a DJVM maintains a data structure called *connection pool* to buffer out-of-order connections. During the replay phase, when an `accept` is executed by a server thread t_s on DJVM-server, it first identifies the *networkEventId* for this `accept` event, *i.e.*, $\langle threadNumof t_s,\ eventNumofacceptwithint_s \rangle$. It then identifies the *connectionId* from the NetworkLogFile with matching *networkEventId* value. DJVM-server then checks the *connection pool* to see if a `Socket` object has already been created with the matching *connectionId*. If the `Socket` object has been created, it simply returns the `Socket` object to complete the `accept`. If a `Socket` object has not already been created with the matching *connectionId*, the DJVM-server continues to buffer information about out-of-order connections in the connection pool until it receives a connection request with matching *connectionId*. It then creates and returns a `Socket` object for the matching connection.

Example: For the example in Figure 1, suppose the solid arrows indicate the connections established during the record phase. Figure 2 shows our mechanism for deterministically replaying the same connections. *L1, L2, L3* are the log entries made during the `accept` events by threads *t1, t2, t3* respectively during the record phase. *Server1Id, Server2Id, Server3Id* are the *networkEventId* values when *t1*, *t2*, *t3* respectively execute the `accept` events. The figure also shows the *connectionId*s sent by each client. For example, the *connectionId* from Client2 is $Client2Id = \langle Client2VMId, tNum2 \rangle$.

Replaying `read`: Socket `read` event is identified as a critical event in a DJVM. Since the number of bytes read via a socket `read` can vary for different executions, in the record phase, the DJVM executes the `read` and logs the thread-specific *eventNum* and number of bytes read (*numRecorded*) in the NetworkLogFile. Since `read` is a blocking call, it is not placed within a *GC-critical section*. Instead, just before the `read` call returns, the DJVM marks the `read` as a critical event. In the replay phase, at the corresponding `read` event, the DJVM thread retrieves the *numRecorded* number of bytes from the NetworkLogFile corresponding to the current *eventNum*. Further, the thread reads only *numRecorded* bytes even if more bytes are available to read or will block until *numRecorded* bytes are available to read. Finally, the execution returns from the `read` call only when the `globalCounter` for this critical event is reached.

Replaying `write`: `write` is a non-blocking call and a critical event. `write` is handled by simply placing it within *GC-critical section* similar to how we handle critical events corresponding to shared variable updates.

Since `SocketInputStream.read` and `SocketOutputStream.write` are not synchronized calls, multiple writes on the same socket may overlap. While replaying the writes and the corresponding reads, we have to ensure that all the writes to the same socket happen in the same order and all the reads (from the socket) read the bytes in the same order in both the record and replay modes.

A solution is to just record the occurrence of such an event and allow other unrelated events (*i.e.*, events that do not operate on the same socket) to proceed. Events that do use the same socket will be blocked by using a lock variable for each socket (Figure 3). This scheme allows some parallelism in the record and replay modes and also preserves the execution ordering of the different critical events. The additional cost in this scheme is the cost of the extra lock variables per socket and the slightly increased implementation complexity.

Replaying `available` and `bind`: The `available` and `bind` events are also treated as critical events. Both these events implement network query. In the case of `available`, it checks the number of bytes available on the stream socket, and `bind` returns the local port to which the socket is bound.

Since `available` is a blocking call, in the record phase, it is executed before the *GC-critical section*. In addition, the DJVM records the actual number of bytes available. In the replay phase, the `available` event can potentially block until it returns the recorded number of bytes, *i.e.*, until the recorded number of bytes are available on the stream socket.

In the case of `bind`, in the record phase, it is executed within a *GC-critical section* and the DJVM records its return value. In the replay phase, we execute the `bind` event, passing the recorded local port as argument.

Other stream socket events: The other stream socket events that are marked as critical events are `create`, `close` and `listen`, all of which have to be recorded to preserve execution order. We handle these critical events by simply enclosing them within our *GC-critical section*, similar to how we handle critical events corresponding to shared variable updates.

4.2. Supporting Datagrams Sockets

TCP socket is designed to be reliable. If data is lost or damaged during transmission, TCP ensures that the data is sent again. If data or packets arrive out of order, TCP rearranges them to be in the correct order. UDP, User Datagram Protocol, is an alternative protocol to send data over the network, and is unreliable. The packets, called datagrams, can arrive out of order, duplicated, or some may not arrive at all. It is the (Java) application's responsibility to manage the additional complexity.

For deterministic replay of applications using UDP, DJVM needs to ensure the same packet delivery behavior during the replay phase as during the record phase. In other words, the replay mechanism must ensure that the packet duplication, packet loss and packet delivery order in the record phase is preserved in the replay phase. The following sections describe how we achieve this replay for datagram sockets. Multicast sockets can be easily accommodated by extending the mechanism for datagram sockets from a point-to-single-point scheme to a point-to-multiple-points scheme.

4.2.1. Datagram Socket API

UDP sockets are created via `DatagramSocket` class. A `DatagramPacket` object is the datagram to be sent or received through the `DatagramSocket` object via `send()` and `receive()` methods of the `DatagramSocket` object. They are both blocking calls. A datagram socket is closed via `close()` method of the socket object. As in the case of stream sockets, each of the datagram socket call can be implemented in a JVM via a low-level native call. We use the names of the UDP socket calls to refer to the low level native calls (network events). The UDP `send`, `receive` and `close` events are critical events in DJVM.

4.2.2. Record Phase

During the record phase, the sender DJVM intercepts a UDP datagram sent by the application, called *application datagram*, and inserts the *DGnetworkEventId* of the `send` event at the end of the data segment of the application datagram.[1] The *DGnetworkEventId* is the pair $\langle dJVMId, dJVMgc \rangle$, where *dJVMId* is the id of the sender DJVM and *dJVMgc* is the globalcounter at the sender DJVM associated with the send event. The receiver DJVM intercepts a datagram to be received by a thread, and delivers to the application only the application datagram by stripping out the datagram id. The datagram size, due to the meta data, can become larger than the maximum size allowed for a UDP datagram (usually limited by 32K). When this happens, the sender DJVM splits the application datagram into two, which the receiver DJVM combines into one again at the receiver side. A split datagram carries the same *DGnetworkEventId*, and a flag to indicate the portion, i.e. *front* or *rear*, it represents so that it can be combined correctly at the receiver. A non-split datagram carries its own flag that distinguishes it from a split datagram.

The receiver DJVM logs all the datagrams received into a log called *RecordedDatagramLog*. Each entry in the log is a tuple $\langle ReceiverGCounter, datagramId \rangle$, where *ReceiverGCounter* is the global counter value at the `receive` event in the receiver DJVM, and *datagramId* is the *DGnetworkEventId* of the received datagram. Multiple datagrams with identical *DGnetworkEventId* are also recorded during the record phase.[2]

4.2.3. Replay Phase

For reliable delivery of UDP packets during replay, we use a *reliable UDP* mechanism[3] that guarantees reliable, but possibly out of order, delivery of intrinsically unreliable UDP datagrams. Note that a datagram delivered during replay need be ignored if it was not delivered during record.

[1] The DJVM also increases the length field of the datagram to include the added size for the datagram id.

[2] Note that the same datagram can be delivered more than once during the record phase, all of which must be delivered to the application during record and replay.

[3] If no reliable UDP is available, a pseudo-reliable UDP can be implemented as part of the sender and the receiver DJVMs by storing sent and received datagrams and exchanging acknowledgment and negative-acknowledgment messages between the DJVMs.

For UDP delivery, the *DGnetworkEventId* of each UDP packet is used for uniquely identifying each datagram. A datagram entry that has been delivered multiple times during the record phase due to duplication is kept in the buffer until it is delivered to the same number of `read` requests as in the record phase.

5. Open and Mixed World Cases

In the open world case, only one component of the distributed application is running on a DJVM. Network events, in this case, are handled as general I/O: any input messages are fully recorded including their contents during the record phase. During the replay phase, any network event at the receiver DJVM is performed with the recorded data, not with the real network. For example, during the record phase, a client DJVM requesting a stream socket connection to a non-DJVM server logs the connection data. During the replay phase, the results of the corresponding connection request are retrieved from the log. The actual operating system-level `connect` call is not executed. Likewise, any message sent to a non-DJVM thread during the record phase need not be sent again during the replay phase.

In a mixed-world case, some components of the application are running on DJVM and others on non-DJVM. If the environment is known before the application executes, one could simply fall back on the DJVM scheme for the open-world case. However, with a little bit more machinery, it is possible to optimize on space overheads by using the closed-world scheme for communication with DJVMs and saving additional state during the communication with non-DJVMs.

6. Implementation and Performance Results

We modified Sun's implementations of the JDK on Windows NT operating system to provide record and replay functionality of stream sockets for both closed and open world environments. In this section, we use a synthetic, multi-threaded, client-server benchmark to demonstrate the performance and various overheads of Dejavu. This benchmark, that uses only stream socket API for network calls, has been written to deliberately contain non-determinism in updating both shared variables and passing the result of computation over these shared variables between the client and the server. For instance, the number of connections performed for the client is a shared variable that is updated without exclusive access by the client threads and this variable is used in the individual thread computations. Further, the client threads perform multiple connects per "session" that introduces additional non-determinism in the order of establishing connections. Because of these many sources of non-determinism, repeated executions of the benchmark invariably complete with different results computed by each thread. However, when DJVM is used, a perfect replay is observed.

The benchmark was run on a IBM Thinkpad with one 300Mhz Pentium processor, running Windows NT operating system. The client and server were run on different DJVMs on the same machine. The performance results of DJVMs for this benchmark are shown under the open and the closed scenarios in table 1 and table 2, respectively. The tables show how the implementation performs with increasing number of threads in each component. Since there are two parts to the application, i.e., the client and the server part, the results are so tabulated. The first and the second columns of each table are self-explanatory. The column labeled *#nw events* is the number of critical events that are also network events, such as `accept`, `connect`, and `read`. The column labeled *log size* is the total size in bytes of the recorded information. This includes the list of scheduling intervals for each thread and information related to network activity. The column *rec ovhd* is the percentage increase in application execution time due to our modifications to the JVM for supporting replay.

First, notice that the number of network events for the server component in both closed and open world cases is the same. (See table 1(a) and table 2(a).) This is because the identification of a network critical event is independent of the recording methodology. For the same reason, the client component exhibits the same number of network events under closed and open scenarios. Second, as expected, the recording overhead in space and time (*log size* and *rec ovhd*) for the closed world is less than that for the open world because the amount of information that is recorded differs. Note that increasing the size of messages sent to the client would not change the size of closed-world log but would cause a consequent increase in the open-world log. Finally, in all cases, as the total number of application threads (client and server threads) executing on the machine increase from 4 to 64, the recording overhead rises super linearly. We attribute to this to the extensive thread context switching, that results in larger number of invocations of replay support code, and thread contention for the GC-critical section. The worst-case overhead for this benchmark is 57% for closed-world implementation and 69.57% for open-world. Given that the implementation is still work-in-progress with unoptimized and debugging code, the result demonstrates that the tool can be practical. However, this needs to be verified against real applications.

7. Related Work

Replay is a widely accepted technique for debugging deterministic sequential applications. Replay for debugging, however, fails to work for non-deterministic applications, such as distributed and multithreaded Java applications. BUGNET's handling of non-deterministic message sent and received by processes is similar to our handling of UDP datagrams [3]. It logs the received message id's during the record phase, and consumes the received messages according to the log during the replay phase while buffering yet to be consumed messages. It does not address the issue of non-deterministic events due to multithreading within a process that interact with non-deterministic message receives, nor does it address non-deterministic partial receive of messages through "reliable" connections.

Replay systems based on *Instant Replay* [5, 9] address both non-determinism due to shared-variable accesses and messages. Each access of a shared variable, however, is modeled after interprocess communication similar to message exchanges. When the granularity of the communication is very small, such is the case with multithreaded applications, the space and time overhead for logging the interactions becomes prohibitively large. Instant Replay also addresses only atomic network messages like the UDP datagram.

Russinovich and Cogswell's approach [8] addresses specifically multithreaded applications running only on a uniprocessor system. They modified the Mach operating system to capture the physical thread scheduling information. This makes their approach highly dependent on an operating system. The scheme by Levrouw et. al. for event logging computes consecutive accesses for each object, using one counter for each shared object [6]. Our scheme differs from theirs in that ours computes logical thread schedule, using a single global counter. Our scheme is, thereby, much simpler and more efficient than theirs on a uniprocessor system. Neither of these addresses replaying distributed applications.

Netzer et. al. address the issue of how to balance the overhead of logging during the record phase with the replay time [7]. Even for a closed world system, they store contents of messages selectively to avoid executing the program from the start. Combined with checkpointing [10], storing contents of messages allows for bounded-time replay to arbitrary program points.

8. Conclusions

We have developed a record/replay tool for distributed and multi-threaded Java applications, called DJVM that provides a deterministic replay of a non-deterministic execution. DJVM is implemented by modifying Sun Microsystem's Java Virtual Machine (JVM). Our approach is independent of the underlying thread scheduler such as the operating system. It also does not require modifications of the user application to enable replay. Future work includes

#threads	#critical events	#nw events	log size(bytes)	rec ovhd(%)
2	493758	69	504	0.7
4	512374	113	824	1.1
8	550021	201	1464	2.4
16	627076	377	4692	4.5
32	781717	729	16072	16.9

(a) **Server**

#threads	#critical events	#nw events	log size(bytes)	rec ovhd(%)
2	492505	72	436	2.3
4	510020	120	676	2.9
8	545130	216	1260	9.3
16	617318	408	1868	16.0
32	761662	792	3908	57.0

(b) **Client**

Table 1. Closed-world results

#threads	#critical events	#nw events	log size(bytes)	rec ovhd(%)
2	20762	69	3856	0.75
4	34328	113	6392	1.62
8	61658	201	11391	2.02
16	117872	377	21571	10.08
32	230084	729	40903	18.44

(a) **Server**

#threads	#critical events	#nw events	log size(bytes)	rec ovhd(%)
2	19354	72	4030	3.63
4	31950	120	6691	4.7
8	57417	216	12010	5.18
16	110084	408	22714	29.13
32	215301	792	44266	69.57

(b) **Client**

Table 2. Open-world results

integrating the system with checkpointing to bound the replay time. DJVM is the first tool that completely addresses the issues in handling all the non-deterministic operations in the context of deterministic replay of distributed and multithreaded Java applications. The approach is general and can be applied to distributed and multithreaded applications written in a language with features similar to Java. We also plan to apply the techniques to *Jalapeño*, a JVM for SMP machines, being developed at IBM Research [1].

Acknowledgments: We thank the anonymous referees for helpful comments and Olivier Gruber for useful discussions.

References

[1] B. Alpern, D. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, T. Ngo, M. Mergen, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalepeno virtual machine. *IBM Systems Journal*, 39(1), 2000.

[2] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, August 1998.

[3] R. Curtis and L. Wittie. BUGNET: A debugging system for parallel programming environments. In *Proceedings of the 3rd IEEE International Conference on Distributed Computing Systems*, pages 394–399, 1982.

[4] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, Seattle, Washington, Mar. 1990. ACM Press.

[5] T. J. Leblanc and J. M. Mellor-Crummy. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–481, April 1987.

[6] L. Levrouw, K. Audenaert, and J. V. Campenhout. Execution replay with compact logs for shared-memory systems. *Proceedings of the IFIP WG10.3 Working Conference on Applications in Parallel and Distributed Computing, IFIP Transactions A-44*, pages 125–134, April 1994.

[7] R. Netzer, S. Subramanian, and X. Jian. Critical-path-based message logging for incremental replay of message-passing programs. In *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems*, June 1994.

[8] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. *Proceedings of ACM SIGPLAN Conference on Programming Languages and Implementation (PLDI)*, pages 258–266, May 1996.

[9] J. Sienkiewicz and T. Radhakrishnan. DDB: A distributed debugger based on repaly. In *Proceedings of the IEEE Second International Conference on ICAPP*, pages 487–494, June 1996.

[10] Y. M. Wang and W. K. Fuchs. Optimistic message logging for independent checkpointing in message-passing systems. In *Proceedings of IEEE Symposium on Reliable Distributed Systems*, pages 147–154, Oct. 1992.