Extending JPF to Verify Distributed Systems

Nastaran Shafiei NASA Ames Research Center Moffett Field California 94035 nastaran.shafiei@nasa.gov Peter Mehlitz NASA Ames Research Center Moffett Field California 94035 peter.c.mehlitz@nasa.gov

ABSTRACT

This paper presents our work on model checking distributed applications. We refer to distributed applications as a collection of communicating processes, regardless of their physical locations and the communication means. Our work targets applications written in Java. It relies on the multiprocess support included in Java Pathfinder (JPF) version 7 which allow for verifying the bytecode of distributed applications. The basic support for distributed applications in JPF does not account for communication between processes. In this work, we address this limitation. The work is implemented as a JPF extension which models interprocess communication (IPC) mechanisms. It uses a form of partial order reduction (POR) to explore all possible executions of a distributed Java application. Moreover, our approach provides a functionality to check the given distributed application against possible network failures which can occur at the operating system or the hardware layer.

Keywords

Java Pathfinder, Verification, Model Checking, Distributed Systems

1. INTRODUCTION

With ubiquitous use of networked and mobile devices, distributed computing is becoming increasingly important. There are several key factors driving the development of distributed applications [1]. Some services intrinsically require the use of a communication network to connect different components, such as massively multiplayer online games. Distributed computing can also be used to develop fault tolerant systems where a failure in one process does not stop the application from completing its task. Distributed systems can utilize multiple machines to finish tasks faster and handle larger problems. Finally, distributed applications can use shared resources to reduce computational requirements on client devices, e.g. by means of cloud computing.

Distributed applications are hard to develop. These applications are inherently concurrent. Moreover, they are subject to problems related to the distributed setting. For instance, they need to deal with asynchronous failures in external processes and the network, resulting in unavailability of data and quality of service (QoS) problems. They might also lack a consistent view of data across the whole system and might have to adhere to complex protocols.

The most common technique to verify distributed systems is testing. However, since different components of the system may have different software and hardware requirements, setting up an environment to test such applications can be difficult, time consuming and expensive. In addition, network failures have to be simulated by instrumenting the system under test (SUT) or replacing operating system level components such as network libraries. Model checking is naturally preferred to testing when concurrency comes into play. Unlike testing that does not have any control over the scheduling of concurrent processes without SUT modification, model checking can examine all relevant scheduling sequences in a systematic way.

Our work to verify distributed applications is based on model checking. Due to its extensibility, we use Java Pathfinder (JPF) as the underlying model checker and implement our work as a standard JPF extension that can be runtime configured. Off the shelf, JPF includes a specialized Java Virtual Machine (JVM) that can model multiple Java processes. However, JPF does not provide library implementations that allow such processes to communicate via standard network APIs. This paper explains our work that addresses this limitation.

2. RELATED WORK

Model checking distributed applications is nontrivial. Most existing model checkers can only be applied to a single process SUT. One of the proposed techniques to model check distributed Java applications is *centralization*. This technique maps separate processes of a distributed system into threads executed within the same process, to capture all possible executions of the resulting program [2, 3, 4].

Every Java process provides a self-contained execution environment including an exclusive set of basic runtime resources. Since centralization captures the entire application within a single process, parts that represent different processes share the same resources. In order to preserve the behavior of the original distributed system, one needs to ensure proper separation of types in absence of process boundaries enforced by the operating system. This is known as a main challenge in applying centralization approaches which can be achieved at different levels.

Most existing approaches use centralization at the SUT level. These techniques modify the SUT code to separate types, and provide their own models of IPC mechanisms. The centralization technique proposed by Artho and Garoche [3] models socket based communication. In the work by Stroller and Liu [2], remote method invocation (RMI) calls are replaced with local ones that simulate RMIs. The Barlas and Bultan [5] work focuses on the environment generation problem for network communication, and presents a framework, *NetStub*, that models the Java networking packages java.net and java.nio.

A major drawback of centralization at the SUT level is that they do not impose type separation to non-SUT code such as standard Java libraries. Therefore, different parts of the SUT representing different processes share the same standard classes which may interrupt the correct behavior of the application. Moreover, since this approach transforms the structure of the SUT, it cannot support code that uses the Java reflection API. These problems are addressed by an alternative approach that performs centralization at the model checker level, requiring the model checker to directly support verification of multiple processes. JPF version 7 provides the basic building blocks for such a support. As an input, it can accept multiple Java processes. However it lacks IPC models and process aware scheduling. Our work adds these missing components for socket based communication. Moreover, compared to the approaches based on the centralization at the SUT level, our IPC model provides a more efficient way to capture thread nondeterminisms. It applies a POR technique which results in smaller state spaces for distributed applications.

3. JPF CENTRALIZATION

JPF can accept a distributed system as a SUT in its original form and uses a specialized JVM (referred to as *multiprocess JVM*) to execute multiple processes. The basic support for distributed applications is achieved through a centralization approach which is included in JPF version 7 and involves three major steps: separating types, modeling processes and adding process aware scheduling.

To separate types between processes, a new class loading model is implemented in JPF which extends the class loading model of Java. Class loaders are system objects that perform on-demand lookup of class files at runtime, and transformation of such data into java.lang.Class instances representing Java types. A Java type is identified by its fully qualified class name *and* its class loader instance. Unlike a normal JVM, JPF supports multiple instances of system class loaders through its gov.nasa.jpf.vm. SystemClassLoaderInfo construct. By giving each process its own system class loader, we can ensure type separation between processes regardless of classfile location (SUT or system libraries), which also gives each process exclusive access to static fields. For further details, the reader is referred to [6].

Processes are modeled in JPF as groups of threads mapping to the same gov.nasa.jpf.vm.ApplicationContext object. Upon system initialization, JPF creates a new ApplicationContext instance for each process and stores it within the instance of gov. nasa.jpf.vm.ThreadInfo that represents the main thread of the process. Each new thread that is created by the SUT automatically inherits the ApplicationContext of the currently executing thread. This mechanism ensures the thread-to-process association during the entire execution. ApplicationContext encapsulates per-process information such as main class, command line arguments, system class loader and classpath.

To capture scheduling points in the case of distributed applications an instance of gov.nasa.jpf.vm.DistributedScheduler-Factory is used. However, this component does not account for communication between processes. Basically, it does not create thread choices upon network interactions involving different processes. As part of this work, we extend this component to capture scheduling points for network API calls.

All three features are controlled by the multiprocess JVM (encapsulated by gov.nasa.jpf.vm.MultiProcessVM) which instantiates process main threads along with their respective Application-Context and SystemClassLoaderInfo objects. It starts the SUT execution from an initial state that has a scheduling choice for each process main thread. The multiprocess JVM also addresses shutdown semantics of each process by terminating its threads and executing related process specific tasks such as shutdown hooks.

4. JAVA NETWORKING

Java has several features that make it suitable for developing distributed applications [1]. It is platform independent, supports multithreaded programming, and offers an exception handling mechanism which is useful for developing fault tolerant applications. Through its standard libraries, Java provides multilevel support for basic networking constructs such as stream and datagram sockets. At a higher level, it provides networking capabilities such as remote method invocations, distributed objects, and communication with external databases. Finally, Java supports a rich set of security mechanisms that are essential for distributed applications, such as user authentication, data encryption and sandboxing (restricted access to the local file system).

The most commonly used networking API in Java is based on sockets [1]. A socket represents an endpoint of a communication channel between two processes. Every socket is identified by two elements, an *IP address* and a *port number*. An IP address is a unique address to identify a host across a network which is based on the Internet protocol (IP). A port number identifies the communication endpoint within a host and is usually associated with a specific service or protocol.

One of the main types of sockets in an IP network is based on the transport control protocol (TCP). TCP provides a reliable communication channel that guarantees the successful delivery of data packets in order. Java provides support for TCP sockets through classes in the java.net package, such as java.net. Socket, java.net.InetAddress, and java.net.ServerSocket. Data is exchanged through TCP sockets via a pair of input and output streams.

4.1 Client/Server Example

The code in Figure 1 demonstrates a simple example in which a server and a client communicate using TCP sockets. The Java program on the left side of the figure represents the server, the one on the right represents the client. Henceforward, we use s.i and c.i as a notation to refer to the i^{th} statement of the server and client, respectively.

A TCP socket in Java can be either passive or active. To establish a connection, the server creates a passive socket (encapsulated by java.net.ServerSocket) that is associated with a given port (s.5), and then it blocks (s.6) until it receives a connection request from a client.

The client in Figure 1 creates an active socket (encapsulated by java.net.Socket) which sends a connection request to a server that is running on the same host and waiting for connections on the specified port (c.5). If there is no server socket associated with the given host and port, Java throws a java.io.IOException that has to be handled in the client code. Otherwise, the server accepts the connection request from the client and obtains a new active socket representing the server endpoint of the connection (s.6). At this point the connection is established, the server and the client can start to exchange data.

The server and the client receive and send data through their respective socket streams, which are obtained by calling the Socket. getInputStream() and Socket.getOutputStream() methods (s.7 & c.6). Next, the server attempts to read a request from the client (s.8). If data has not been sent yet, the server input operation blocks until the client writes some data (c.7). Once the server receives the request, it terminates by closing its sockets (s.9 & s.10). Closing the ServerSocket object avoids new clients from connect-



Figure 1: A simple client and server that communicate through TCP sockets.

ing to this server, and closing the **Socket** object disconnects this server from the client.

5. JPF-NAS

Our networked asynchronous systems extension (jpf-nas) of JPF provides the functionality to verify distributed Java applications. At this stage, jpf-nas supports Java processes that communicate via TCP sockets, and models the basic networking classes such as java.net.ServerServer and java.net.Socket. Our implementation consists of two main components: *connection manager* and *scheduler*. The former maintains communication channels along the current execution path, and the latter captures scheduling points upon network interactions.

5.1 Connection Manager

Modeling socket-based communication in JPF requires shared buffers that are invisible from the SUT. This can be achieved through the model Java interface (MJI) of JPF. MJI is used to transfer the execution from the JPF level to the host JVM level by means of native peers. These classes are completely unknown to the SUT and execute on top of the host VM. Using a specific name pattern, JPF associates methods of the SUT classes to the methods of native peers. Whenever JPF has to execute a SUT method associated with a native peer method, the execution is transferred to the host VM which executes the native peer method.

The connection manager, encapsulated by nas.java.net. connection.ConnectionManager, exists at the host JVM level. It maintains communication channels between processes which are represented by instances of nas.java.net.connection. ConnectionManager.Connection. The class Connection captures the following information:

(1) Endpoints - each connection is identified by two endpoints which represent server and client sockets. To capture endpoints, each connection stores the references of the corresponding SUT socket objects and the respective ApplicationContext instances to identify processes.

(2) Buffers - to capture the communication data, the connection has two buffers, one buffer stores data sent from the client socket to the server socket, and the other one stores data sent from the server socket to the client socket. The buffers are represented by dynamically growing cyclic queues which store raw bytes.



Figure 2: The list of connections is kept at the host JVM level.

(3) State - every connection can be in four possible states, *pend-ing* (the connection has not been established, and one endpoint is waiting for the other end to connect), *established* (the connection is established and is ready for I/O operations), *closed* (at least one socket has been closed, and no more data can be transmitted), and *terminated* (at least one of the connection socket objects has been garbage collected without getting closed).

The connection manager maintains a list of connections that were created along the current execution path. As Figure 2 shows, the list of connections is maintained at the host JVM level and therefore, invisible from the SUT. Any communication between processes requires accessing and (possibly) updating a connection object which is performed through our native peers. For example, to send some data, the sender process writes the data in a connection object, and the receiver process reads it from there.

Since connections do not exist at the SUT level, JPF does not include their states when restoring the SUT state. Therefore when JPF backtracks, the state of the connections may not be in sync with the SUT. To address this issue, jpf-nas uses a gov.nasa.jpf. util.StateExtensionListener which has a map from state id to list of connections. Each time JPF reaches a new state, it creates a deep copy of the current list of connections and inserts it to the map. When JPF returns back to a previously visited state, given the state id, it restores the connection list from the map.

5.2 Scheduler

To capture nondeterministic choices during SUT execution, JPF uses gov.nasa.jpf.vm.ChoiceGenerator objects that correspond to state space branches such as scheduling points. For distributed systems, there are two types of scheduling points: process *local* and system *global*. The first one considers only runnable threads within the current process as scheduling candidates, whereas global scheduling points consider all runnable threads regardless of process association.

A naive way to explore the state space of a distributed system is to use only global scheduling points for all operations in which different schedules might result in different program behavior. In a distributed system, this approach would greatly aggravate the state space explosion problem. Since different processes can only observe each other at network interaction points, we restrict global scheduling points to such operations plus process termination, and use local schedules for process internal operations such as lock acquisition or shared field access. This approach ensures to capture all points in which context switches across processes can yield different results. For example, to establish a connection, a server needs to create and block on a ServerSocket object before a client sends a connection request, or otherwise the connection will fail. The corresponding ServerSocket.accept call is therefore a global scheduling point.

To distinguish between local and global scheduling types we provide the nas.java.net.choice.Scheduler class. This class extends the JPF SchedulerFactory functionality to create the ThreadChoiceGenerator objects that capture global scheduling points. In addition, our scheduler can be used to inject exceptions which correspond to failures that occur at the network layer. Such failures occur outside of the processes but they make processes throw exceptions. Therefore, exercising all possible behaviors of the processes does not allow for verifying the SUT against such failures. To capture these failures, our scheduler creates choice generator objects (encapsulated by Exception-ThreadChoiceFromSet) which include choices associated with exceptions such as java.io.IOException and java.net.Socket-TimeoutException. Such functionality is essential to simulate network failures without SUT modification. This injection is configurable and is performed by the native peer methods that model respective network APIs.

5.3 Verifying Client/Server Example

This section explains how jpf-nas verifies the client/server example in Figure 1. Applying this extension requires JPF to use the multiprocess JVM and the distributed scheduler factory (see Section 3). The default configuration of jpf-nas includes the following setting.

```
vm.scheduler_factory.class =
   gov.nasa.jpf.vm.DistributedSchedulerFactory
vm.class =
   gov.nasa.jpf.vm.MultiProcessVM
```

To apply jpf-nas on the client/server example, the following configuration is used.

```
@using = jpf-nas
target.0 = Server
target.1 = Client
listener +=
,gov.nasa.jpf.listener.DistributedSimpleDot
```



Figure 3: The search graph of the example in Figure 1. T0 and T1 represent the server and client main threads respectively.

The first line makes JPF to use the jpf-nas extension. The next two lines specify the initial classes from the main methods of which the execution of the server and client processes start. Finally, the last line makes JPF use our listener DistributedSimpleDot. This listener extends the SimpleDot listener of JPF and generates a dot file including the search graph of the SUT. It distinguishes between local and global scheduling points by using different shapes (local and global scheduling points are represented by circles and polygons respectively). Figure 3 shows the automatically generated search graph from running jpf-nas on the example. The states S and 6 represent the initial state and the end state respectively. The state labeled with e1 is an error state. Transitions labeled by $T\theta$ are taken by the server process and the ones labeled by T1 are taken by the client process. Moreover, the graph shows the transitions last statements that result in generating new states.

As was mentioned earlier, different ordering of statements involved in establishing a connection between a server and a client can have different outcomes. Figure 3 shows how our extension captures the different orderings of s.6 and c.5. In the start state, S, both the client and the server main threads are enabled. The execution path that starts with the client execution leads to an error state, e1. In this path, since the client sends a connection request without the server waiting, an exception is thrown. The other possible execution starts with the server which blocks until it receives a connection request. To capture this execution, after the server blocks, the state 0 is created from which only the client can proceed. After the client unblocks the waiting server by executing c.5, jpf-nas creates the state 1 from which both the client and the server main threads can proceed next. At this state, the connection is established, and a new connection object is added to the connection manager.

Figure 3 also presents possible orderings of the I/O operations, s.8 and c.7, captured by jpf-nas. After the connection is established at the state 1, the server proceeds and executes the read operation, s.8. Since the client has not written any data yet, the server blocks and a new state (the state 2) is created from which only the client is enabled. Next the client proceeds and unblocks the server by writing a request (c.7). At this point the state 3 is created from which both processes can continue.

Closing a socket from a process can affect the I/O operations performed by the process at the other end. To capture this effect, jpf-nas creates scheduling choices upon socket close operations. That can be seen in Figure 3. The server can proceed from the state 3 until it gets to the close operation. Before closing the corresponding connection in the connection manager, jpf-nas breaks the transition and creates the state 4 from which both the client and the server main threads are enabled. The execution followed by the server from the state 4 allows the server to close its socket before the client performs any further operations. In this simple example the client does not perform any I/O operations after this point. However, if it would, generating a global scheduling point at the state 4 could capture the effect of the client accessing a closed socket. The other possible execution which is followed by the client from the state 4 allows the client to proceed before the server socket is closed.

To show the effect of the failure injection functionality, we ran jpf-nas on the same example with the property scheduler. failure_injection set to true. This setting allows for checking the code against possible network failures. Figure 4 represents the resulting search graph including injected failures. As an example, consider the state 0 at which the server is waiting (s.6) to receive a connection request from a client. Due to a failure at the network layer, the server can throw a java.io.IOException object. To capture such a failure, jpf-nas includes a transition which leads out of the state 0 and corresponds to the server failure. Since our example does not handle the exception properly, the execution ends up in the error state e5.

6. CONCLUSIONS

In this paper, we presented our work to model check distributed Java applications by implementing an extension of JPF, *jpfnas*. This extension models **java.net** inter-process communication APIs based on a centralization approach, and optimizes state space branches with a specialized, process aware scheduler.

As future work we plan to extend our model to support nonblocking IPC, higher level communication mechanisms such as remote method invocation (RMI), and abstraction of external processes by means of scripts. Our final goal is to apply jpf-nas to real world distributed Java applications.

Acknowledgment

The authors would like to thank Franck van Breugel for his comments, and the reviewers for their constructive feedback.



Figure 4: The search graph of the example in Figure 1 including injected failures at the network layer.

7. REFERENCES

- [1] Esmond Pitt. Fundamental Networking in Java. Springer-Verlag, 2010.
- [2] Scott D. Stoller and Yanhong A. Liu. Transformations for model checking distributed Java programs. In SPIN, 2001.
- [3] Cyrille Artho and Pierre-Loic Garoche. Accurate centralization for applying model checking on networked applications. In ASE, 2006.
- [4] Yoshihito Nakagawa, Richard Potter, Mitsuharu Yamamoto, Masami Hagiya, and Kazuhiko Kato. Model checking of multi-process applications using SBUML and GDB. In Proceedings of Workshop on Dependable Software: Tools and Methods, 2005.
- [5] Elliot Barlas and Tevfik Bultan. Netstub: a framework for verification of distributed Java applications. In ASE, 2007.
- [6] Nastaran Shafiei and Peter C. Mehlitz. Modeling class loaders in Java PathFinder version 7. ACM SIGSOFT Software Engineering Notes, 37(6):1–5, November 2012.