# NetStub: A Framework for Verification of Distributed Java Applications*

Elliot D. Barlas and Tevfik Bultan
Computer Science Department
University of California
Santa Barbara, CA 93106, USA
{ebarlas,bultan}@cs.ucsb.edu

## ABSTRACT

Automated verification of distributed programs is a challenging problem. Since the behavior of a distributed program encompasses the behavior of the network, possible configurations of the network have to be investigated during verification. This leads to very large state spaces, and automated verification becomes infeasible. We present a framework that addresses this problem by decoupling the behavior of distributed programs from the behavior of the network. Our framework is based on a set of stub classes that replace native methods used in network communication and enables verification of distributed Java applications by isolating their behavior from the network. The framework supports two modes of verification: unit verification and integration verification. Integration verification checks multiple interacting distributed application components by running them in a single JVM and simulating the behavior of the network within the same JVM via stub classes. Unit verification targets a single component of a distributed application and requires that the user write an event generator class that utilizes the API exported by the framework. While unit verification only checks a single application component, it benefits from a greatly reduced state space compared do that of integration verification.

## 1. INTRODUCTION

In recent years, model checking software has become an active area of research [6, 4, 3, 11, 12, 5, 8]. Model checking is an attractive alternative to software testing since it provides a way to systematically explore the state space of a program, and produces a counter-example trace in case a bug is detected.

Software model checkers can be divided into two categories: symbolic model checkers (such as [3, 5, 8]) and explicit state model checkers (such as [6, 4, 11, 12]). Symbolic model checkers encode a program's states symbolically (for example as Boolean predicates as in [3]) which typically requires an abstraction that maps the states of the program to symbolic states. Explicit-state model checkers, on the other hand, explicitly store the states of the program and do not require an extra abstraction step. An explicit-state model checker systematically explores the state space of a program by visiting the program states based on a traversal algorithm such as depth-first or breadth-first search.

In order to develop an explicit-state model checker, one has to decide how to capture the program state. Since computer systems themselves are built using layers of abstractions there is no obvious answer to this question. For example, capturing the program state at the hardware level by recording the configurations of the processor and memory would not be suitable. On one hand, such a representation will contain too much information by recording the states of the operating system and all the processes running on the same system in addition to the state of the application that is being verified. On the other hand, such a representation will contain too little information if we are interested in verification of a distributed application that involves interaction among multiple computers through the network.

In this paper, we focus on the problem of capturing the state space of a distributed Java application for software model checking. Our framework is based on a set of stub classes written in Java that simulate the behavior of the network and enable verification of a distributed Java application using a single Java Virtual Machine (JVM). Hence, using our approach, it is possible to verify a distributed Java application by only recording the configurations of a single JVM. In this paper, we demonstrate the use of our framework with the Java PathFinder model checker.

Java PathFinder (JPF) [4] is an explicit-state model checker for verification of Java applications. JPF captures the program state at the JVM level by keeping track of the JVM configurations using its own JVM implementation. This way JPF is able to check behavior of a Java application for all possible thread interleavings. However, since JPF only keeps track of the JVM configurations, it is not capable of checking Java programs that use native methods. This means that JPF is unable to verify distributed Java applications since network communication requires invocation of native methods that move the program execution outside the scope of a JVM.

Capturing the program state at the JVM level by excluding native methods represents a compromise between accurate state depiction and scalability. As the state represen-

*This work is supported by NSF grants CCF-0614002 and CCF-0341365.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*ASE'07,* November 4–9, 2007, Atlanta, Georgia, USA.
Copyright 2007 ACM 978-1-59593-882-4/07/0011 ...$5.00.

tation moves downward closer to the hardware level, it becomes more accurate, but complexity increases and the state space explodes due to the extra information that the model checker must store, even though this extra information is not always relevant to the program behavior.

In the absence of native method support by a model checker, one must implement environments that model the behaviors of native methods. This is necessary particularly for verifying distributed applications, since they are built on top of a network stack implemented in the operating system using native code. Modeling a network need not, and should not, encompass all of the complexities of a real network, for they are not relevant to most applications and needlessly increase the state space. Instead, it is sufficient to implement a network model as a highly simplistic program module which does not consider packet formats, routers, routing paths, fragmentation, reordering, duplicates, or the multitude of other complexities and artifacts seen in the Internet. Such a simplification enables us to focus on the verification of the target distributed application rather than possible errors in the implementation of the network operations at the operating system level. This type of modularization is necessary for the efficiency of software model checking. It is possible to model the behavior of the network using a moderate amount of information that is much less than the information required to keep track of a realistic network environment.

In this paper, we present a framework to facilitate the verification of distributed Java applications using JPF, including those that use non-blocking I/O. The framework is based on a set of stub classes that replace native methods used in network communication. Our framework supports two modes of verification: unit verification and integration verification. Integration verification checks multiple interacting distributed application components together in a single JVM. Unit verification targets a single component of a distributed application and requires that the user write an event generator class that utilizes the API exported by our framework. Unit verification leads to more efficient verification by focusing on a single application component and, hence, reducing the state space that has to be explored by the model checker. We demonstrate the use of our framework by applying it to verification of a simple echo application and a large peer-to-peer overlay network system called Pastry. Since high-level networking constructs such as RMI, RPC, and web services are built on top of Java's networking primitives that we target here, we feel that the framework lends itself nicely to extensions that support these alternative communication mechanisms.

The rest of the paper is organized as follows. In Section 2 we discuss some of the related work. In Section 3 we give an overview of the NetStub framework. in Section 4 we discuss the replacement packages provided by the NetStub framework and in Section 5 we discuss the network representation used by these replacement packages. In Section 6 we briefly discuss the different verification strategies used by the NetStub framework. In Section 7 we discuss our experiments on a simple application and in Section 8 we discuss our experiments on a large peer-to-peer networking application. We conclude the paper in Section 9.

## 2. RELATED WORK

Artho and Garoche [1] present a centralization approach for verification of distributed Java programs using JPF. They focus on the challenges of wrapping applications as threads and running them in the same JVM, and apply the concepts to simple distributed Java applications. Their approach uses bytecode instrumentation to stub out native methods in the Java network libraries. They disclose few details about the performance of their system aside from JPF runtimes, which makes comparisons difficult. Their automated approach contrasts with our framework which is driven manually by the user. In a follow-up paper, Artho, Sommer, and Honiden [2] develop a fault model for model checking networked Java applications. The model generates exceptions non-deterministically for Java networking methods. This approach is automated, employing bytecode instrumentation techniques. We believe that a fully automated approach is unlikely to scale to verification of large applications. The NetStub framework supports a spectrum of verification approaches which can be used to improve the efficiency of verification by sacrificing the level of automation.

Musuvathi, Park, et al [11] present CMC, a C and C++ model checker. They discuss the need for a test environment in the context of network protocols, suggesting a replacement, or stub, approach analogous to the NetStub replacement packages. They utilize the stubbing approach in checking several AODV protocol implementations, finding numerous bugs in each. One of the benefits of the NetStub framework is it is not tied to a single verification tool. NetStub can be used by different verification or testing tools that focus on individual Java applications, in order to extend their applicability to distributed Java applications.

There have been previous work on verification of synchronization behavior in distributed systems that are based on analysis of design or architectural specifications (such as [10] and [9]). Our focus in this paper is verification of Java programs, not UML design models or architectural specifications. Moreover, our framework can be used to check arbitrary assertions in addition to synchronization behavior.

Environment generation problem is a critical problem in model checking and it has been studied before [7, 14]. In this paper we focus on the environment generation problem in the context of network communication. Using a domain specific approach enables us to provide a faithful environment for distributed applications. Our approach can be combined with other environment generation techniques to handle interactions that do not involve network communication.

## 3. NETSTUB FRAMEWORK

The NetStub framework components are illustrated in Figure 1. The framework consists of several components in addition to the application under test. The components in the NetStub framework can be divided to three categories: 1) The code provided by the NetStub framework that is used as is without any modification. 2) The application code that is being verified. 3) The code that the user has to write in order to do verification with the NetStub framework.

The components that are provided by the NetStub framework and do not require any modification are:

- **NetStub** is the component that models the network. It is responsible for maintaining the data buffers and connection queues associated with sockets. It is a foundation class that provides an API to facilitate the replacement of native methods. It also has an event

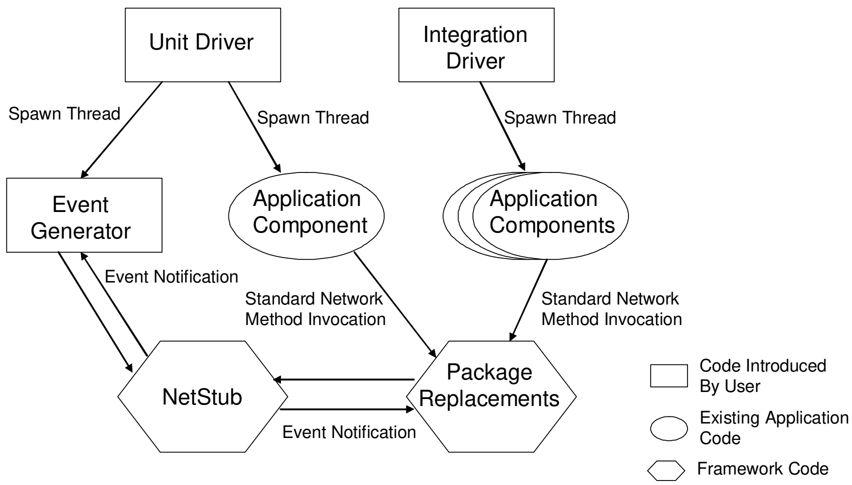


**Figure 1: NetStub framework**

notification mechanism by which listeners are notified of various network events.

- **Replacement Packages** contain replacements for Java networking packages and the classes therein. The classes are based directly on their standard Java library counterparts except they do not contain any native method invocations. The native methods in the standard Java classes are replaced with calls to the `NetStub` in the replacement classes. At verification time, the replacement packages are used by the application components instead of the standard Java packages. This can be accomplished by replacing all occurrences of the standard Java networking package names with the names of the replacement packages.

In order to use the NetStub framework, the user is responsible for writing one or two components, depending on the desired mode of verification. For integration verification, the user must implement an integration driver. For unit verification, the user must implement both a unit driver and an event generator.

- **Integration Driver** is used during integration verification. Integration driver is responsible for starting each application component under test in its own thread. Ordering constraints, such as "the client must start after the server enters the listening state," must be established by the integration driver.

- **Event Generator** is a component used during unit verification. Its sole purpose is to generate network events for the target application component to process. It can use the `NetStub` class API directly to do so, thereby avoiding the overhead incurred by using the high-level Java networking replacement classes. An event generator can be implemented in a thread-based manner or it can be entirely event-driven, responding to event notifications from the NetStub class.

- **Unit Driver** is analogous to the integration driver, but is used during unit verification. It spawns the target application component and the event generator component in their own threads. It must establish any ordering constraints among these components.

The integration and unit drivers are small modules that are easy to write. The complexity of the event generator depends on the application. In the following sections we will discuss all the components used in the NetStub framework in detail.

## 4. REPLACEMENT PACKAGES

Although native methods are confined to relatively few classes in the Java networking packages, in our framework, we provide stub classes that replace the contents of entire Java packages. The main reason for this is networking classes are highly interdependent. Writing a stub for a widely-used class with native methods, such as `InetAddress`, requires a stub to be written for all classes that depend on `InetAddress`, all classes that depend on classes that depend on `InetAddress`, and so on. For this reason, it is more practical to simply replace all classes within a given networking package even though many may require no changes other than package membership. Moreover, creating entire package replacements makes verification setup easier. Users of our framework can switch to the stub classes by making only simple import statement modifications to the application under test.

The replacement packages replace all core, public, network-related classes in the `java.net` and `java.nio` packages, and the packages therein. The replacement packages use the prefix `netstub` instead of `java`. The following describes the challenges and replacement strategies for the various packages:

**java.net replacement.**

This is the core networking package in Java. It contains three socket abstractions: `Socket`, `ServerSocket`, and `DatagramSocket`. A `Socket` is an active TCP communication endpoint capable of sending and receiving data through a stream-oriented connection. A `ServerSocket` is a passive TCP communication endpoint capable of accepting incoming connections and creating Sockets for them. A DatagramSocket is an active UDP communication endpoint capable of sending and receiving data packets. Each of the three socket classes implements error-checking, synchronization, and high-level socket state management while leaving the low-level socket management to an implementation class. Socket and ServerSocket are built on top of a SocketImpl instance variable. SocketImpl is an abstract class specifying low-level methods relevant to a Socket and ServerSocket such as accept, connect, getInputStream, getOutputStream, and listen. Similarly, the DatagramSocket class is built on top of a DatagramSocketImpl instance variable. The *impl* class that backs a socket is not accessible to the user. Concrete implementations of SocketImpl and DatagramSocketImpl in the java.net package have package-level access. This is where the bulk of native methods in the java.net package occur. Custom concrete implementations of SocketImpl and DatagramSocketImpl were written for the netstub.net package. These classes are built on top of the NetStub foundation class, discussed in the following section. Most calls to an impl class are simply forwarded to the NetStub class.

The public replacement classes have identical interfaces to those in the java.net package, with the exception of several classes with extended interfaces in the netstub.net package. The socket and impl classes have extended interfaces to support non-blocking operations. The non-blocking operations are utilized by replacement classes in the netstub.nio.channels package. All blocking operations available in Socket, ServerSocket, and DatagramSocket are available in a non-blocking form. The following non-blocking methods are part of the netstub.net socket API: attemptAccept, startConnect, finishConnect, attemptSend, and attemptReceive. In addition, non-blocking extensions to InputStream and OutputStream are available with attemptRead and attemptWrite non-blocking methods. The socket classes also have a public instance method called setChannel for setting the channel associated with a socket if one exists. The setChannel method passes a channel reference down to the underlying socket implementation instance. When the socket implementation is notified of an event relevant to the current socket by the NetStub class, it can wakeup the selector that the channel is registered with. The netstub.nio.channels package is discussed in more detail below. InetAddress also has an extended interface to allow the binding of an application component to a host name and IP address using thread groups. This extension to InetAddress is discussed in the Integration Verification and Unit Verification sections.

**java.nio replacement.**

This is the non-blocking I/O base package. It contains the Buffer abstract base class as well as abstract Buffer implementations such as ByteBuffer, IntBuffer, and CharBuffer. A buffer is a finite sequence of elements of a specific primitive type. Buffers are used by the channel classes specified in the java.nio.channels package. They are instantiated via a static allocate method of an abstract Buffer implementation. The most commonly used abstract Buffer implementation is ByteBuffer, since it is a generic buffer capable of converting other primitive data types to a sequence of bytes via calls such as putChar, putDouble, and putInt. ByteBuffer is the only Buffer subclass that is in the netstub.nio replacement package since it can effectively store any primitive data type. With the decomposition of other primitive types into bytes, it is necessary to know the system byte order. ByteOrder is a class in the java.nio package for precisely this purpose. The static nativeOrder method of the ByteOrder class makes native system calls to determine the byte order. In the replacement package, the ByteOrder class is abandoned and big endian is always used.

**java.nio.channels replacement.**

This package contains the classes for performing non-blocking network operations. It contains abstract channel classes for the three socket abstractions. The channel classes are SocketChannel, ServerSocketChannel, and DatagramChannel. The channels can be configured to be non-blocking. Channels can be registered with a Selector. A selector's select method is a blocking call that returns when channel operations are available, such as a read from a SocketChannel, much like the select UNIX system call in C. A single thread can process many channels using a Selector. Concrete implementations of SocketChannel, ServerSocketChannel, DatagramChannel, and Selector are obtained via public static open methods. Concrete implementations do not accompany the package but rather platform specific implementations are obtained internally with native method invocations. This made replacement more challenging since Java source implementations were not available. The channel replacement classes in netstub.nio.channels are built on top of the netstub.net socket abstractions. SocketChannel is built on top of Socket, ServerSocketChannel on ServerSocket, and DatagramChannel on DatagramSocket. This made for a clean implementation since non-blocking functionality was available in the extended socket classes in the netstub.net package, as explained above. A socket channel passes a self-reference to the underlying socket via the setChannel method. This way, the socket can notify the selector associated with the channel when a relevant network event occurs, waking a thread waiting on the selector's select method.

**java.nio.channels.spi replacement.**

This package is a small extension to the java.nio.channels package containing abstract channel subclasses of the abstract base classes in the parent package. Virtually no changes occurred in this package. The netstub equivalent contains the same set of public classes with few changes.

## 5. NETSTUB CLASS

The replacement packages are built on top of a singleton class called NetStub. This class provides an abstraction for all behavior below the application layer and it encapsulates all network activity. It simulates an ideal network in which no packets are lost, reordered, or corrupted. Note that in a real network implementation such behavior is handled by the TCP/IP protocol suite. By providing an idealized network, NetStub class decouples the behavior of a distributed Java application from the behavior of the network level protocols such as TCP/IP. Extending the framework to model imperfect network scenarios, similar to that proposed in [2], is left as future work.

Figure 2 shows a high level overview of our approach. A distributed Java application consists of multiple application

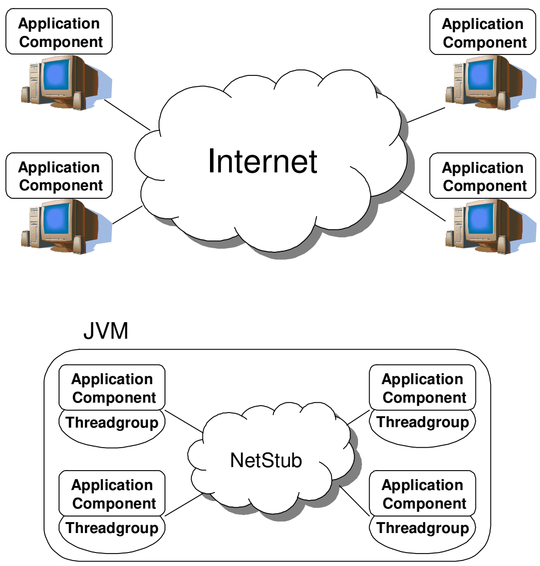


**Figure 2: NetStub Approach**

components executing in different computers. These components interact with each other through the Internet using network communication. Using the `NetStub` class we are able to replace the entire network with a set of buffers that keep track of the current state of the network traffic. This allows us to execute the distributed Java application within a single JVM without executing any native methods. Each application component is mapped to a different thread group in order to manage the host addressing as we will discuss in the following section. Note that, by encapsulating the behavior of the whole distributed Java application in a single JVM, we are able to use a model checker such as JPF for verification of a distributed Java application. Below we discuss some of the important features of the `NetStub` class.

The `NetStub` class provides a rich interface consisting of dozens of methods for performing blocking and non-blocking network operations. The interface also contains methods for registering event listeners. Registered event listeners are notified when various types of events occur. The five classes of listeners are `ByteStreamListener`, `PacketListener`, `ConnectionListener`, `ServerListener`, and `DatagramSocketBindListener` (Figure 3 shows the listener classes and the methods that are relevant to a `ServerListener` as an example). Classes in the replacement packages use this API instead of invoking native methods. The event notification mechanism enables an efficient implementation of the non-blocking networking classes. The `SocketImpl` and `DatagramSocketImpl` concrete subclasses in the `netstub.net` package register themselves as listeners of various events with the `NetStub`. When an event occurs, such as the receipt of data, the implementation class is notified by the `NetStub` and can subsequently notify the selector associated with the corresponding channel if one has been set. The event notification mechanism is also a helpful tool in implementing both drivers and event generators. A verification driver may spawn one or more clients upon receiving notification that a server has entered the listening state. An event-driven event generator may read a sequence of bytes from a connection upon receiving notification that the destination sent a message.

```
new Thread(new ThreadGroup("server"), new Runnable() {
  public void run() {
    InetAddress.register("server.cs.ucsb.edu");
    // start server ...
  }
}).start();
```

**Figure 4: Host Addressing Code Example**

```
Socket socket = new Socket(servAddr, servPort);
socket.getOutputStream().write(data);

for(int i=0; i<data.length; i++) {
  byte b = (byte)socket.getInputStream().read();
  assert(data[i] == b);
}

socket.close();
```

**Figure 5: Client for Echo Server**

The `NetStub` class maintains different state for a `Socket`, `ServerSocket`, and `DatagramSocket`. For a `Socket`, the `NetStub` maintains two queues: an incoming byte stream queue and an outgoing byte stream queue. The other `Socket` in the connection has references to the same queues, but for the opposite purpose. The input queue of one `Socket` is the output queue of the other and vice versa. When one `Socket` issues a send, bytes are placed on the back of its output queue and the destination's input queue. The queues are bounded to allow situations where a `write` is attempted and fewer than the requested number of bytes are written, which is a case that is rarely checked in applications and can cause elusive bugs. Note that this behavior does not resemble a dropped packet, but rather a full queue on the sending host. For a `ServerSocket`, the `NetStub` maintains an accept queue for pending connections. When a `Socket` issues a connect to a `ServerSocket`, a connection request is placed on the back of the `ServerSocket`'s accept queue and the issuing `Socket` enters a wait state. When the `ServerSocket` accepts the connection request, `Socket` state is initialized for both parties and the issuing `Socket` is awoken. For a `DatagramSocket`, the `NetStub` maintains only an input queue of bytes. All sent packets arrive to the same input queue since there is no notion of a connection.

The `NetStub` class does full synchronization with a lock member object. This not only guarantees thread safety among the possibly many application component threads using it, but it also resembles a discrete network event model that greatly reduces the state space by limiting the number of thread interleavings. In addition, the lock is available for use by other classes via the `getLock` instance method. This allows for the use of a single, shared lock among network classes that require synchronization. This prevents nested lock acquisitions, which can cause deadlocks in methods that block, and reduces the state space further by limiting the number of thread interleavings outside of the `NetStub` class.

# 6. VERIFICATION USING NETSTUB

As mentioned above, our framework supports both integration and unit verification. Both integration verification and unit verification drivers spawn the application components under test in their own threads. Note that, application

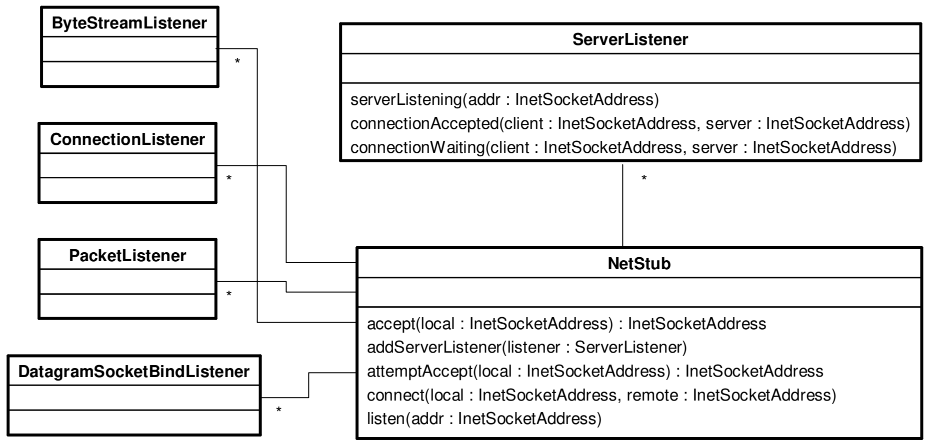


**Figure 3: NetStub and Listener Classes**

```
ServerSocket serverSocket = new ServerSocket(port);

for(int i=0; i<maxConnections; i++) {
  Socket socket = serverSocket.accept();

  while(true) {
    int b = socket.getInputStream().read();
    if(b < 0) break;
    socket.getOutputStream().write(b);
  }

  socket.close();
}

serverSocket.close();
```

**Figure 6: Echo Server**

components should appear to be running on unique hosts, despite sharing a single JVM. The `getLocalHost` static method of the `InetAddress` class should return different addresses for different application components. Similarly, the loopback address of an application component should point to itself and not to other application components. The NetStub framework achieves this isolation by associating applications with thread groups.

Application components are spawned in threads with unique thread groups. When an application component calls `getLocalHost`, `InetAddress` looks into a table mapping thread groups to addresses and retrieves the `InetAddress` corresponding to the thread group of the current thread. The map entry for an application component can be set explicitly inside of a thread via the `registerThreadGroup` static methods, as shown in Figure 4, or it can be generated lazily by `InetAddress` upon the first address query. The `registerThreadGroup` methods can be used to register a host name, IP address, or both with the thread group of the current thread. Java has a built-in thread group facility. If no thread group is provided to the thread constructor, Java puts the new thread in the parent's thread group. As long as the application components being verified do not themselves employ thread groups, all threads in a given application component will belong to the same thread group and therefore will be on the same logical host. Figure 2 illustrates addressing in the NetStub framework where each application component in the distributed application is mapped to a separate thread group in a single JVM.

When using the NetStub framework for *integration verification* several application components are executed in a single JVM. In order to use a model checker such as JPF, all native code must be eliminated from each application component. Native code that is used in network packages is eliminated by replacing the standard Java networking packages with NetStub replacement packages. Integration verification requires that the user write an integration driver that spawns each application component in its own thread. Each application component thread should be instantiated with a unique thread group to ensure that each application component runs on its own logical host. Running multiple application components in separate threads generates a massive state space even for very small programs. Unit verification offers an alternative to running multiple high-level application components.

*Unit verification* targets a single application component. As with integration verification, native methods present in the standard Java networking code is eliminated by using the NetStub replacement packages. Unit verification requires that the user write a unit driver and an event generator. The unit driver is analogous to the integration driver. It spawns the application component under test and the event generator in their own threads. The application component thread should be instantiated with a unique thread group to ensure it runs on its own logical host. An event generator is a module engineered specifically for testing the application component. It is optional for an event generator to belong to a unique thread group. If the event generator doesn't invoke `getLocalHost` or make use of the loopback address, it need not worry about which thread group it belongs to. An event generator can utilize the NetStub API instead of the high-level socket API, allowing it to avoid a considerable number of instructions and variables. This property leads unit verification to perform better than integration verification.

Event generators can be thread-based or event-driven. A thread-based event generator executes as a separate thread and interacts with the target application component using blocking or non-blocking network communication primitives provided by the NetStub API. Note that, a thread-based event generator executes concurrently with the target application component. Possible interleavings of the event generator and the target application component increase the state space, and, hence, the cost of verification. Event-driven event generation is an efficient alternative to thread-based event generation.

Event-driven event generators utilize the event notification mechanism offered by the `NetStub` class. Rather than performing network operations in a separate thread, event-driven event generators react to network events initiated by the target application component. Even for an event-driven event generator a separate (short-lived) thread is necessary to do initialization operations, but all subsequent code is executed in event handlers, by the target application thread. For example, if a server application component sends a message, the server thread will execute the `byteSent` event handler of all `ByteStreamListeners` from within the `NetStub`'s `send` method. This can greatly reduce concurrency, and state space, since event notifications are made in the same thread that initiated the operation. However, it also imposes some restrictions on what can be called from within an event handler. One must avoid making calls that block until being acknowledged in some way by the application component, otherwise deadlock will ensue. Event-based event generators offer the highest degree of performance through reduced concurrency while increasing the difficulty of the event generator implementation.

# 7. A SIMPLE EXAMPLE: ECHO SERVER

In this section we will demonstrate the use of the NetStub framework using an example distributed application. The echo application is a simple distributed application composed of two application components, a client and a server. The server accepts a fixed number of connections, and for each, echoes every byte received back to the client until the connection is closed by the client. The code for the server is shown in Figure 6. The client connects to a server, sends a fixed number of bytes, receives the same number of bytes, and then closes the connection. The client code is shown in Figure 5. Despite the brevity and triviality of the application, it cannot be verified by the JPF model checker due to the native methods within Java's standard networking classes. The `NetStub` framework provides two approaches to verifying the components of this application. The simplest is via integration verification, and a slightly more involved approach to verifying the components individually is unit verification.

Integration verification requires that the user write a driver that spawns the client and server in their own threads, with separate thread groups. If the driver does not take any measures to prevent the client from starting before the server has entered the listening state, the client will throw an exception when it tries to connect to the server. Soon thereafter, JPF will generate a spurious deadlock error since the server will be blocking indefinitely on the accept method invocation. This dependency between client and server can be established with the event notification mechanism provided by the `NetStub` class. The driver can register a server listener that is notified when the server enters the listening state. The methods relevant to a `ServerListener` are shown in the abbreviated class diagram in Figure 3. Upon notification, the driver can start the client in its own thread and thread group, thereby establishing the dependency. The code depicting the integration driver for the echo application is shown in Figure 7.

```
netStub.addServerListener(new ServerListener() {
  public void connectionAccepted(...) {}
  public void connectionWaiting(...) {}
  public void serverListening(InetSocketAddress addr) {
    for(int i=0; i<numClients; i++) {
      String grp = "client" + i;
      Client client = new Client(data, addr);
      Thread t = new Thread(new ThreadGroup(grp), client);
      t.start();
    }
  }
});

new Thread(new ThreadGroup("server"), new Runnable() {
  public void run() {
    InetAddress.register("server.cs.ucsb.edu");
    new Server(25000, numClients).run();
  }
}).start();
```

**Figure 7: Integration Driver**

```
netstub.connect(clientAddr, serverAddr);
netstub.getOutputStream(clientAddr, serverAddr).write(data);

for(int i=0; i<data.length; i++) {
  byte b = (byte)netstub.getInputStream(clientAddr,
                                        serverAddr).read();
  assert(data[i] == b);
}

netstub.closeConnection(clientAddr, serverAddr)
```

**Figure 8: Thread-based Event Generator**

Unit verification for the echo server requires both an event generator and a driver. The driver also must satisfy the constraints mentioned above to ensure proper ordering. As discussed above there are two types of event generators for unit verification: thread-based and event-driven. The thread-based event generator mimicks the echo client, except calls are made directly to the `NetStub` instead of the high-level socket interface. It is a simple module that does not use the event mechanism offered by the `NetStub`. Figure 8 shows the code for the thread-based event generator for testing the echo server.

The event-driven event generator reacts to event notifications from the `NetStub`. Only the connection to the server is implemented in the event generator thread and all other operations are implemented in two event handlers: `connectionAccepted` and `byteSent`. The `connectionAccepted` event handler is invoked by the `Net-Stub` when a server accepts a connection. The event-driven event generator sends the bytes to be echoed upon receiving this notification. This interaction is depicted in a sequence diagram in Figure 9. It demonstrates that the event handler method is invoked from the server application component thread. The `byteSent` event handler is invoked by the `NetStub` when a byte is sent over a connection. The event generator reads a byte and asserts that it is correct upon receiving this notification. The event-driven event generator creates the exact same sequence of network events as the high-level client, but

**Table 1: Echo Application Results**

| Connections | Time (MM:SS) | New States | Max Depth | Memory (MB) | Mode |
|---|---|---|---|---|---|
| 1 | 21:09 | 625528 | 912 | 34 | Integration |
| 1 | 02:19 | 55537 | 587 | 13 | Thread-Based |
| 1 | 00:46 | 5989 | 581 | 12 | Event-Based |

**Table 2: Buggy Echo Application Result**

| Connections | Time (MM:SS) | New States | Max Depth | Memory (MB) | Bug Found | Mode |
|---|---|---|---|---|---|---|
| 1 | 01:04 | 29744 | 550 | 13 | Yes | Integration |
| 2 | 02:45 | 54833 | 1689 | 24 | Yes | Integration |
| 5 | 11:53 | 104257 | 6259 | 72 | Yes | Integration |
| 10 | 46:07 | 191513 | 18755 | 240 | Yes | Integration |
| 1 | 00:32 | 12368 | 425 | 12 | Yes | Thread-Based |
| 2 | 01:00 | 15915 | 956 | 18 | Yes | Thread-Based |
| 5 | 03:13 | 22327 | 2808 | 44 | Yes | Thread-Based |
| 10 | 10:13 | 33824 | 6705 | 108 | Yes | Thread-Based |
| 1 | 00:06 | 498 | 130 | 8 | Yes | Event-Based |
| 2 | 00:09 | 1126 | 613 | 16 | Yes | Event-Based |
| 5 | 00:48 | 3608 | 2528 | 43 | Yes | Event-Based |
| 10 | 03:45 | 8975 | 6950 | 125 | Yes | Event-Based |

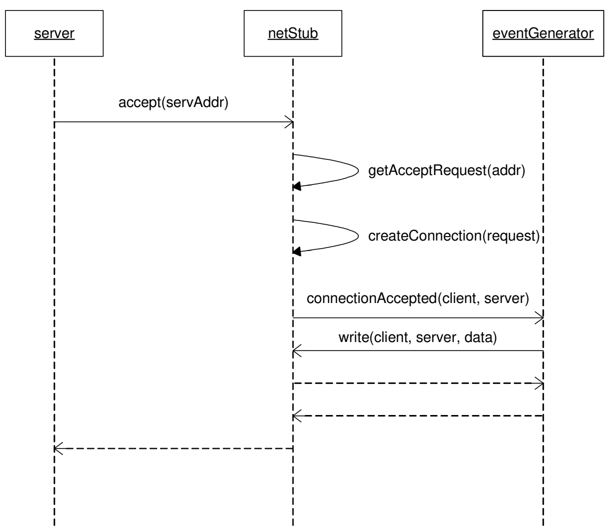


**Figure 9: Event-based Event Generator**

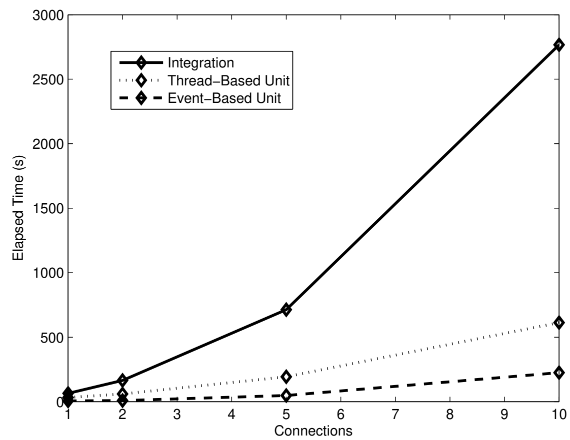


**Figure 10: Verification Comparison**

it does so in response to network events instead of executing blocking methods while concurrently running with the server. From the server's perspective, there is no difference between the original client, a thread-based event generator, and an event-driven event generator.

**Experiments with the Echo Server.**

We conducted two sets of experiments with the Echo application. The purpose of the first echo server experiment was to measure the cost of exhaustive verification with JPF using integration verification and unit verification for a single connection and ten bytes echoed on that connection. The results confirm that the state space is reduced considerably from integration verification to unit verification. Furthermore, an event-based event generator has much smaller state space than a thread-based event generator. Table 1, shows the results of the various runs. Note that integration verification for this simple application takes more than 21 minutes whereas unit verification with an event-based event generator takes less than one minute. The low memory consumption, large elapsed time, and low maximum depth indicates that the bottleneck for checking the applications was the number of thread interleavings.

In the second echo server experiment, the server application component was faced with multiple connections and JPF's `Verify` class was employed to insert a bug in the server on some execution paths. For the last accepted connection, the server uses JPF's `Verify` class to generate multiple execution paths, some of which echo a corrupted byte back to the client. Both modes of verification successfully checked the application for 10 connections. The unit verification techniques performed much better than integration verification due to a reduced state space through the use of the event generator rather than a high-level client. The results are shown in Table 2. Memory was an issue for this experiment since it included multiple connections to the server. Figure 10 shows an elapsed time comparison for the three configurations. The elapsed time increases exponentially with the number of connections for integration verification whereas the elapsed time increase is nearly linear for unit verification with an event-driven event generator.

Table 3: Buggy Pastry Application - Integration Verification Results

| Time (MM:SS) | New States | Max Depth | Memory (MB) | Bug Found |
|---|---|---|---|---|
| 15:31 | 40848 | 40847 | 893 | No |

Table 4: Buggy Pastry Application - Thread-Based, Capture-Replay Unit Verification Results

| Time (MM:SS) | New States | Max Depth | Memory (MB) | Bug Found |
|---|---|---|---|---|
| 04:38 | 39554 | 39553 | 893 | Yes |

# 8. PASTRY

Pastry [13] is a scalable, decentralized peer-to-peer overlay network. Its predecessors include systems such as Napster and Gnutella. These systems lack a scalable and decentralized method for communicating with other peers in the network. Pastry uses a technique known as key-based routing. With key-based routing, each node is associated with a unique and uniformly distributed nodeId. Given a message and a key, Pastry reliably routes to the node with the nodeId that is numerically closest to the key. Pastry can perform this service in log(N) hops, where N is the number of nodes in the network. Pastry peers maintain routing table and leaf set data structures that allow them to intelligently route a message towards a destination based on the message key.

FreePastry is an open source Pastry implementation written in Java. It is an active project at Rice University. The core system code consists of hundreds of classes. FreePastry makes complete use of Java's networking packages. It uses the blocking network operations in the `java.net` package as well as the non-blocking operations in the `java.nio` packages. FreePastry does its own message serialization. This makes it an ideal candidate for the NetStub framework because it does not use Java's built-in object serialization found in the `java.io` package, which is not supported by the NetStub framework as it is not strictly a network component and does not reside within the core networking packages. FreePastry is a massive, highly concurrent system, which is far beyond the scope of JPF. However, it serves as a good benchmark for the NetStub framework and showcases the breadth of the NetStub replacement packages.

The first step in utilizing the NetStub framework for verification of FreePastry was to replace all links to the standard Java networking packages with links to the `netstub` replacement packages. FreePastry contained a small number of other native methods in addition to the those in the networking classes, mainly related to file I/O and platform-specific properties. Most of this native code was eliminated without modifying the FreePastry system, since FreePastry allows the user to supply preferred logging, time source, and property classes. However, other native code required minor modifications to the system, including the replacement of hash functionality from the `MessageDigest` class of the `java.security` package.

**Integration Verification.**

The FreePastry application we analyzed consists of two peers. One peer starts the overlay network and the other peer joins the existing one. The initial peer generates an assertion violation when a neighbor successfully joins it. Since this assertion error occurs on every execution path, it acts as a depth gauge for JPF. The integration verification driver simply spawns the two peers in their own threads with unique thread groups. With integration verification, JPF did not reach the assertion violation. During state space exploration JPF exhausted the allocated memory bound of 900 MB, with a reported usage of 893 MB. Initially, we allocated the system's entire 1 GB memory to JPF, but this caused JPF to slow down to a halt after a short period, making no progress. Hence, we tried a 900 MB memory allocation which ran smoothly. JPF's failure to reach the inevitable assertion was not unexpected, as the volume of system code for a FreePastry Peer is quite large. JPF's state storage was the bottleneck. The results of the verification are shown in Table 3.

**Unit Verification with Capture-Replay.**

The initial peer was the target application component for unit verification. Unit verification for a large system such as FreePastry is difficult because implementing an event generator requires knowledge of all network communication within the system. This knowledge is required because the event generator must invoke the same network operations on the `NetStub` class directly. In the echo application example, this was a simple task. The event generator was based directly on the client, with all networking calls made to the `NetStub` instead of the high-level networking API. In FreePastry, network calls are initiated throughout the system and determining the sequence they are invoked can be difficult. We overcame this problem by using a thread-based, capture-replay style event generator. The goal is to record the network events during a normal execution of the distributed application and then to replay the recorded event sequence within the event generator during unit verification.

To generate a capture-replay style event generator for the FreePastry peer we recorded the sequence of calls to the `NetStub` by that peer during normal program execution, without JPF. The recorded sequence of calls provides a trace that the event generator should follow during unit verification. The `NetStub`'s event notification mechanism provides support for this recording capability. A recording class can simply register interest in all types of network events and therefore receive notification for all network operations. After capturing the sequence of network operations that take place when a peer joins another peer, the event generator can replay the same sequence of network operations using the `NetStub` API during unit verification. The capture-replay technique was employed to implement a thread-based event generator for performing unit verification on the FreePastry application. This lean event generator allowed JPF to reach the assertion violation in the FreePastry peer under test despite reaching the same memory threshold as the failed integration verification run. The JPF results are shown in Table 4. We believe that capture-replay is a useful technique for writing event generators for larger applications and for applications unfamiliar to the user of the NetStub framework.

## 9. CONCLUSION

We presented a framework for verification of distributed Java applications. The NetStub framework enables verification of distributed Java applications by running them in a single JVM. This is achieved by using the replacement packages provided by the framework which simulate the behavior of the network. The NetStub framework supports both integration and unit verification. We conducted some experiments demonstrating the use of the NetStub framework for verification of distributed Java applications using the JPF model checker. These distributed applications cannot be verified with the JPF model checker without the NetStub replacement packages.

## 10. REFERENCES


[1] C. Artho and P.-L. Garoche. Accurate centralization for applying model checking on networked applications. In *Proc. 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*, pages 177–188, 2006.

[2] C. Artho, C. Sommer, and S. Honiden. Model checking networked programs in the presence of transmission failures. In *Proc. 1st Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE 2007)*, 2007.

[3] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. 8th International SPIN Workshop*, volume 2057 of *LNCS*, pages 103–122, 2001.

[4] G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In *Proc. 15th IEEE/ACM International Conference on Automated Software Engineering (ASE 2000)*, pages 3–12, 2000.

[5] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, pages 168–176, 2004.

[6] P. Godefroid. Model checking for programming languages using verisoft. In *Proc. 24th ACM Symposium on Principles of Programming Languages (POPL 1997)*, pages 174–186, 1997.

[7] P. Godefroid, C. Colby, and L. Jagadeesan. Automatically closing open reactive programs. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1998)*, pages 345–357, 1998.

[8] T. A. Henzinger, R. Jhala, and R. Majumdar. The BLAST software verification system. In *Proc. 12th International SPIN Workshop (SPIN 2005)*, pages 25–26, 2005.

[9] P. Inverardi and M. Tivoli. Deadlock-free software architectures for com/dcom applications. *J. Syst. Softw.*, 65(3):173–183, 2003.

[10] N. Kaveh and W. Emmerich. Deadlock Detection in Distributed Object Systems. In V. Gruhn, editor, *Joint Proc. of the* $8^{th}$ *European Software Engineering Conference and the* $9^{th}$ *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 44–51. ACM Press, 2001.

[11] M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, 2002.

[12] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular software model checking framework. In *Proc. 11th ACM Symposium on Foundations of Software Engineering held jointly with the 9th European Software Engineering Conference (ESEC/FSE 2003)*, pages 267–276, 2003.

[13] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.

[14] O. Tkachuk, M. B. Dwyer, and C. Pasareanu. Automated environment generation for software model checking. In *Proceedings of the 4th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003)*, pages 116–129, 2003.