Software Model Checking of UDP-based Distributed Applications

Nazim Sebih[1], Franz Weitl[2], Cyrille Artho[3], Masami Hagiya[1],
Yoshinori Tanabe[4], and Mitsuharu Yamamoto[2]
[1] *The University of Tokyo, Tokyo, Japan*
nazimsebih@lyon.is.s.u-tokyo.ac.jp, hagiya@is.s.u-tokyo.ac.jp
[2] *Chiba University, Chiba, Japan*
franz@chiba-u.jp, mituharu@math.s.chiba-u.ac.jp
[3] *AIST/RISEC, Amagasaki, Japan*
c.artho@aist.go.jp
[4] *National Institute of Informatics, Tokyo, Japan*
y-tanabe@nii.ac.jp

***Abstract*—We extend exhaustive verification of networked applications to applications using the User Datagram Protocol (UDP). UDP maximizes performance by omitting flow control and connection handling. High-performance services often offer a UDP mode in which they handle connections internally for optimal throughput. However, because UDP is unreliable (packets are subject to loss, duplication, and reordering), verification of UDP-based applications becomes an issue. Even though unreliable behavior occurs only rarely during testing, it often appears in a production environment due to a larger number of concurrent network accesses.**

Our tool systematically tests UDP-based applications by producing packet loss, duplication, and reordering for each packet. It is built on top of `net-iocache` for the `Java PathFinder` model checker. We have evaluated the performance of our tool in a multi-threaded client/server application and detected incorrectly handled packet duplicates in a file transfer client.

***Keywords*-Software Model Checking; Java PathFinder; Testing of Distributed Systems; User Datagram Protocol; Unreliable Network IO.**

I. Introduction

Modern software often involves both multi-threading and network communication based on TCP (Transmission Control Protocol) or UDP (User Datagram Protocol). Testing such systems is complex due to non-determinism in thread scheduling and in messages transmitted across the network. Software model checking ensures that a program conforms to formal properties by exhaustive exploration of its state space, given enough memory and time.

In contrast to TCP, UDP is neither connection-oriented nor reliable: Connections between communicating peers are not established and terminated explicitly, and data packets sent by UDP may get lost, duplicated, or arrive at the destination in a different order [1].

Despite its unreliability, UDP is widely adopted for specialized applications such as real-time communication because of its lower latency and higher achievable throughput. However, it places the responsibility on the developer to ensure a sufficient level of data integrity by implementing a suitable application-level protocol. Specialized application-level protocols must be tested thoroughly since their implementation cannot be expected to have the same level of maturity as widely used implementations of TCP.

In local test environments with limited network traffic, problematic behavior such as packet loss can hardly be observed and reproduced. In test environments, UDP often behaves like TCP: All packets are received exactly once and arrive in the same order in which they have been sent. We call this the *reliable behavior* of UDP.

We consider the following cases of *unreliable behavior* of packets: 1) *loss:* a packet does not arrive at its destination, 2) *duplication:* a packet arrives more than once, and 3) *reordering:* packets arrive in a different order.

Testing a UDP-based application requires checking its behavior for both the reliable and unreliable cases of UDP input/output (I/O). Existing approaches [2], [3], [4], [5] generate unreliable UDP behaviour with a configurable stochastic distribution. However, it is hard to guarantee coverage and to reproduce rarely occurring errors by randomly generating unreliable UDP behaviour. To ensure a desired level of *coverage*, combinations of unreliable UDP behavior need to be generated systematically. For *reproducibility*, control over the outcome of UDP-based I/O is necessary.

We propose the use of software model checking for systematically executing the system under test (SUT) for the different possible outcomes of UDP I/O operations in a reproducible way (*systematic UDP simulation*). We implement our approach using the software model checker Java PathFinder [6] and its extension net-iocache [7] for distributed systems.

Considering both the reliable and (combinations of) unreliable behaviors for each UDP I/O operation leads to an exponential growth of the state space in the number of exchanged messages. To ensure scalability, we provide means for restricting the simulation of UDP behavior in two dimensions: 1) to certain kinds of unreliable behavior and 2) to certain locations of interest in the program code.

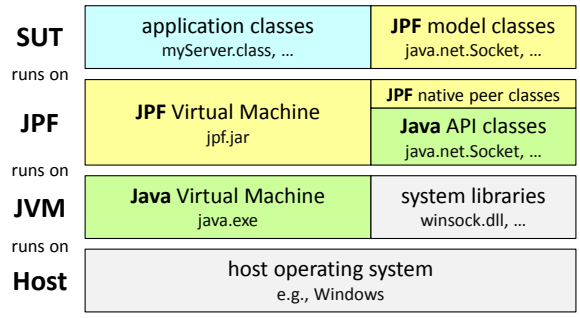

Figure 1. Levels of execution when model checking an SUT (system under test) with JPF (Java PathFinder).

Our contribution is as follows:
Method: We propose software model checking as a suitable method for testing the behavior of an SUT for possible outcomes of UDP I/O in a systematic and reproducible way.
Implementation: We add support of UDP to the JPF extension net-iocache, including the configurable simulation of UDP's unreliable behavior.
Evaluation: We compare the performance of our tool to previously implemented TCP-based benchmark scenarios and demonstrate its usefulness for finding defects in the application-level communication protocol of a client/server application for file transfer.

This paper is structured as follows. We give some background on software model checking with JPF and its extension net-iocache for networked systems, as well as UDP in Section II. Section III explains net-iocache's architecture, its redesign and extension towards UDP, and the implementation and configuration of systematic simulation of UDP behavior. We report on experimental results in Section IV and discuss related work in Section V before concluding the paper in Section VI.

II. Background

In this section, we introduce the concept of software model checking through Java PathFinder (JPF) and its extension for network applications net-iocache, and explain how the UDP protocol is supported by the Java API.

A. Software Model Checking with Java PathFinder

Our implementation extends JPF [8], [6], an explicit state software model checker for Java bytecode which explores multiple outcomes due to non-determinism such as thread interleaving and random input data.

Figure 1 shows the basic architecture of JPF. JPF is a custom Java Virtual Machine written in Java, i.e., in runs on top of a host Java Virtual Machine (JVM). The application verified by JPF is called the system under test (SUT). In contrast to a standard JVM, JPF executes the SUT for *all* outcomes of non-deterministic operations such as thread scheduling, random numbers, or certain I/O operations. To cover all combinations of non-deterministic outcomes, JPF backtracks the SUT to previous states with unexplored choices and creates an new execution branch for each choice. If JPF is able to find a state that violates a property, it shows the execution trace from the initial to the error state. The properties to be verified against the SUT can be generic properties such as data races and deadlocks, or user-defined assertions in the SUT. Because JPF targets Java bytecode, it can be applied to any language that can be compiled to bytecode (e. g., Scala, C/C++, Ruby).

JPF cannot backtrack native code. Such code may execute system calls such as I/O that have side effects on the host environment. In that case, *model classes*, which simulate the original API, need to be provided; They must be entirely written in Java. Part of our effort to support verification of UDP-based distributed applications was to implement a model class for `DatagramSocket`. JPF allows model classes to invoke methods of so called *native peer classes* that run on the host Java Virtual Machine and thus have access to the standard Java API. This way, the execution of methods with native code such as network I/O can be delegated from the JPF to the JVM execution level which in turn may interface with the host operating system to perform the operation. Figure 1 summarizes the different levels of execution.

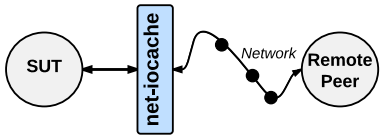

Figure 2. Tool net-iocache intercepts the communication between the SUT executed by JPF and remote peers.

B. Cache-based Model Checking of Networked Systems with net-iocache

Our approach builds on net-iocache, an extension to JPF that enables it to verify distributed systems [9], [7]. For scalability, net-iocache verifies one process at a time (the SUT), while the other processes are executed as remote peers on the host JVM (see Figure 2).

By tracking the state of all objects involved in network communication (sockets, I/O streams, network ports) and caching the result of I/O operations, net-iocache synchronizes the state of remote peers with the SUT if its state is backtracked by JPF. For model checking of UDP applications, we added support of datagram sockets to net-iocache as described in Section III.

C. UDP in Java

Java supports UDP through the `java.net` package. The following two classes provide the basic functionality:

- `DatagramPacket` contains the data and the remote destination (IP address and port) the packet will be sent to or received from.
- `DatagramSocket` handles the transmission of datagram packets. A datagram socket can be connected, restricting the exchange of datagram packets to a dedicated destination. The connection semantics is however different from TCP sockets in that no communication channel over the network is maintained.

Note that, in case a datagram packet arrives at its destination, lower-level protocols ensure that its data is unmodified.

III. UDP SUPPORT AND SIMULATION OF UNRELIABLE BEHAVIOR IN NET-IOCACHE

For software model checking UDP-based applications with systematic generation of packet loss, duplication, and reordering, we solve the following problems:

A) extension of JPF towards UDP support;
B) simulating UDP's unreliability by systematically generating combinations of packet loss, duplication, and reordering;
C) controlling the combinational complexity of UDP's unreliability simulation by restricting it 1) to locations of interest in the SUT and 2) to certain types of unreliable behavior, according to user-defined settings.

A. UDP Support in JPF

JPF does not cover package `java.net` of the Java library: When an SUT calls methods of a class such as `DatagramSocket`, JPF stops with an exception because of non-supported native methods.

`jpf-nhandler` [10] is a JPF extension that adds generic support of native method calls to JPF by delegating them to the host JVM. In the case of network sockets, a delegation-based approach does not suffice: When JPF backtracks the SUT, the states of the backtracked model-level and corresponding host-level sockets may become inconsistent, causing spurious behavior such as I/O exceptions.

`net-iocache` [7], [11] is a JPF extension that supports, in contrast to jpf-nhandler, the backtracking of network I/O. However, it did not support UDP because, originally, it has been designed and highly optimized for connection- and stream-oriented network communication between a single server and one or more client processes using TCP.

Rather than adding specific backtracking support of datagram sockets to the generic tool jpf-nhandler, we opted for extending net-iocache towards support of UDP.

1) Redesign of net-iocache: Extending net-iocache towards packet-oriented communication using UDP turned out to be difficult because, in a connection-less protocol such as UDP, it is not always possible to distinguish whether a communicating peer takes the role of a server or a client. UDP support required a redesign of net-iocache, targeted at general applicability (e. g., support of peer-to-peer communication) and easy extensibility towards new communication protocols.

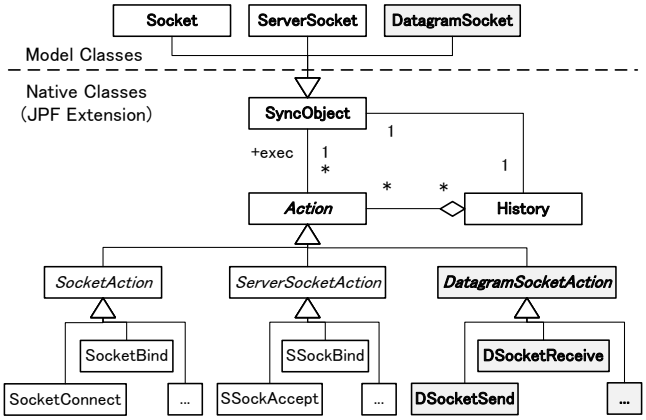

Figure 3. Conceptual model of net-iocache v2 (abstract classes in italics).

Figure 3 depicts the conceptual model of net-iocache v2, including its extension towards UDP. The upper part of Figure 3 contains JPF *model classes* for the covered Java network API which are the classes `Socket`, `ServerSocket`, and `DatagramSocket`. While model classes are executed by JPF, *native classes* are executed by the host Java Virtual Machine as part of JPF. Native class `SyncObject` (Figure 3, center) is a generic host-level representation of network model classes. Similar to `jpf-nhandler` [10], net-iocache v2 delegates method calls of model classes to host-level objects of the same type. Each such method call is represented as an object of class `Action` (Figure 3, center). For instance, a single invocation of `dsocket.send(packet)` is represented as an `Action` object, associated with sync object `dsocket`.[1]

When JPF backtracks the SUT, the state of model class objects, which are controlled by JPF, may differ from the state of their corresponding host-level objects not controlled by JPF. For synchronizing host objects with model objects, net-iocache v2 maintains a `History` of executed actions for each sync object (Figure 3, center right). This history is used for 1) detecting state mismatches between model and host objects and 2) synchronizing host objects with model objects after backtracking: A host object o_h is synchronized with the backtracked state of its corresponding model object o_m, by resetting o_h to its initial state and re-executing recorded actions on o_h until its state matches that of o_m.

2) Adding support for UDP to net-iocache: Classes `SyncObject`, `Action`, and `History` form the generic core of net-iocache v2. Since these classes are entirely abstract from the kind of executed actions and manipulated objects, the architecture is easily extensible. Adding support of UDP amounts to providing a simple model class for

[1]Sync objects are implemented as native classes for performance reasons.

`DatagramSocket` (Figure 3, top) and implementing the supported actions as subclasses of abstract class `Action` (Figure 3, bottom right). Each (non-abstract) subclass of class `Action` needs to override methods for 1) determining the set of sync objects modified by the action, 2) comparing the action with other actions (cache matching), 3) executing it natively, 4) capturing the execution results such as return values or thrown exceptions.

In the case of UDP, one model class and 8 `Action` subclasses had to be implemented with an average of 27 lines of code (lines without statements excluded). This compares to 5 model classes and 13 `Action` subclasses with an average of 23 lines of code for TCP support.

B. Systematic Simulation: Design and Implementation

Consider a UDP-based network communication where a server sends two distinct UDP packets (p, q) to a client. Each individual packet can be subject to different types of unreliability intrinsic to UDP. The following lists the packet sequences the client could possibly receive, assuming that duplication occurs at most once per packet:

()	(p)	(p,p)	(p,p,q)	(p,p,q,q)
	(q)	(q,q)	(q,q,p)	(q,q,p,p)
		(p,q)	(p,q,p)	(p,q,p,q)
		(q,p)	(q,p,q)	(q,p,q,p)
			(q,p,p)	(p,q,q,p)
			(p,q,q)	(q,p,p,q)

Our objective is to generate these non-deterministic UDP I/O outcomes in net-iocache and verify the SUT against them.

Under exhaustive simulation each packet can either be (1) lost, (2) transmitted exactly once, or (3) duplicated and transmitted twice, resulting in 3 cases per packet. This leads to a combined complexity of 3^n for n packets. While simulation of loss and duplication is a combinatorial problem, packet reordering is a permutation problem with a complexity of $n!$. Furthermore, the set of packet sequences, where the number of duplications of a single packet is not restricted, is infinite. To avoid an infinite number of outcomes and to control complexity, we set upper limits on the number of duplications one packet can experience and for the reordering window size. For exhaustive simulation within *finite bounds*, we use the term *systematic simulation*. This sets our work apart from other work [2], which uses stochastic methods to generate unreliable UDP behavior.

The modular approach for software model checking of distributed applications that net-iocache adopts, by verifying one peer at a time (the SUT), enables it to act similarly to a proxy of that SUT where it can manipulate packets coming from and going to the remote peers [7]. When the SUT calls method `send` to trigger a packet transmission to a remote peer, the physical transmission of that packet is delegated to net-iocache. Conversely, the same logic applies when the SUT executes `receive`.

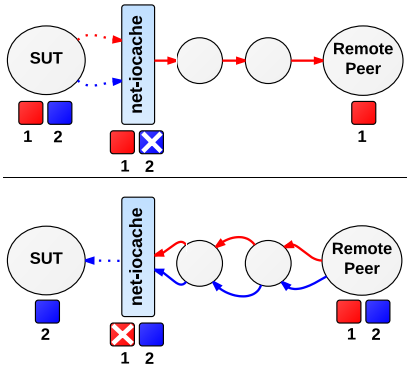

Figure 4. Generation of packet loss by net-iocache.

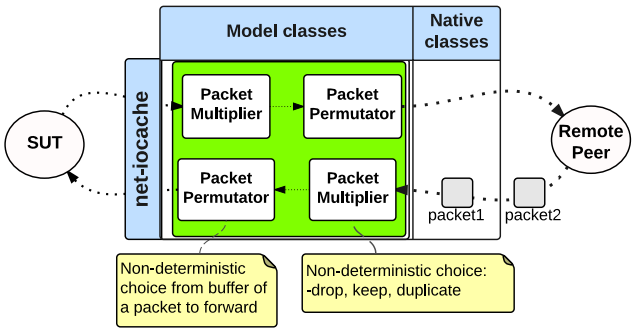

Figure 5. Systematic generation of UDP unreliable I/O in model class `DatagramSocket` of net-iocache.

Figure 4 illustrates the ability of net-iocache to drop a packet sent to or received from the remote peer, by not forwarding the packet to the network or to the SUT, respectively. In the upper part of Figure 4, net-iocache drops packet 2 sent by the SUT, while in the lower part, packet 1 sent by the remote peer is not forwarded to the SUT.

Figure 5 depicts the components involved in systematic simulation of unreliable UDP behavior. As mentioned in Section III-A, net-iocache is divided into model classes and native classes. We implemented our solution in the model class of `DatagramSocket` in the form of two modules: the `PacketMultiplier` module, and the `PacketPermutator` module. The `PacketMultiplier` generates cases of packet loss and/or duplication, while the `PacketPermutator` uses a buffer to change the order of datagram packets.

Both the multiplier and permutator modules perform non-deterministic choices. These are implemented in the `chooseFrom` method in Algorithms 2, 3, 5, and 6.

In the sequel, we describe the realization of the `PacketMultiplier` and `PacketPermutator` modules in send direction (Algorithms 2 and 3). The receive direction (Algorithms 5 and 6) is implemented similarly.

When the SUT sends a datagram packet, the sent packet is forwarded to the `PacketMultiplier` (Algorithm 1, line 2). `PacketMultiplier` determines the number of

```
1 Function send(packet)
2     packetMultiplierPush(packet);
```

Algorithm 1: *send* method of `DatagramSocket` in net-iocache.

```
1 Function packetMultiplierPush(packet)
2     counter_packet ← chooseFrom(MULTIPLYCHOICES);
3     if counter_packet > 0 then
4         packetPermutatorPush(packet);
```

Algorithm 2: `PacketMultiplier` module in *send* direction.

```
1 Function packetPermutatorPush(packetIn)
2     sendBuffer ← sendBuffer ∪ {packetIn};
3     flushSendBuffer(BUFFERLIMIT − 1);
4 Function flushSendBuffer(newSize)
5     while |sendBuffer| > newSize do
6         packetOut ← chooseFrom(sendBuffer);
7         sendToNetwork(packetOut);
8         counter_packetOut ← counter_packetOut − 1;
9         if counter_packetOut = 0 then
10            sendBuffer ← sendBuffer\{packetOut};
```

Algorithm 3: `PacketPermutator` module in *send* direction.

```
1 Function receive(packet)
2     packetPermutatorPull(packet);
```

Algorithm 4: *receive* method of `DatagramSocket` in net-iocache.

```
1 Function packetPermutatorPull(packetOut)
2     while receiveBuffer= ∅ ∨ (|receiveBuffer| <
      BUFFERLIMIT ∧ ¬timeout) do
3         packetMultiplierPull(packetIn);
4         receiveBuffer ← receiveBuffer ∪ {packetIn};
5     packetOut ← chooseFrom(receiveBuffer);
6     counter_packetOut ← counter_packetOut − 1;
7     if counter_packetOut = 0 then
8         receiveBuffer ← receiveBuffer\{packetOut};
```

Algorithm 5: `PacketPermutator` module in *receive* direction.

```
1 Function packetMultiplierPull(packet)
2     flushSendBuffer(0);
3     receiveFromNetwork(packet);
4     counter_packet ← chooseFrom(MULTIPLYCHOICES);
5     if counter_packet < 1 then
6         packetMultiplierPull(packet);
```

Algorithm 6: `PacketMultiplier` module in *receive* direction.

instances to be generated from the passed packet, based on the user-configured list `MULTIPLYCHOICES`. For instance, if `MULTIPLYCHOICES` is set to "`0,1`", line 2 of Algorithm 2 performs a non-deterministic choice from the cases "packet loss" and "packet delivery exactly once", and stores the result of this choice in a counter for the packet to be sent.

Function `packetPermutatorPush` (Algorithm 3) adds packets passed from function `packetMultiplierPush` to a set `sendBuffer` whose maximum size is constrained by the configurable number `BUFFERLIMIT`. Line 6 of Algorithm 3 makes a non-deterministic choice every time a packet is selected from `sendBuffer` and forwarded to the network. The combination of non-deterministic choices in the `PacketMultiplier` and `PacketPermutator` modules generates all instances of reliable and unreliable UDP behavior in the limits given by `MULTIPLYCHOICES` and `BUFFERLIMIT`.

The non-deterministic choice `chooseFrom` in Algorithms 2, 3, 5, 6 is implemented using the choice generation mechanism provided by JPF's verification API. Choice generators create a separate execution branch for each choice from a specified range. For instance, the statement `int i=Verify.getInt(min, max)` instructs JPF to execute the rest of the SUT for all values of `i` between `min` and `max` in `max-min+1` separate execution branches. This is used in `chooseFrom` to let JPF exhaustively explore the different possibilities of packet loss, duplication, and reordering within the constraints of `MULTIPLYCHOICES` and `BUFFERLIMIT`.

Note that in many applications, some packets are sent or received in response to previous network traffic [11]. We do not reorder packets across such logical message boundaries, as this would produce communications that are not possible in reality. In our implementation, before receiving packets, all packets in the send buffer are forwarded to the network in line 2 of Algorithm 6 to ensure that responses to previously sent packets are transmitted. Emptying the send buffer on receive restricts the reordering of outgoing packets.

Conversely, if the remote peers do not send sufficiently many packets to the SUT, it is not possible to fill the `receiveBuffer` up to its limit which restricts the reordering of incoming packets. In this case, the `while` loop in line 2 of Algorithm 5 terminates early because of a time out.

As a result, the communication pattern between the SUT and the remote peers may restrict packet reordering beyond the user-defined `BUFFERLIMIT`. In particular, a sequential request-response pattern, where a peer waits for the single response to a previous request before sending the next request, prevents packet reordering.

C. Configuration of Systematic Simulation

For UDP-based distributed applications, the size of the SUT's state space depends mainly on two factors:

the number of packets exchanged and the settings of `MULTIPLYCHOICES` and `BUFFERLIMIT`, constraining the explored UDP behaviors.

In the case of single threaded SUTs and few exchanged packets (<5), it may be feasible to verify the SUT against all combinations of packet loss, duplication, and reordering. In other cases, the number of I/O outcomes may be too large to be explored exhaustively within the available amount of memory and time. We leverage JPF's configuration framework to enable user-defined setting of systematic simulation to control complexity.

Simulation of packet loss and duplication are defined in the `MULTIPLYCHOICES` parameter, while the window for reordering is defined in `BUFFERLIMIT` in `PacketMultiplier` and `PacketPermutator`.

For the configuration of these parameters, we added the following JPF properties:

```
jpf-net-iocache.UDP.multiplyChoices=1,0,2
jpf-net-iocache.UDP.bufferLimit=1
```

The setting of the `BUFFERLIMIT` to 1 disables reordering of packets. The sequence `1,0,2` set for `MULTIPLYCHOICES` configures the `PacketMultiplier` module to explore three execution branches for each packet; first, packet sent/received exactly once (1), second, sent/received packet lost (0), and third packet sent/received twice (2). In the case of `jpf-net-iocache.UDP.multiplyChoices=0,2,1` JPF explores the same cases as above but with packet loss as the first choice, followed by duplication, and finally normal delivery of packets.

Using similar properties we can set the parameters `MULTIPLYCHOICES` and `BUFFERLIMIT` for `receive` and `send` methods separately.

JPF's framework enables us to set properties statically in a configuration file for each SUT or by inserting instructions inside the SUT's source code using the API method `Verify.setProperties` to dynamically change configuration settings during exploration. This method can be used to restrict systematic simulation to sections of interest in the SUT's source code. The dynamic configuration mechanism increases scalability and modularity of unreliability simulation, because it can be tailored to specific requirements of each SUT component.

IV. Experimental Results

In this section, we analyze the performance of net-iocache v2 with UDP support, and demonstrate its usefulness for finding defects in a protocol for UDP-based file transfer.

A. Performance Analysis

In a first experiment, we analyzed the performance of net-iocache v2 w. r. t. runtime and memory consumption, using the *alphabet client/server* application as described in [11]. Figure 6 depicts the basic architecture of the application.

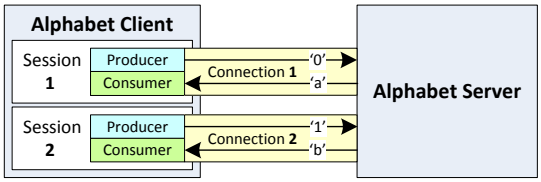

Figure 6. Architecture of the alphabet client/server application.

The alphabet client starts concurrent sessions, each communicating independently with the alphabet server through an own dedicated connection, implemented either by a TCP or UDP socket. Each session runs two threads: the *producer* thread sending requests and the *consumer* thread receiving the corresponding responses from the alphabet server. The alphabet server answers requests, consisting of a single digit '0', '1', ... with corresponding letters 'a', 'b', ... For comparability, the UDP-based alphabet client is kept similar to the original TCP-based version and hence does *not* cope with UDP's unreliability.

The complexity of the alphabet client is determined by

- the number of sessions/connections to the server;
- the number of requests sent on each connection.

In the subsequent experiments, we report the results obtained for 3 connections and 1–8 requests, because this setting turned out to be challenging but still manageable.

We run the alphabet client as an SUT on JPF with net-iocache to check whether the received responses are correct for all interleavings of producer and consumer threads, sending and receiving messages concurrently on different connections. Therefore, the consumer thread checks the following assertion after receiving $response_i$ for the i-th request $request_i$ sent on its connection to the server:

$$response_i = request_i + OFFSET$$

$OFFSET$ denotes the difference of the ASCII codes of letters 'a' and '0'. The assertion holds as long as connections to the server are reliable because on each connection, the alphabet server sends responses in the order of the received requests. However, it is likely to fail if, in the case of UDP, packets are lost, duplicated, or reordered.

Table I shows the results of executing the TCP and UDP-based alphabet client on the standard Oracle JVM (32 Bit Java RTE 1.7.0_25-b15 / Java HotSpot Server VM 23.25-b01). Both the alphabet client and server were executed on the same 8 core Mac Pro workstation with 24 GB of memory running Ubuntu 12.04.4.

Since the SUT passes the assertion in all test cases we conclude that UDP behaves reliably in the test setting. The runtime in column 4 of Table I includes the invocation of the JVM. The total number of I/O operations

Table I
TCP AND UDP ALPHABET CLIENT, 3 CONN. AND 1–3 REQUESTS, ON THE ORACLE JVM.

#Req	Protocol	Fault found	Time [s]	Heap [MB]	#I/O
1	TCP	–	0.082	2.6	15
	UDP	**no**	0.078	2.6	15
2	TCP	–	0.082	2.6	21
	UDP	**no**	0.079	2.6	21
3	TCP	–	0.083	2.6	27
	UDP	**no**	0.080	2.6	27

Table II
TCP AND UDP ALPHABET CLIENT, 3 CONNECTIONS AND 1–3 REQUESTS, ON NET-IOCACHE.

#Req	Protocol	iocache version	Pkt loss	Assertion	Fault found	Time [hh:mm:ss]	Heap [MB]	#I/O	#Transitions
1	TCP	v1	–	active	–	0:00:18	176	5,512	109,286
		v2				0:00:08	77	1,158	2,143
	UDP	v2	no	active	no	0:00:03	109	1,158	2,146
			yes			0:01:13	77	10,758	28,450
2	TCP	v1	–	active	–	0:01:31	672	47,898	573,711
		v2				0:00:37	77	5,190	7,807
	UDP	v2	no	active	no	0:00:29	77	5,190	7,810
			yes		**yes**	<00:00:01	61	26	56
				inactive	no	0:45:12	109	231,984	534,592
3	TCP	v1	–	active	–	0:06:13	**1,017**	233,367	2,303,302
		v2				0:02:06	77	17,346	23,440
	UDP	v2	no	active	no	0:01:36	77	17,346	23,443
			yes		**yes**	<00:00:01	61	32	68
			yes	inactive	no	**11:03:02**	77	**3,276,081**	**7,066,609**

(column #I/O in Table I) comprises operations for creating/connecting/closing sockets and transmitting messages.

Table II shows the results of executing the TCP and UDP alphabet client on JPF 7 rev 1155 with `net-iocache` v1 (no UDP support [7]) and v2 (with UDP support as described in Section III-A). The UDP test cases were executed both without and with simulation of packet loss using the configuration property `jpf-net-iocache.UDP.multiplyChoices` (see Section III-C). Failing cases were re-executed with the failing assertion deactivated, to let JPF explore the entire state space of the SUT. This simulates an SUT without defects.

In addition to the test result, the execution time, used heap memory, the number of executed I/O operations, and the number of transitions were determined for each test case (columns 6–10 in Table II). The number of transitions is proportional to the number of instructions JPF executed for the exploration of the SUT's state space, including instructions in called methods of library classes such as `java.net.Socket`. It is a good indicator of the problem size. The results are as follows:

- All test cases meet the assertion except UDP verified with packet loss simulation for more than one request. Similar as in the case of using the standard Oracle JVM (Table I), we did not observe unreliable UDP behavior even in test cases with more than 10,000 I/O operations, as long as net-iocache does not inject it in the form of packet loss. If net-iocache drops packets, JPF finds defects early in its state space search which is consistent with results in our previous work [9].
- Exhaustive packet loss simulation is expensive if no error is detected (see, e. g., last row of Table II). Packet loss simulation introduces—in addition to thread schedules—another dimension of exponential growth of the state space in the number of requests. This is because on each send and receive, JPF executes the rest of the SUT twice, to check its behavior for the two outcomes "packet delivered" and "packet lost".
- net-iocache v2 performs better than v1, especially in terms of memory consumption. The major source of overhead in JPF/net-iocache v1 is the number of executed transitions, caused by the code in model classes such as `java.net.Socket` that net-iocache provides to support TCP sockets in JPF (Section III-A). In net-iocache v2, we reduced the code in model classes to a minimum.
- Without simulation of unreliable behavior, UDP performs slightly better than TCP, although the number of I/O operations and transitions are (almost) identical (entries in blue in Table II). We suppose that establishing and terminating connections, which is missing in UDP, takes significant extra time.

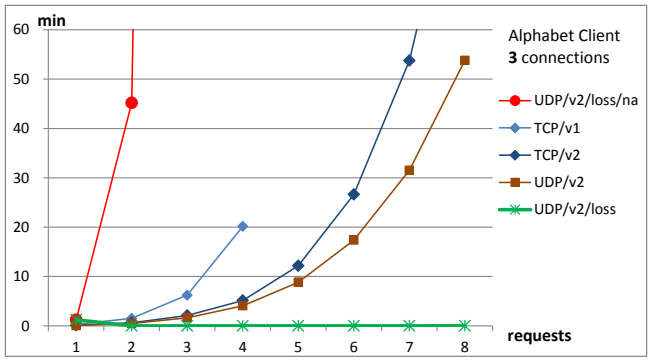

Figure 7. Runtime behavior of net-iocache on different variants of the alphabet client. The first case refers to UDP on net-iocache v2 with packet loss enabled and no assertions.

Figure 7 shows the runtime results for model checking each test configuration for up to 8 requests (horizontal axis). The vertical axis shows the time in minutes it took JPF to explore the entire state space of the SUT or to detect an assertion violation. Our observations are:

- Except for UDP/v2/loss, the runtime of all configurations increases exponentially in the number of requests. This is because the number of interleavings of producer and consumer threads grows exponentially in the number of requests. For UDP without assertions, packet loss simulation adds another dimension of exponential growth (UDP/v2/loss/na in Figure 7). Conversely, the runtime of UDP/v2/loss (with assertion) is largest for one request (1.2 min) because this case still passes. If more than one request is sent, JPF reports the failing assertion within one 1 second.
- net-iocache v1 runs out of 2 GB heap memory when executing it on the TCP alphabet client for more than 4 requests (TCP/v1 in Figure 7). net-iocache v2, however, succeeds to verify 8 requests on each connection using only 109 MB of memory, in 1 hour and 41 minutes (TCP/v2 in Figure 7).
- The difference between TCP/v2 and UDP/v2 (no packet loss) rises from a factor of 1.3 for smaller cases to a factor of 1.9 in the case of 8 requests. We assume that the system runs out of ephemeral ports when executing larger series of benchmarks and the OS requires more time to re-allocate used TCP ports than UDP ports.

B. Verifying a UDP-based File Transfer Protocol

We choose as an example a UDP-based client/server application for the transfer of multiple files from the server to the client whose main method is shown in Algorithm 7. Using a TCP connection, the client first requests the number of packets necessary to transfer a specific file (line 3 of Algorithm 7) from the server provided the file exists in the server's repository. After that, the client calls `getFile` to initiate the file transfer using UDP. Function `getFile` should ensure the validity of the returned packet array by compensating for packet loss, duplication and reordering (Algorithm 8). For testing purposes, we compare the contents of the received data against a local reference copy of the file (line 5 in Algorithm 7).

```
1  Function main()
       Input: fileNames: identifiers of files to be retrieved from
              the server
2      for fileName ∈ fileNames do
3          filePackets ← numberOfPackets(fileName);
4          data ← getFile(fileName, filePackets);
5          assert isContentEqual(fileName, data);
```

Algorithm 7: `main` method of the file transfer client.

Function `getFile` (Algorithm 8) copes with loss, duplication, and out-of-order arrival of packets by using the packet sequence numbers the server adds to the data of each packet. The UML sequence diagram in Figure 8 shows correctly handled cases of packet duplication and out-of-order delivery.

```
1  Function getFile(fName, fPackets): Packet[]
2      data ← new Packet[fPackets];
3      missingPackets ← {0, 1, ..., fPackets − 1};
4      send(request(fName, missingPackets));
5      while missingPackets ≠ ∅ do
6          receive(packet);
7          if timeout then
8              send(request(fName, missingPackets));
9          else
10             index ← sequenceNumber(packet);
11             if index ∈ missingPackets then
12                 data[index] ← packet;
13                 missingPackets ← missingPackets\{index};
14     return data;
```

Algorithm 8: Function `getFile()` of the client side of file transfer application.

However, net-iocache demonstrates that incorrectly handled packets may cause files to be received incorrectly, resulting in an assertion failure (see UML sequence diagram in Figure 9). The defect is revealed when the client of the file transfer application is model checked and packet duplication simulation is enabled using the following configuration (see Section III-C):

```
jpf-net-iocache.UDP.multiplyChoices=1,2
jpf-net-iocache.UDP.bufferLimit=1
```

For two or more files requested by the client, the duplication of the last packet of a file f_i is not discarded if the subsequently requested file f_{i+1} has at least as many packets. This is, because the client interprets the duplicated packet of file f_i as a missing packet of file f_{i+1} in line 11

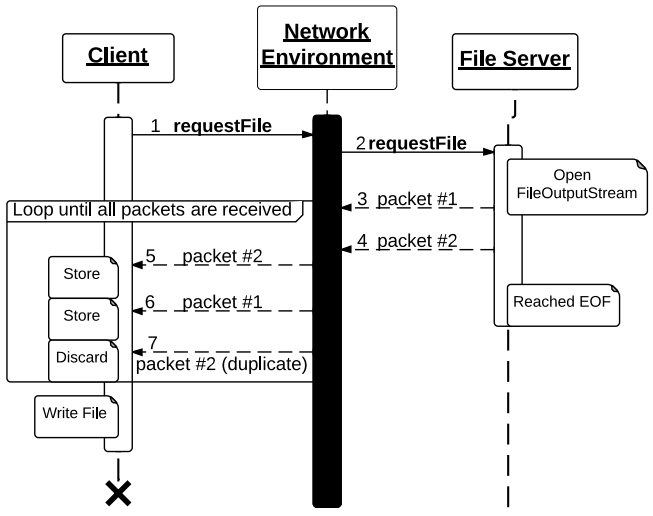

Figure 8. Scenario where the client compensates for packet reordering and duplication.

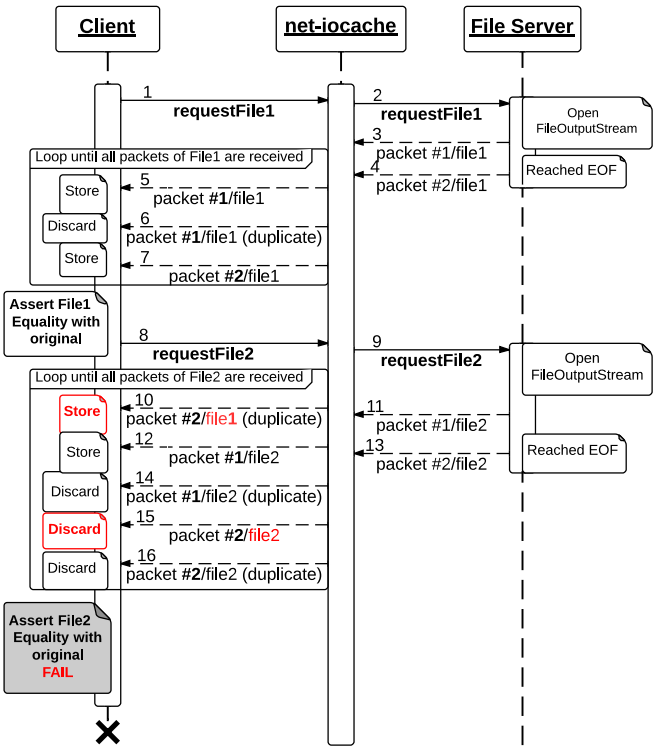

Figure 9. Execution trace generated by net-iocache that finds a defect.

of Algorithm 8, and stores it in the data array for file f_{i+1}.

A possible solution is including a file ID in each packet the server sends and adding a condition to compare file IDs in the `if` statement at line 11 of Algorithm 8. This allows the client to distinguish packets of different files.

Classical testing of this file transfer application did not reveal any bug, and would hardly detect any in network environments where unreliable UDP behavior rarely occurs. Classical testing therefore gives poor coverage of the application-level protocol code (in `getFile`) because scenarios where it has to compensate for unreliable I/O rarely occur.

Because the file download application fails in a very specific I/O schedule, stochastic approaches that randomly generate unreliable UDP I/O behavior [2], [3], [4], [5] (see Section V-A) are likely to miss the fault.

V. Related Work

Although testing of the UDP protocol is largely documented, software model checking targeting UDP-based applications has not enjoyed much attention in research.

A. Analysis of UDP-based Applications

Previous work [2] proposes a framework for testing UDP-based applications by adding a layer over the `java.net` API that simulates network noises. The network emulator NetEm [3] and its extensions [4], [5] are Linux modules that inject stochastically packet delays, loss, duplication, reordering, and IP packet corruption to simulate non-deterministic unreliable UDP I/O. In contrast to these approaches, instead or randomly introducing faults, we *systematically* explore outcomes of UDP-based communication while model checking the application.

Rathje [12] proposes the concept of using JPF to simulate UDP packet loss and reordering in a small proof of concept implemented for educational purposes.

B. Systematic Testing of Actor-based Systems

In actor-based systems, multiple autonomous agents communicate by exchanging messages asynchronously. The distributed nature of agents in actor-based systems results in the non-deterministic processing of messages since in-order delivery of messages is not guaranteed unless constrained in some actor language. Lauterburg [13] proposes a framework for systematically testing actor-based systems by using JPF for exploring different scheduling of messages. While this work has in common with our approach the use of JPF to enable exhaustive coverage of the state space and to control the way messages are exchanged across the network, it targets a different kind of applications (actor-based systems) while our focus is on UDP-based applications.

C. Software Model Checking of Distributed Applications

There are two major approaches to software model checking of communicating peer processes: 1) merging multiple peer processes into a single multi-threaded process and verifying it (*centralized approach*) and 2) verifying a single process at a time (*modular approach*). The former approach is complete but does not scale well in the number of processes. The latter offers better scalability but is not complete in general [7].

Stoller and Liu [14] introduced the concept of *centralisation* first when they suggested to extend software model checking of Java programs to distributed applications by merging multiple processes into one and simulate Java RMI method invocations by local method calls. This work has since then been extended to full TCP sockets [15], [16]. A similar approach analyzes the complete state space of all processes by extending JPF itself [17], [10] rather than preprocessing the SUT.

These approaches abstract from the physical behavior of communicating processes by simulating the exchange of messages using queues in shared memory. Low-level network behavior such as packet loss or I/O exceptions resulting from dropped connections, are not in the focus of these approaches, except in one case where scalability was addressed by limiting the types of faults injected [18]. The above-mentioned approaches also take a global view of all processes, whereas our approach is process-modular [7].

Conceptually our approach is related to the use of stubs [19], but we execute unmodified peer processes [7],

rather than then user-defined stubs, to obtain the input data for the SUT.

VI. Conclusion and Future Work

We combine software model checking (exhaustive execution of non-deterministic thread-interleavings) and systematic simulation of UDP I/O outcomes at the packet-level, to analyze UDP-based applications.

This required the redesign of net-iocache, which now decouples the caching and communication mechanisms. The exhaustive simulation of UDP I/O outcomes results in a state space explosion, which we curb by a user-configurable limitation of possible outcomes. Our experiments show that the resulting tool is scalable and can detect subtle defects that are not found by classical testing. This helps to reduce the effort for quality assurance and to increase the reliability of distributed applications using UDP.

Future work includes the complexity analysis of the presented algorithms, based on a formal model of UDP I/O. We also want to check the compliance of our implementation of the `DatagramSocket` API with the standard Java library, using the Modbat [20] tool for model-based testing, similar to past work [9]. Finally, we consider integrating UDP support with work on jpf-nas [17], which verifies distributed systems based on centralization.

References

[1] L. Eggert and G. Fairhurst, "Unicast UDP usage guidelines for application designers," BCP 145, IETF RFC 5405, 2008, available on http://www.hjp.at/doc/rfc/rfc5405.html, accessed: 7th Aug 2014.

[2] E. Farchi, Y. Krasny, and Y. Nir, "Automatic simulation of network problems in UDP-based Java programs," in *Proc. 18th International Parallel and Distributed Processing Symposium*. IEEE, 2004.

[3] L. Foundation, "Network emulation with netem." http://www.linuxfoundation.org/collaborate/workgroups/networking/netem, accessed: 7th Oct 2014.

[4] P. Reinecke, M. Drager, and K. Wolter, "Netemcg – ip packet-loss injection using a continuous-time gilbert model," Freie Universitt Berlin, Germany, Tech. Rep. TR-B-11-05, 2011.

[5] J. Sliwinski, A. Beben, and P. Krawiec, "Empath: Tool to emulate packet transfer characteristics in ip network," in *Proc. Second International Workshop, TMA 2010*. Zurich, Switzerland: Springer Berlin Heidelberg, 2010.

[6] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering Journal*, vol. 10, no. 2, pp. 203–232, 2003.

[7] W. Leungwattanakit, C. Artho, M. Hagiya, Y. Tanabe, M. Yamamoto, and K. Takahashi, "Modular software model checking for distributed systems," *IEEE Transactions on Software Engineering*, vol. 40, no. 5, pp. 483–501, 2014.

[8] K. Havelund and T. Pressburger, "Model checking Java programs using Java PathFinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000.

[9] C. Artho, M. Hagiya, R. Potter, Y. Tanabe, F. Weitl, and M. Yamamoto, "Software model checking for distributed systems with selector-based, non-blocking communication," in *Proc. 28th Int. Conf. on Automated Software Engineering (ASE 2013)*. IEEE, 2013, pp. 169–179.

[10] N. Shafiei and F. V. Breugel, "Automatic handling of native methods in Java PathFinder," in *Proc. International SPIN Symposium on Model Checking of Software (SPIN 2014)*, San Jose, California, USA, 2014.

[11] C. Artho, W. Leungwattanakit, M. Hagiya, and Y. Tanabe, "Efficient model checking of networked applications," in *Proc. 46th Int. Conf. on Objects, Models, Components, Patterns (TOOLS EUROPE 2008)*, ser. LNBIP, vol. 19. Zurich, Switzerland: Springer, 2008, pp. 22–40.

[12] B. Rathje, "A novel framework for model checking UDP network interactions," *Summer Research*, no. 205, 2013.

[13] S. Lauterburg, M. Dotta, D. Marinov, and G. Agha, "A framework for state-space exploration of Java-based actor programs," in *ASE*, 2009, pp. 468–479.

[14] S. D. Stoller and Y. A. Liu, "Transformations for model checking distributed Java programs," in *Proc. 8th Int. SPIN Workshop (SPIN 2001)*. NY, USA: Springer-Verlag New York, Inc., 2001, pp. 192–199.

[15] C. Artho and P. Garoche, "Accurate centralization for applying model checking on networked applications," in *Proc. 21st Int. Conf. on Automated Software Engineering (ASE 2006)*, Tokyo, Japan, 2006, pp. 177–188.

[16] L. Ma, C. Artho, and H. Sato, "Analyzing distributed Java applications by automatic centralization," in *Proc. 2nd IEEE Workshop on Tools in Process*. Kyoto, Japan: IEEE, 2013.

[17] N. Shafiei and P. C. Mehlitz, "Extending JPF to verify distributed systems," *ACM SIGSOFT Software Engineering Notes*, vol. 39, no. 1, pp. 1–5, 2014.

[18] C. Artho, C. Sommer, and S. Honiden, "Model checking networked programs in the presence of transmission failures," in *TASE 2007: Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 219–228.

[19] E. D. Barlas and T. Bultan, "NetStub: A framework for verification of distributed Java applications," in *Proc. 22nd Int. Conf. on Automated Software Engineering (ASE 2007)*, Georgia, USA, 2007, pp. 24–33.

[20] C. Artho, A. Biere, M. Hagiya, E. Platon, M. Seidl, Y. Tanabe, and M. Yamamoto, "Modbat: A model-based API tester for event-driven systems," in *Proc. 9th Haifa Verification Conference (HVC 2013)*, ser. LNCS, vol. 8244. Haifa, Israel: Springer, 2013, pp. 112–128.