International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 5, Number 2, pages 373–402, July 2015

Software Model Checking of UDP-based Distributed Applications

Nazim Sebih, Masami Hagiya
The University of Tokyo
Tokyo, Japan

Franz Weitl, Mitsuharu Yamamoto
Chiba University
Chiba, Japan

Cyrille Artho
AIST/RISEC
Amagasaki, Japan

Yoshinori Tanabe
Tsurumi University
Yokohama, Japan

Received: February 13, 2015
Revised: April 30, 2015
Accepted: June 1, 2015
Communicated by Hiroyuki Sato

Abstract

An extension to the software model checker Java Pathfinder for verifying networked applications using the User Datagram Protocol (UDP) is presented.

UDP maximizes performance by omitting flow control and connection handling. For instance, media-streaming services often use UDP to reduce delay and jitter. 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. 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 I/O

1 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 [12].

Despite its unreliability, UDP is adopted for a broad range of safety- and mission-critical applications and contributes significantly to the Internet traffic volume [45]. Its lower latency and higher achievable throughput motivate its adoption for application-layer protocols such DNS [29] and DHCP [11]. Particularly security-sensitive applications are authentication and password services such as Kerberos [28], remote desktop access [19, 31], and network management protocols such as SNMP [26]. Virtual private networks (VPN) use UDP as an underlying transport protocol to avoid redundant layers of flow control when tunneling TCP connections [17]. *Multicast* protocols such as IPTV [7] or the Starbust Multicast File Transfer Protocol (MFTP) [34] depend on UDP since TCP does not support connections with one source and multiple destinations. *Real-time* protocols such as RTP/RTCP [18, 36], used for telephony and teleconference applications for instance, build preferably on UDP because of its more predictable delay and jitter. Recent mission-critical real-time applications include smart meters [14] used for optimizing the consumption and distribution of electrical power. Highly scalable middleware uses UDP for performance optimization. For instance, the distributed coordination system Apache ZooKeeper [20] includes a UDP-based version of *leader election* which is the core algorithm for recovering from failures. Recently, Google is pushing the new UDP-based protocol QUIC [35] to be adopted in future versions of HTTP. It has shown potential to speed up web applications beyond HTTP/2 and is already supported by Google services and the Chrome browser.

Each of these applications must be dependable. Developing dependable UDP-based systems is difficult because an unreliable protocol 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 UDP, summarized under the term *packet perturbation*:

1. *Loss:* a packet does not arrive at its destination;
2. *Duplication:* a packet arrives more than once;
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 [13, 23, 33, 40] generate unreliable UDP behavior with a configurable stochastic distribution. However, it is hard to guarantee coverage and to reproduce rarely occurring errors by randomly generating unreliable UDP behavior. 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 and configurable way (*systematic generation of packet perturbation*). We implement our approach using the software model checker Java Pathfinder [42] and its extension net-iocache [22] for distributed systems.

Considering both the reliable and (combinations of) unreliable behaviors for each UDP I/O operation often 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.[37]

This article is an extended version of previous work [37]. Its contributions are:

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 by generating packet perturbation.

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 article adds the following to previously published results [37]: 1) formal description of UDP's unreliable behavior which defines the output of the proposed algorithms for generating packet perturbation, 2) additional experimental results obtained with different configurations of packet perturbation on two distributed applications.

This article 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 2. Section 3 formalizes the concept of unreliable UDP transmissions, presents the algorithms that generates them, and explains their implementation in net-iocache. We report on experimental results in Section 4 and discuss related work in Section 5 before concluding the article in Section 6.

2 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.

2.1 Software Model Checking with Java Pathfinder

Our implementation extends JPF [15, 42], 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.

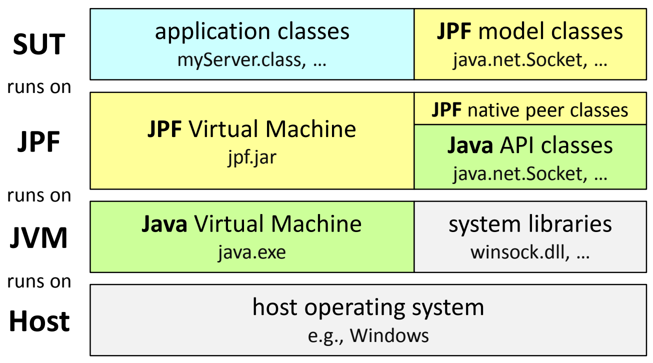

Figure 1: Levels of execution when model checking an SUT (system under test) with JPF (Java Pathfinder).

Figure 1 shows the components of JPF. JPF is a custom Java Virtual Machine written in Java, i. e., it 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 a 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.

2.2 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 [4, 22]. 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).

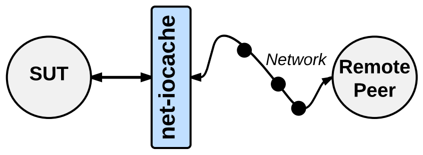

Figure 2: Tool net-iocache intercepts the communication between the SUT executed by JPF and remote peers.

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 JPF backtracked its state. For model checking of UDP applications, we added support of datagram sockets to net-iocache as described in Section 3.3.1.

2.3 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 has been 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.

3 Simulation of Unreliable UDP in net-iocache

The proposed approach for software model checking UDP-based applications with systematic generation of packet loss, duplication, and reordering consists of the following parts:

1. Formal model of unreliable UDP transmissions with packet loss, duplication, and reordering.
2. Algorithms for exhaustively generating possible UDP transmission outcomes, based on the formal model and JPF's choice generator mechanism.
3. Extension of JPF towards UDP support and implementation of the algorithms simulating unreliable UDP behavior.

3.1 Formal Model of Unreliable UDP Transmissions

The formal model presented in this section clarifies our assumptions and defines the output of the algorithms for generating packet perturbation.

Example 1 (UDP Transmission Outcomes)**.**
Consider a UDP-based network communication where a server sends subsequently two UDP packets (p, q) to a client. Each packet may be subject to *perturbation*, i. e. , it may be lost, duplicated and/or reordered. The following lists the packet sequences the client could possibly receive, assuming that duplication occurs at most once per packet:

$$\begin{array}{ccccc}
() & (p) & (p,p) & (p,p,q) & (p,p,q,q) \\
 & (q) & (q,q) & (q,q,p) & (q,q,p,p) \\
 & & (p,q) & (p,q,p) & (p,q,p,q) \\
 & & (q,p) & (q,p,q) & (q,p,q,p) \\
 & & & (q,p,p) & (p,q,q,p) \\
 & & & (p,q,q) & (q,p,p,q)
\end{array}$$

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

To exclude an infinite number of UDP transmission outcomes for a transmitted packet sequence, we limit the number of times each packet may be duplicated in our model. For instance, we assume that each packet is not duplicated more than once. Under this assumption, each transmitted packet may either be (1) lost, (2) transmitted exactly once, or (3) duplicated and transmitted twice, resulting in 3 cases per packet and 3^n cases for n packets. Reordering of n packets leads to $n!$ cases. Hence testing an SUT against all UDP transmission outcomes may not be feasible for larger number of packets.

To control complexity, it is desirable to restrict the generated transmission outcomes to cases of perturbation that match the test goals and properties of the network environment. We therefore identify restrictions on packet loss, duplication, and reordering and formally define the set of UDP transmission outcomes w. r. t. these restrictions which are generated by net-iocache. This systematic and exhaustive generation of non-deterministic UDP I/O w. r. t. constraints on packet perturbation distinguishes our work from other work [13], applying stochastic methods for the generation of unreliable UDP behavior.

3.1.1 Modeling UDP Behavior

For our model of UDP transmissions, we consider two endpoints where one endpoint, the *sender*, sends a sequence of n distinct UDP packets to the *receiver* (Figure 3).

We model the network as an unreliable channel with limited capacity, i. e. , it can hold at most c packets at a given time. Figure 3 depicts the scenario where the first c of n packets to be sent have already been put onto the network but not yet delivered to the receiver. In this scenario, any further packet forwarded to the network is lost unless one of the c packets on the network has been delivered to the receiver or is lost. We call the sequence of packets $(p_1, ..., p_n)$ forwarded by the

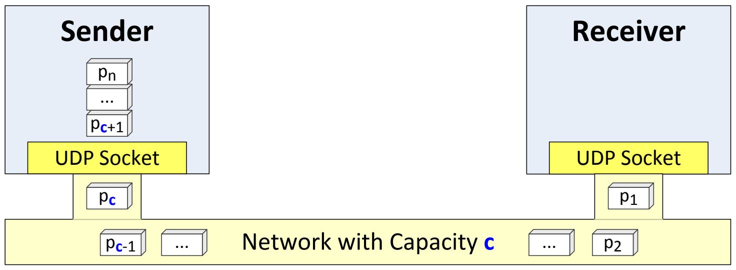

Figure 3: Transmission of a sequence of n UDP packets on a network with capacity c.

sender to the network *dispatch* while *delivery* denominates the sequence of packets arriving at the UDP socket of the receiver. The unreliability of the channel causes differences between dispatches and their deliveries: In the delivered sequence, packets may appear in an order different to that of the dispatched sequence, some packets may be missing and others may appear more than once. However, we assume that lower-level network protocols ensure the integrity of each individual packet: If a packet is delivered its content is unmodified. In total, our assumptions regarding the behavior of the network are:

- Sequential dispatch/delivery: In one execution step, the UDP socket of the sender dispatches at most one packet to the network and at most one packet is delivered to the UDP socket of the receiver;
- Delay: A packet is not dispatched and delivered at the same time;
- Progress: Every packet on the network is eventually not on the network anymore (although not necessary delivered, see below);
- Finite capacity: At any time, there are at most $c \geq 1$ packets on the network;
- Packet integrity: Only packets are delivered to the receiver which have been dispatched by the sender;
- Loss and duplication: Each packet on the network is delivered arbitrarily many times, including 0;
- Unordered delivery: Any of the packets on the network at a certain time may be delivered next.

These assumptions represent a worst-case scenario, allowing a maximum of packet perturbation. In real networks, the number of duplicates and reordering may be restricted, for instance, by limitations on the time a single packet can remain on the network.

In our model, the set of possible deliveries for a given dispatched sequence of packets depends on the network capacity and the number of duplicates introduced by the network. In the sequel, we define our notion of UDP transmissions formally.

3.1.2 Formalization of UDP Transmissions

We denote the set of natural numbers including 0 as $\mathbb{N}$. Furthermore, $\mathbb{N}_1 =_{def} \mathbb{N} \setminus \{0\}$ denotes the set of positive natural numbers; $[n, m] =_{def} \{i \in \mathbb{N} \mid n \leq i \leq m\}$ denotes a closed interval in $\mathbb{N}$.

Definition 1 (Packet, Packet Sequence)**.**

- P denotes an infinite set of *packets*.
- P^n with $n \in \mathbb{N}$ denotes the set of *packet sequences* of length n. Elements of P^n are denoted as $(p_1, ..., p_n)$ or as $(p)_n$. "()" denotes the empty sequence for $n = 0$.

- $P^{<\infty} =_{def} \bigcup_{n\in\mathbb{N}} P^n$ is the set of finite packet sequences.

Definition 2 (Dispatch, Delivery, Dispatch Order)**.**
Let $(p)_n \in P^{<\infty}$ be a packet sequence of length n. Then

- $(p)_n$ is a *dispatch* iff $p_i = p_j$ implies $i = j$ for all $i, j \in [1, n]$.
- $(q)_m \in P^{<\infty}$ is a *delivery* of a dispatch $(p)_n$ iff for each $j \in [1, m]$ there is $i \in [1, n]$ such that $q_j = p_i$.
- $D_{(p)_n} =_{def} \{(q)_m \in P^{<\infty} \mid \forall j \in [1, m]\ \exists i \in [1, n] : q_j = p_i)\}$ denotes the *set of deliveries* of a dispatch $(p)_n$.
- The *dispatch order* of packets in a dispatch $(p)_n$ is:
$p_i < p_j \Leftrightarrow_{def} i < j$ for all $i, j \in [1, n]$

Remark 1 (Dispatch, Delivery).
A *dispatch* represents a sequence of packets the UDP socket of a sender forwards to the network. Definition 2 characterizes each packet of a dispatch $(p)_n$ as unique. This models the behavior of the network which does not identify packets with the same content but delivers each packet individually: Every packet dispatched to the network is treated as a new packet different from previously dispatched packets regardless of its content.

Each element in the delivery set $D_{(p)_n}$ represents a packet sequence that is possibly received when $(p)_n$ is sent on a network with *unlimited* capacity. There are no restrictions regarding the amount of packet perturbation in the form of packet loss, duplication, and reordering. For non-empty dispatches $(p)_n$ with $n > 0$, $D_{(p)_n}$ is infinite because the number of the duplications and hence the length of a delivery is not restricted in Definition 2.

Example 2 (Dispatch, Delivery)**.**
Let $p, q, r \in P$ be distinct packets, i. e., $p \neq q, p \neq r, q \neq r$. Then

- (p, q, p) is not a dispatch (packets are not unique);
- (p, q) is a dispatch and it holds for its delivery set $D_{(p,q)}$:

$$\begin{array}{rl}
() \in D_{(p,q)} & \text{empty delivery (all packets lost)} \\
(p, q) \in D_{(p,q)} & \text{normal delivery (no loss/duplication/reordering)} \\
(q, q) \in D_{(p,q)} & \text{loss and duplication, no reordering} \\
(q, p, q) \in D_{(p,q)} & \text{no loss, duplication and reordering} \\
(q, r, q) \notin D_{(p,q)} & \text{packet } r \text{ not in } (p, q)
\end{array}$$

As mentioned in Remark 1, $D_{(p)_n}$ represents the set of deliveries for network with unbounded capacity. Real networks have a finite capacity which restricts reordering: Assuming that packets are not reordered in the (socket of the) sender and receiver but only while they are on the network, only groups of packets are reordered which are at the network at the same time. For instance, in the scenario of Figure 3, packet p_c may be delivered before packet p_1 but packet p_{c+1} cannot be delivered before packet p_1 unless some packet in between is lost. Generally, after delivering packet p_i, at most $c - 1$ packets can be delivered which have been dispatched before p_i. This is because packets dispatched before p_i but delivered after p_i must be, together with p_i, on the network at the time p_i is about to be delivered. There cannot be more than $c - 1$ such packets because the network can hold at most c packets.

Formally, for a packet q_i of a delivery $(q)_m \in D_{(p)_n}$, let $L_{(q)_m,i} = \{q_j \mid j > i \wedge q_j < q_i\}$ denote the set of *late packets* in $(q)_m$ w. r. t. q_i, i. e., packets $q_j < q_i$ that have been dispatched before q_i but delivered after q_i (see dispatch order in Definition 2).

Just before q_i is delivered, all packets in $L_{(q)_m,i}$ and q_i must be on the network with capacity c, which implies that $|L_{(q)_m,i} \cup \{q_i\}| \leq c$. Since $q_i \notin L_{(q)_m,i}$ we get $|L_{(q)_m,q_i}| < c$.

Definition 3 formalizes the effect of the network capacity on the set of deliveries.

Definition 3 (Capacity-Bounded Deliveries)**.**

Let $D_{(p)_n}$ be the set of deliveries of a dispatch $(p)_n$. Let $c \in \mathbb{N}_1$ be the maximum number of packets the network can hold at a given time.

For a delivery $(q)_m \in D_{(p)_n}$ and $i \in [1, m-1]$, let $L_{(q)_m,i} =_{def} \{q_j \mid j > i \wedge q_j < q_i\}$ denote the set of *late packets* in $(q)_m$ w. r. t. q_i. Then

$$D_{(p)_n,c} =_{def} \{(q)_m \in D_{(p)_n} \mid \forall i \in [1, m-1] : |L_{(q)_m,i}| < c\}$$

is the *capacity c bounded delivery set* of $(p)_n$.

Remark 2 (Capacity-Bounded Deliveries).

$D_{(p)_n,c}$ represents the set of packet sequences that may be received when sending n UDP packets $(p)_n$ on a network with capacity c. It is still infinite for $n > 0$ since the number of duplications and thus the length of the received packet sequences remains unconstrained by the network capacity.

In Definition 3, we model the capacity of the network in terms of UDP packets, i. e. , we abstract from the fragmentation of datagram packets into smaller IP packets or network frames. Moreover, we assume that the capacity of the network is sufficiently large to hold at least one complete datagram packet.

Note that $D_{(p)_n,c} = D_{(p)_n}$ if $n \leq c$ because, by Definition 2 it holds for all $(q)_m \in D_{(p)_n}$ and $i \in [1, m-1]$: $L_{(q)_m,i} \subseteq \{p_j \mid j \in [1, n]\} \setminus \{q_i\}$ and thus $|L_{(q)_m,i}| < n$ which implies that $|L_{(q)_m,i}| < c$ for all $i \in [1, m-1]$. This is consistent with the intuition that only networks with a capacity smaller than the number of subsequently sent packets restrict the reordering of packets.

Note further that a network with capacity 1 does not permit reordering because $D_{(p)_n,1} = \{(q)_m \in D_{(p)n} \mid \forall i \in [1, m-1] : L_{(q)_m,i} = \emptyset\}$. I. e. , after the delivery of a packet q_i no packets are delivered that have been dispatched before q_i and it holds: $i < j$ implies $q_i \leq q_j$ for all $(q)_m \in D_{(p)_n,1}$ and $i, j \in [1, m]$.

As a direct consequence of Definition 3 we get the following monotonicity for all $c, c' \in \mathbb{N}$:
$c < c' \Rightarrow D_{(p)_n,c} \subseteq D_{(p)_n,c'}$
For $c, c' \in [1, n]$ we get a strict monotonicity: $c < c' \Leftrightarrow D_{(p)_n,c} \subset D_{(p)_n,c'}$. I. e. , networks with lower capacity allow less cases of reordering as illustrated in the subsequent example.

Example 3 (Capacity-Bounded Deliveries)**.**

Consider three packets $p, q, r \in P$ and a capacity $c \geq 3$. Then the following deliveries are (not) contained in the capacity-bounded delivery sets $D_{(p,q,r),c}$ (equal to $D_{(p,q,r)}$), $D_{(p,q,r),2}$, and $D_{(p,q,r),1}$:

	$D_{(p,q,r),c}$	$D_{(p,q,r),2}$	$, D_{(p,q,r),1}$
(p, q, r)	$\in$	$\in$	$\in$
(p, p, r)	$\in$	$\in$	$\in$
(r, p, r)	$\in$	$\in$	$\notin$
(r, p, q)	$\in$	$\notin$	$\notin$

$(r, p, r) \notin D_{(p,q,r),1}$ and $(r, p, q) \notin D_{(p,q,r),1}$ because a network with capacity 1 does not allow reordering (Remark 2). Hence packet p cannot be delivered after the first delivery of the later packet r.

$(r, p, r) \in D_{(p,q,r),2}$ because the following sequence of events realizes the delivery (r, p, r) of dispatch (p, q, r) on a network with capacity 2:

Event	**Packets on the network**	**Delivered packets**
-	$\emptyset$	$()$
p dispatched	$\{p\}$	$()$
q dispatched	$\{p, q\}$	$()$
q lost	$\{p\}$	$()$
r dispatched	$\{p, r\}$	$()$
duplicate of r delivered	$\{p, r\}$	(r)
p delivered	$\{r\}$	(r, p)
r delivered	$\emptyset$	(r, p, r)

Conversely, the delivery (r, p, q) requires a network with capacity $c \geq 3$ as demonstrated by its generating event sequence:

Event	Packets on the network	Delivered packets
-	$\emptyset$	$()$
p dispatched	$\{p\}$	$()$
q dispatched	$\{p, q\}$	$()$
r dispatched	$\{p, q, r\}$	$()$
r delivered	$\{p, q\}$	(r)
p delivered	$\{q\}$	(r, p)
q delivered	$\emptyset$	(r, p, q)

In the sequel, we identify and characterize finite subsets of the delivery set $D_{(p)_n}$ of a non-empty dispatch $(p)_n$, suitable for testing distributed applications.

Finite subsets of $D_{(p)_n}$ can be obtained, for instance, by

- setting an upper bound on the length of deliveries (global constraint);
- restricting the number of times each dispatched packet may appear in a delivery (local constraint).

When testing an SUT, the number of packets that will be transmitted in a given test run, is often not known in advance, making it difficult to decide and enforce a limit on the length of deliveries. Hence we opted for constraining the number of generated duplicates of each individual packet instead.

Therefor, we introduce a finite set $M \subset \mathbb{N}$ of *multiplication choices* which restricts the number of times each dispatched packet may appear in each individual delivery as follows:

Definition 4 (Multiplication-Bounded Deliveries)**.**

Let $M \subset \mathbb{N}$ be a non-empty, finite set of natural numbers, called *multiplication choices*. Let $D_{(p)_n}$ be the delivery set of dispatch $(p)_n$. Then

$$D_{(p)_n,M} =_{def} \{(q)_m \in D_{(p)_n} \mid \forall i \in [1, n] : |\{j \in [1, m] \mid q_j = p_i\}| \in M\}$$

is the set of *multiplication M bounded deliveries*.

Remark 3 (Multiplication-Bounded Deliveries).

$D_{(p)_n,M}$ is defined in such a way that for each delivery $(q)_m \in D_{(p)_n,M}$ the number of times a dispatched packet p_i appears in $(q)_m$ is contained in M. Dependent on the multiplication choices M, different cases of packet loss and duplication are obtained. For instance, $D_{(p)_n,\{1\}}$ is the set of permutations of $(p)_n$, representing all reorderings without packet loss and duplications. $D_{(p)_n,M}$ contains deliveries with packet loss if and only if $0 \in M$, and it contains cases of packet duplication if and only if M contains an element larger than 1. With $M = [l, u]$ we obtain the set of deliveries where each dispatched packet is delivered at least l and at most u times.

In the shortest deliveries $(q)_m \in D_{(p)_n,M}$, each dispatched packet p_i appears $min(M)$ times while in the longest deliveries, each packet p_i appears $max(M)$ times. Hence we get for the length m of each delivery in $(q)_m \in D_{(p)_n,M} : n \cdot min(M) \leq m \leq n \cdot max(M)$.

Since by Definition 2, each delivered packet q_i is contained in the finite set $\{p_i \mid i \in [1, n]\}$, we get with $l = n \cdot min(M), u = n \cdot max(M)$, and $\bar{P} = \{p_i \mid i \in [1, n]\} : D_{(p)_n,M} \subseteq \bigcup_{m \in [l,u]} \bar{P}^m$. This shows that $D_{(p)_n,M}$ is finite.

Similar to the case of capacity-bounded deliveries, we get the following strict monotonicity as direct consequence of Definition 4 for all $M, M' \subset \mathbb{N}$ and $n \in \mathbb{N}_1 : M \subset M' \Leftrightarrow D_{(p)_n,M} \subset D_{(p)_n,M'}$. I. e., the set of multiplication-bounded deliveries $D_{(p)_n,M}$ can be extended/reduced by adding/removing elements from the multiplication choice set M.

Example 4 (Multiplication-Bounded Deliveries)**.**

Let $p \neq q \in P$ be distinct packets. Then

$D_{(p,q),\{1\}}$	$= \{(p, q), (q, p)\}$	no packet loss and duplication
$D_{(p,q),\{0,1\}}$	$= \{(), (p), (q), (p, q), (q, p)\}$	packet loss but no duplication
$D_{(p,q),\{1,2\}}$	$= \{(p, q), (q, p), (p, p, q), ..., (q, q, p, p)\}$	no packet loss but duplication at most once
$D_{(p,q),\{0,1,2\}}$	$= \{(), (p), (q), (p, q), ..., (q, q, p, p)\}$	as listed in Example 1

To control both duplication and reordering, multiplication-bounded delivery sets are combined with capacity-bounded delivery sets:

Definition 5 (Multiplication- and Capacity-Bounded Deliveries)**.**
Let $(p)_n$ be a dispatch, $M \subset \mathbb{N}$ a non-empty, finite set of multiplication choices, and $c \in \mathbb{N}_1$ a network capacity. Then

$$D_{(p)_n,M,c} =_{def} D_{(p)_n,M} \cap D_{(p)_n,c}$$

is the set of *multiplication M and capacity c bounded deliveries.*

Remark 4 (Multiplication- and Capacity-Bounded Deliveries)*.*
The set of multiplication- and capacity-bounded deliveries $D_{(p)_n,M,c}$ represents the set of packet sequences a receiver may obtain when sending to it n UDP packets on a network which can hold at most c packets and delivers each packet a number of times that is contained in M.

Since $D_{(p)_n,M}$ is finite (Remark 3), also $D_{(p)_n,M,c}$ is finite and hence can be enumerated as test input.

Combining capacity restriction c with a multiplication restriction M enables the representation of reliable transmissions without packet loss, duplication, and reordering, by setting $c = 1$ and $M = \{1\}$. Here the only possible delivery is equal to its dispatch: $D_{(p)_n,\{1\},1} = \{(p)_n\}$.

Furthermore, with $c > 1$ and $M = \{1\}$ we obtain the cases with capacity-bounded reordering but without packet loss and duplication, as illustrated in the following example.

Example 5 (Capacity-bounded Reordering without Packet Loss and Duplication)**.**
For the dispatch (p, q, r) of the three distinct packets $(p, q, r) \in P$ and the multiplication choice set $\{1\}$ we get:

$$\begin{aligned}
D_{(p,q,r),\{1\},1} &= \{(p, q, r)\} \\
D_{(p,q,r),\{1\},2} &= \{(p, q, r), (p, r, q), (q, p, r), (q, r, p)\} \\
D_{(p,q,r),\{1\},3} &= \{(p, q, r), (p, r, q), (q, p, r), (q, r, p), (r, p, q), (r, q, p)\}
\end{aligned}$$

$D_{(p,q,r),\{1\},3}$ is equal to $D_{(p,q,r),\{1\}}$ (Remark 2), which is the set of permutations of the three packets p, q, r (Remark 3).

In the case of network capacity 2, $D_{(p,q,r),\{1\},2}$ does not contain the sequences where the last packet r is delivered first, because this would require the network to hold all three packets p, q, r at the time r is delivered.

3.2 Generating Non-Deterministic UDP Transmissions with JPF

Our goal is to generate the set of multiplication- and capacity-bounded deliveries of packet sequences sent by the SUT to a remote peer or vice versa, w. r. t. a multiplication choice set M and an assumed network capacity c, as defined in Definition 5. The parameters M and c are chosen according to the properties and test goal of the SUT at hand, and do not necessarily correspond with properties of the real network. Moreover, we generate deliveries with packet perturbation on the fly, without fixing a number n of packets to be transmitted, using JPF's choice generator mechanism.

3.2.1 General Architecture

Packet perturbation is implemented in the JPF extension net-iocache [22] which intercepts, similar to a proxy, the communication between the SUT and the remote peers (Figure 4). When the SUT calls method `send` of a datagram socket to trigger a packet transmission to a remote peer, the transmission of that packet is delegated to net-iocache. Calls of method `receive` are handled similarly. This way, incoming and outgoing packet sequences can be manipulated by net-iocache for generating the desired cases of packet loss, duplication and reordering. 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.

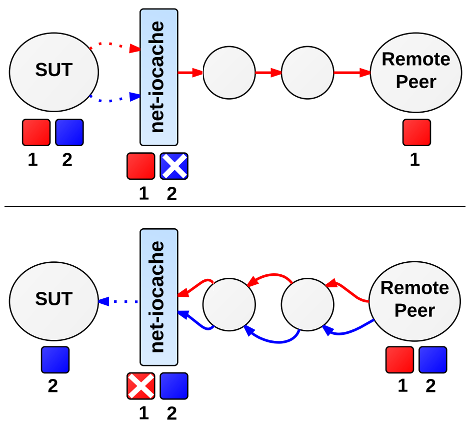

Figure 4: net-iocache generates packet loss by dropping outgoing and incoming packets.

In our setting, the SUT and remote peers run in a local, controlled test environment where UDP I/O behaves reliably without packet loss, duplication, and reordering. All instances of packet perturbation are injected by net-iocache, giving us full control over the type and amount of generated packet perturbation and ensuring the reproducibility of test runs.

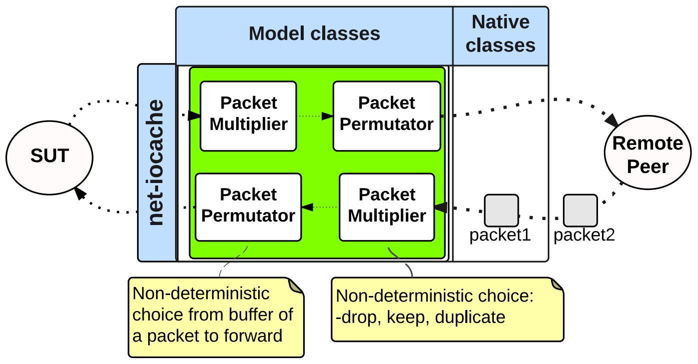

Figure 5: Systematic generation of packet perturbation in model class `DatagramSocket` of net-iocache.

Figure 5 depicts the components involved in systematic simulation of unreliable UDP behavior. As mentioned in Section 2.1, JPF uses *model classes* to support Java API classes with native code. We implemented our solution in the model class of `DatagramSocket` in the form of two modules: the `PacketMultiplier` module, and the `PacketPermutator` module (Figure 5 center). The `PacketMultiplier` generates cases of packet loss and duplication according to a given set of multiplication choices M. The `PacketPermutator` uses a buffer to change the order of datagram packets and thus simulates a reordering network with capacity c. The multiplier and permutator modules perform non-deterministic choices when deciding the number of instances and order of packets to be forwarded to the SUT or remote peer, making use of JPF's choice generator mechanism.

3.2.2 Algorithms for Generating Packet Perturbation

Algorithms 2, 3, 5, and 6, describe how the packet multiplier and permutator are realized in send and receive direction. 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, it is forwarded to the `PacketMultiplier` (Algorithm 1, line 2). `PacketMultiplier` determines the number of instances to be generated from the passed

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

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

```
Function packetMultiplierPush(packet)
    $counter_{\text{packet}} \leftarrow$ chooseFrom(MULTIPLYCHOICES);
    if $counter_{\text{packet}} > 0$ then
        packetPermutatorPush(packet);
```

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

```
Function packetPermutatorPush(packetIn)
    sendBuffer $\leftarrow$ sendBuffer $\cup$ {packetIn};
    flushSendBuffer($BUFFERLIMIT - 1$);

Function flushSendBuffer(newSize)
    while |sendBuffer| $> newSize$ do
        packetOut $\leftarrow$ chooseFrom(sendBuffer);
        sendToNetwork(packetOut);
        $counter_{\text{packetOut}} \leftarrow counter_{\text{packetOut}} - 1$;
        if $counter_{\text{packetOut}} = 0$ then
            sendBuffer $\leftarrow$ sendBuffer\{packetOut};
```

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

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

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

```
Function packetPermutatorPull(packetOut)
    while receiveBuffer= $\emptyset \vee$ (|receiveBuffer| $< BUFFERLIMIT \wedge \neg$timeout) do
        packetMultiplierPull(packetIn);
        receiveBuffer $\leftarrow$ receiveBuffer $\cup$ {packetIn};
    packetOut $\leftarrow$ chooseFrom(receiveBuffer);
    $counter_{\text{packetOut}} \leftarrow counter_{\text{packetOut}} - 1$;
    if $counter_{\text{packetOut}} = 0$ then
        receiveBuffer $\leftarrow$ receiveBuffer\{packetOut};
```

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

```
Function packetMultiplierPull(packet)
    flushSendBuffer(0);
    receiveFromNetwork(packet);
    $counter_{\text{packet}} \leftarrow$ chooseFrom(MULTIPLYCHOICES);
    if $counter_{\text{packet}} < 1$ then
        packetMultiplierPull(packet);
```

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

packet, based on the user-configured list `MULTIPLYCHOICES` which corresponds to the set M of multiplication choices in Definition 4. For instance, if `MULTIPLYCHOICES` is set to "`0,1`", function **chooseFrom** in 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` which is the assumed network capacity c as in Definition 3. 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`, according to Definition 5.

3.2.3 On-the-fly Generation of Non-Deterministic I/O with JPF's Choice Generators

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 finite set of options. 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. In the first execution branch, `i` is assigned the value of `min`. When the execution branch is fully explored, JPF backtracks the SUT to the state before executing the assignment of `min` to `i`. After that, it assigns the next open option `min+1` to `i` and executes the rest of the SUT. This is continued until the SUT is executed for all options `min, min+1, ..., max`.

This mechanism is used in `chooseFrom` to let JPF generate and explore the different possibilities of packet loss, duplication, and reordering within the constraints of `MULTIPLYCHOICES` and `BUFFERLIMIT`.

3.2.4 Configuration of Multiplication Choices and Reordering Buffer

Considering all forms of packet perturbation, the number of non-deterministic I/O outcomes and executing branches generated by the packet multiplier and permutator modules may quickly grow large. 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 the user-defined setting and dynamic adaption of the parameters `MULTIPLYCHOICES` and `BUFFERLIMIT` that determine the kind and number of generated cases according to Definition 5.

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

```
jpf-net-iocache.UDP.packetMultiplicationChoices=<MULTIPLYCHOICES>
jpf-net-iocache.UDP.reorderWindowSize=<BUFFERLIMIT>
```

For instance, the setting of the `BUFFERLIMIT` to 1 disables reordering of packets (Remark 2). 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). For `jpf-net-iocache.UDP.packetMultiplicationChoices=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. Similar properties allow the configuration of the parameters `MULTIPLYCHOICES` and `BUFFERLIMIT` for methods `receive` and `send` separately.

JPF's framework supports the static setting of properties in a configuration file for each SUT, as well as their dynamic manipulation by inserting instructions inside the SUT's source code using the API method `Verify.setProperties`. 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.

3.2.5 Limitations

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.

In many applications some packets are sent or received in response to previous network traffic [3]. We do not reorder packets across such logical message boundaries, as this would produce communications that are not possible in reality. To ensure that responses to previously sent packets are transmitted and can be received by the SUT, line 2 of Algorithm 6 forwards all packets to the network that have been kept in the send buffer for reordering. Emptying the send buffer on receive, however, 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` in Algorithm 5 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.

For checking real time applications, it may be useful to manipulate time-related communication properties such as delay and jitter [13]. In contrast to network emulators such as netem [23], net-iocache does not offer means to control such properties. The SUT, executed by JPF, runs slower than the remote peers running on a host JVM because of the model checking overhead (see runtime results in Sections 4.1 and 4.2). This introduces hard to control delays which may have an impact on the communication between the SUT and its remote peers. To check how the SUT reacts on large delays, we consider the extension of net-iocache towards the configurable injection of timeouts on I/O operations.

An additional limitation of our approach is that the buffering of incoming and outgoing packets for reordering may mask some I/O exceptions. When holding back a sent packet in the send buffer, we take care to throw predictable exceptions such as a "socket exception" in the case the socket is closed (omitted in Algorithm 3 for brevity). However, unpredictable I/O exceptions caused, for instance, by network failures are missed.

3.3 Implementation of 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` [38] 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` [22, 3] 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.

3.3.1 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.

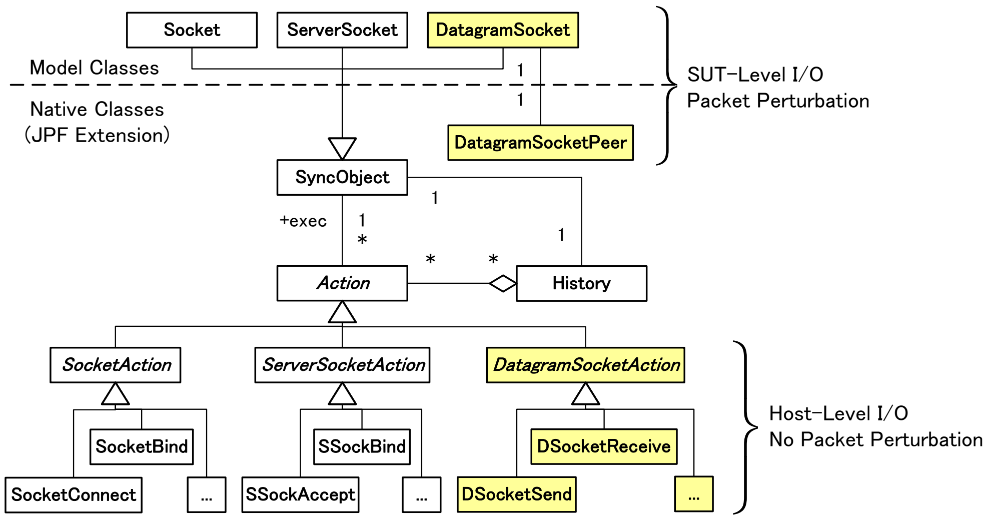

Figure 6: Conceptual model of net-iocache v2 (abstract classes in italics).

Figure 6 depicts the conceptual model of net-iocache v2, including its extension towards UDP. The upper part of Figure 6 contains JPF *model classes* for the covered Java network API which are the classes `Socket`, `ServerSocket`, and `DatagramSocket`. Each model class interacts with a *native peer* class such as `DatagramSocketPeer` in Figure 6 to execute code in the host environment. While model classes are executed by JPF, native peer classes and other native classes are executed by the host Java Virtual Machine as part of JPF.

Model class `DatagramSocket` and its native peer `DatagramSocketPeer` implement the algorithms for packet perturbation as described in Section 3.2.2.

Native class `SyncObject` (Figure 6, center) is a generic host-level representation of network model classes. Similarly to `jpf-nhandler` [38], 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 6, center). For instance, a single invocation of `dsocket.send(packet)` is represented as an `Action` object, associated with sync object `dsocket`.

Subclasses of `Action` such as `SocketConnect` in the case of TCP sockets, or `DSocketSend` in the case of UDP sockets (Figure 6 bottom), execute a network I/O operation in the host environment and capture its result. In contrast to SUT-level UDP I/O, host-level I/O is not subject to packet perturbation in the form of injected packet loss, duplication, or reordering.

When JPF backtracks the SUT, the reverted state of objects of model classes such as `DatagramSocket` may differ from the state of their corresponding host-level objects which are 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 6, 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.

3.3.2 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 model class and native peer class for `DatagramSocket` (Figure 6, top) and implementing the supported actions as subclasses of abstract class `Action` (Figure 6, 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 in the host environment, 4) capturing the execution results such as return values or thrown exceptions.

	`DatagramSocket` (UDP)		`Socket` (TCP)	
	classes	**lines**	**classes**	**lines**
model classes	1	183	4	93
native peer classes	1	245	4	64
action subclasses	9	262	13	313
utility classes	3	152	2	129
total	**14**	**842**	**23**	**599**

Table 1: Size of code for UDP and TCP sockets in net-iocache v2 alpha rev 862.

Table 1 summarizes the number of classes and lines of code for the implementation of UDP support in net-ioache as compared to the code for TCP sockets. Lines in the source code without statements are excluded. In addition to the model classes, their native peer classes, and subclasses of class `Action` as described above, utility classes for the representation of packet content etc. have been provided. While the code for implementing actions and for utility functionality has a similar size for UDP and TCP, the `DatagramSocket` model class and its native peer are more complex than the TCP counterparts because of the additional code for generating packet perturbation.

3.3.3 Applying JPF/net-iocache — the User's Perspective

Using JPF and net-iocache for the testing of distributed Java applications is similar to classical testing: The user provides a number of test cases in the form of 1) selected input data, 2) definitions of the expected output and/or assertions of desired properties, 3) scripts or Java code for running the test cases. To reduce the effort for the manual construction of test cases, JPF can also be combined with automatic testing techniques such as model-based testing [4, 6].

The manually provided or automatically generated test cases can be executed using either a standard JVM (classical testing) or JPF (software model checking). JPF executes the SUT for all outcomes of non-deterministic operations such as thread scheduling choices, random numbers, or unreliable I/O, resulting in a higher coverage of program states. The standard JVM is preferable for evaluating the *real-time* behavior, and the *performance* and robustness of the system in scenarios with large input and heavy load, because JPF slows down the execution and the number of execution branches often grows exponentially in the size of the input data and the number of threads.

Using custom choice generators of JPF's verification API, different choices of input data can be explored in the same way as outcomes of non-deterministic operations. However, this is limited to input parameters with a comparably small finite domain and is not practical for large data domains such as floating point values, or compound data structures such as lists and maps. For verifying large data domains, other formal methods such as symbolic model checking [27], satisfiability modulo theories solving [32], or theorem proofing [24] are more appropriate.

4 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.

4.1 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 [3]. Figure 7 depicts the components of the application.

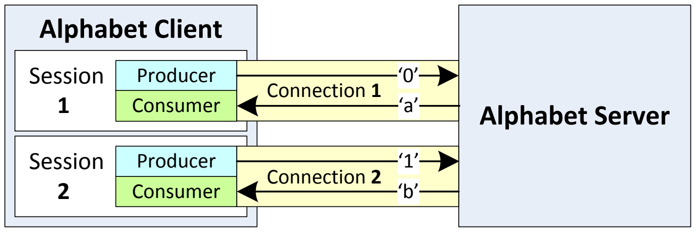

Figure 7: Components of the alphabet client/server application.

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. Therefor, 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.

#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 2: TCP and UDP alphabet client, 3 connections and 1–3 requests, on the Oracle JVM.

Table 2 shows the results of executing the TCP and UDP-based alphabet client on the standard Oracle JVM (32 Bit Java RTE 1.8.0_31-b13 / Java HotSpot Server VM 25.31-b07). Both the alphabet client and server were executed on the same 8 core Mac Pro workstation with 24 GB of memory running Ubuntu 14.04.1 LTS.

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 2 includes the invocation of the JVM. The total number of I/O operations (column #I/O in Table 2) comprises operations for creating/connecting/closing sockets and transmitting messages.

#Req	Protocol	iocache version	Pkt loss	Assertion	Fault found	Time [h:mm:ss]	Heap [MB]	#I/O	#Transitions
1	TCP	v1	–	active	–	0:00:39	720	9,902	310,908
		v2				0:00:06	77	1,158	2,143
	UDP	v2	no	active	no	0:00:03	77	1,158	2,146
			yes			0:00:13	77	2,406	5,734
2	TCP	v1	–	active	–	0:06:36	1009	174,356	2,993,066
		v2				0:00:25	77	5,190	7,807
	UDP	v2	no	active	no	0:00:19	110	5,190	7,810
			yes		yes	<0:00:01	77	23	49
				inactive	no	0:05:40	77	42,855	90,493
3	TCP	v1	–	active	–	0:47:19	**2,044**	**1,199,905**	**16,156,279**
		v2				0:01:12	110	17,346	23,440
	UDP	v2	no	active	no	0:01:00	77	17,346	23,443
			yes		yes	<0:00:01	61	29	61
			yes	inactive	no	**1:54:18**	110	828,501	1,637,530

Table 3: TCP and UDP alphabet client, 3 connections and 1–3 requests, on JPF/net-iocache (largest number of column in bold face).

Table 3 shows the results of executing the TCP and UDP alphabet client on JPF v8.0 rev 2 with `net-iocache` v1 rev 813 (no UDP support [22]) and v2 alpha rev 862 (with UDP support as described in Section 3.3). The UDP test cases were executed both without and with simulation of packet loss using the configuration property `jpf-net-iocache.UDP.packetMultiplicationChoices` (see Section 3.2.4). 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 3). 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 2), 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 [4].

- Exhaustive packet loss simulation is expensive if no error is detected (see, e. g., last row of Table 3). 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 3.3). 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 3). We suppose that establishing and terminating connections, which is missing in UDP, takes significant extra time.

As compared to an earlier revision of net-iocache v2 reported in previous work [37], the runtime is reduced between 30% for the cases without packet loss generation and 80% for cases with packet loss.

The performance improvement has been achieved mainly by moving expensive parts of generating packet perturbation from the `DatagramSocket` model class to its native peer class. Native peer code executes quicker and introduces less states because it runs on the host JVM instead of JPF (see Section 3.3).

Moreover, net-iocache v1 performs worse on JPF v8 than on JPF v7 (cf. previous work [37]). We suppose that JPF v8 explores more possible interactions between threads than JPF v7, which leads to a larger state space in the case of net-iocache v1. In contrast, the higher precision of JPF v8 did not cause a larger state space in the case of net-iocache v2 because the code in model classes, that may introduce dependencies between threads, is reduced to a minimum in net-iocache v2.

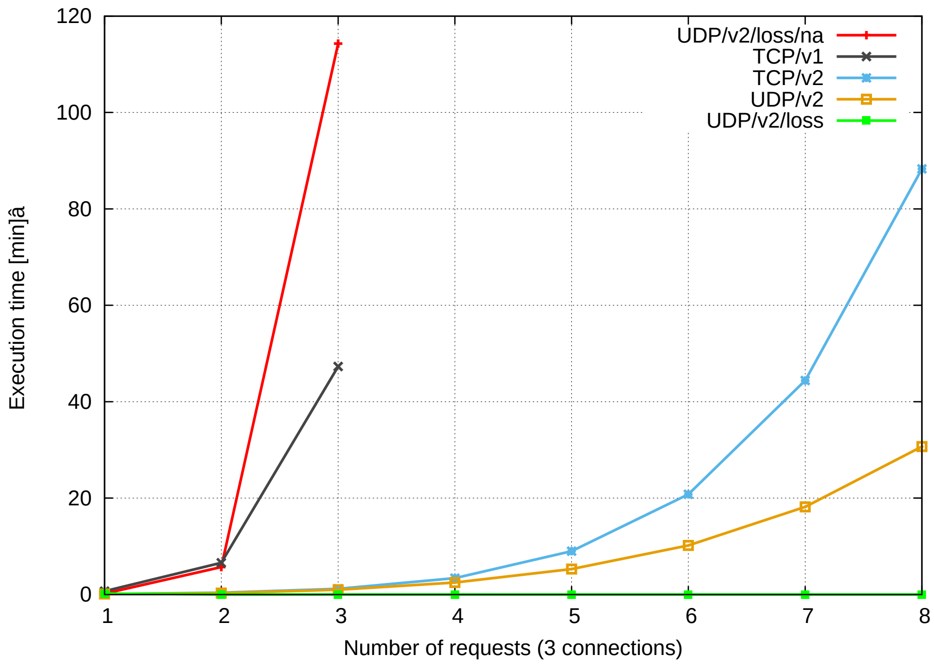

Figure 8: Runtime behavior of net-iocache on different variants of the alphabet client. The first case (red line) refers to UDP on net-iocache v2 with packet loss enabled and <u>n</u>o <u>a</u>ssertions.

Figure 8 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 UDP/v2/loss, the runtime of all configurations increases more than linearly 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 8). Conversely, the runtime of UDP/v2/loss (with assertion) is largest for one request (13 sec) 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 3 requests (TCP/v1 in Figure 8). net-iocache v2, however, succeeds in verifying 8 requests on each connection using only 77 MB of memory, in 1 hour and 28 minutes (TCP/v2 in Figure 8).

- 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 2.9 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.

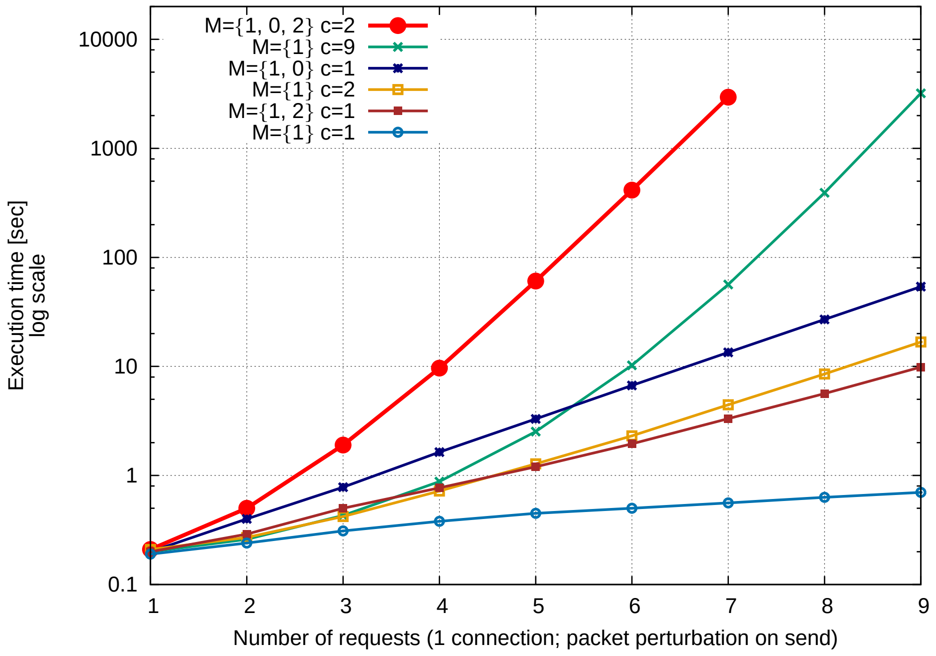

Figure 9: Runtime behavior of net-iocache on alphabet client with different configurations of generated packet perturbation (M: multiplication choices; c: network capacity).

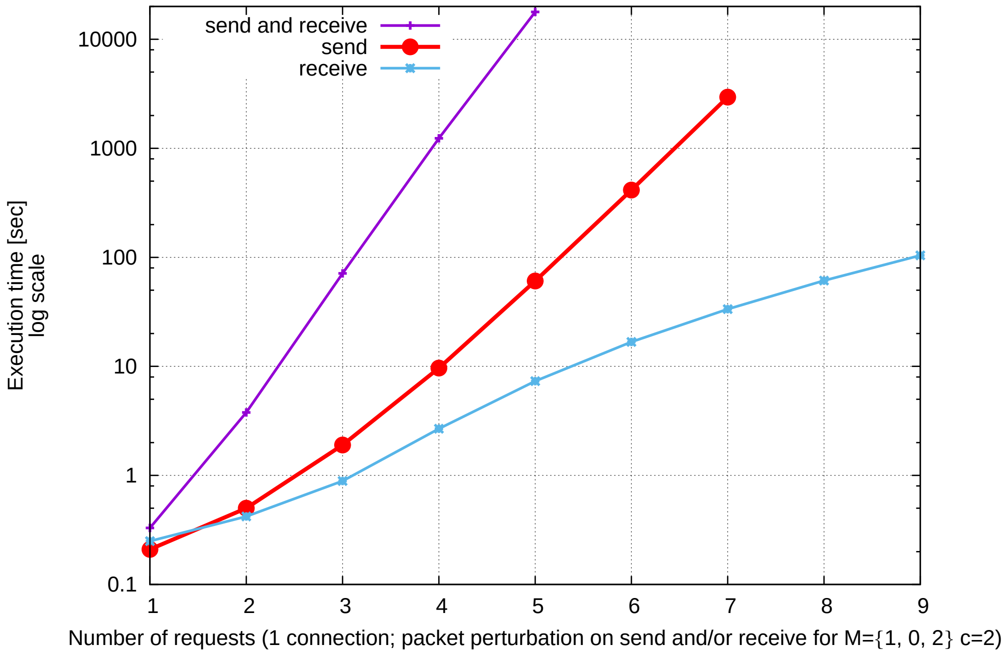

Figure 10: Runtime comparison of alphabet client with packet perturbation on send, receive, and both directions.

Figure 9 shows the runtime of model checking the alphabet client sending up to 9 requests on a single connection, when using different configurations of packet perturbation. To let JPF explore the entire state space of the SUT, the assertion on the received responses is deactivated. All runtimes in Figure 9 have been obtained by injecting different forms of packet perturbation on outgoing packets only (send direction). If packet perturbation is disabled ($M = \{1\} \quad c = 1$ in Figure 9, cf. Remark 4), all cases execute in less than one second. The cases with injected packet duplication ($M = \{1, 2\} \quad c = 1$), capacity 2 bounded reordering ($M = \{1\} \quad c = 2$), and packet loss ($M = \{1, 0\} \quad c = 1$) show a moderate exponential growth in runtime. Also the combined case of packet loss, duplication, and bounded reordering ($M = \{1, 0, 2\} \quad c = 2$) appears to grow exponentially but with a larger growth rate. In contrast, unbounded reordering ($M = \{1\} \quad c = 9$) grows more than exponentially. This is consistent with the growths of the number of permutations which is factorial.

Figure 10 shows the impact on the runtime performance when packet perturbation is generated for outgoing packets only (send), incoming packets only (receive), or both directions (send and receive). Here, we fix the configuration to combined generation of packet loss, duplication, and capacity 2 bounded reordering ($M = \{1, 0, 2\} \quad c = 2$), which is the most expensive case in Figure 9. We observe that applying packet perturbation to both outgoing and incoming packets increases the runtime significantly. When both outgoing requests and the respective responses are duplicated, the number of responses is up to 4 times higher than without packet duplication. This in turn increases the number of cases resulting from reordering.

Furthermore, we observe that packet perturbation on receive is less expensive than on send. This is a consequence of the simplicity of the SUT: The alphabet client does not process the received responses in any form, leaving its state unchanged regardless of the received packet. The increasing likelihood, that JPF detects a "visited" state and prunes the state space, seems to keep the growth rate down below exponential growth in the case of packet permutation on receive.

4.2 Verifying a UDP-based File Transfer Protocol

We choose as a demonstrator case a client/server application that uses UDP for the transfer of files from the server to the client. This is similar to the Starbust Multicast File Transfer Protocol (MFTP), that uses UDP for distribution of files [34].

```
1 Function main()
      Input: fileNames: identifiers of files to be retrieved from the server
      Input: retries: number of maximum consecutive resend requests for a file
2     for fileName $\in$ fileNames do
3         filePackets $\leftarrow$ getNumberOfPackets(fileName);
4         data $\leftarrow$ getFile(fileName, filePackets, retries);
5         assert isContentEqual(fileName, data);
```

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

Algorithm 7 shows the main method of the file transfer client. Using a TCP connection, the client first requests the number of packets necessary to transfer a specific file from the server (line 3 of Algorithm 7). For simplicity, we assume that all requested files exists in the server's repository. After that, the client calls `getFile`, shown in Algorithm 8, to initiate the file transfer using UDP. Function `getFile` is supposed to ensure the validity of the returned packet array by compensating for packet loss, duplication, and reordering. For testing purposes, we compare the contents of the received data against a local reference copy of the file (line 5 in Algorithm 7).

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 11 shows correctly handled cases of packet duplication and out-of-order delivery. 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 12).

```
Function getFile(fName, fPackets, retries)
    data ← new Packet[fPackets];
    missingPackets ← {0, 1, ..., fPackets − 1};
    consecutiveTimeouts ← 0;
    send(request(fName, missingPackets));
    while missingPackets ≠ ∅ do
        receive(packet);
        if timeout then
            consecutiveTimeouts ← consecutiveTimeouts + 1 ;
            if consecutiveTimeouts > retries then
                return ABORTED;
            send(request(fName, missingPackets));
        else
            consecutiveTimeouts ← 0;
            index ← getSequenceNumber(packet);
            if index ∈ missingPackets then
                data[index] ← packet;
                missingPackets ← missingPackets\{index};
    return data;
```

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

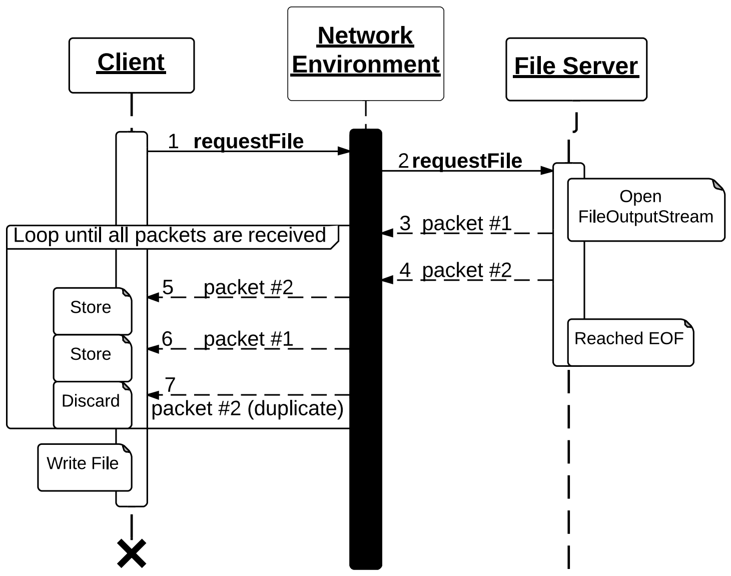

Figure 11: Scenario where the client compensates for packet reordering and duplication.

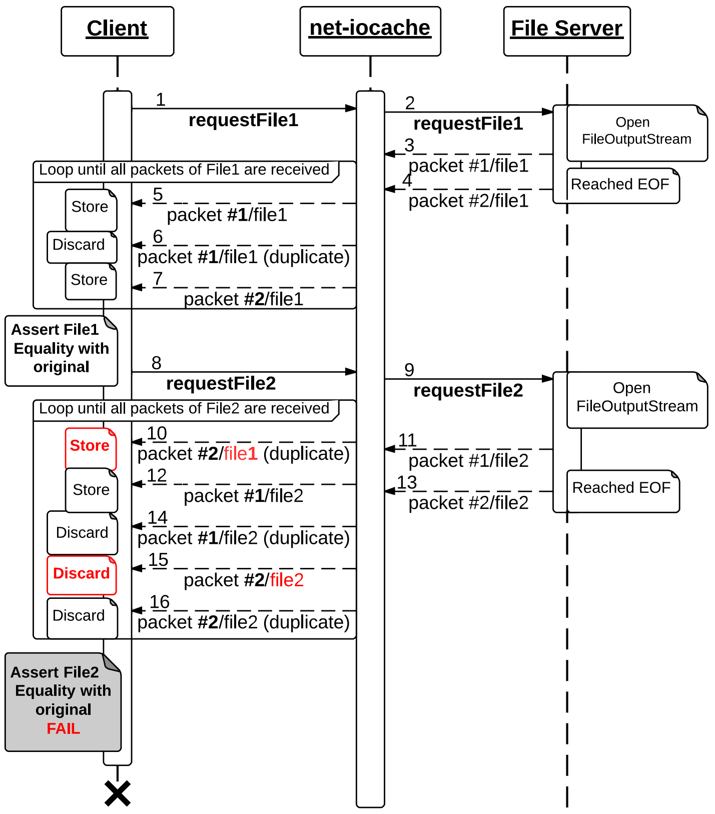

Figure 12: I/O sequence generated by net-iocache revealing a case of an incorrectly handled packet.

JPF detects the defect in the client when packet duplication is enabled using the following configuration (see Section 3.2.4):

```
jpf-net-iocache.UDP.packetMultiplicationChoices=2
jpf-net-iocache.UDP.reorderWindowSize=1
```

For two files f_i and f_{i+1} subsequently requested by the client, a duplicate of the last packet of file f_i is not discarded if 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 16 of Algorithm 8, and stores it in the data array for file f_{i+1}.

This defect can be corrected by including a file ID in each packet the server sends, allowing the client to distinguish packets of different files, and replacing line 16 of Algorithm 8 with the following two lines:

```
$\textbf{fileId} \leftarrow \textbf{getFileId(packet)}$;
$\textbf{if}(\mathit{index} \in \mathit{missingPackets}\ \textbf{and fileId} = \textbf{fName})\ \textbf{then}$
```

Table 4 presents the results of checking the file transfer client as SUT in a scenario where it downloads 2 files of 1.5 and 2.3 kB size, delivered in 2 UDP packets for the first and 3 UDP packets for the second file. The data has been collected in the same runtime environment as in Section 4.1. The two versions of the client “seeded bug” and “corrected version” were executed 1) using the standard Oracle JVM (rows 1 and 2 in Table 4) and 2) using JPF v8 with net-iocache v2. Execution on the standard JVM did not reveal the seeded bug because of UDP’s reliable behavior in the local test environment. However, when configured to generate packet duplicates, JPF detects the bug and stops the exploration of the state space. This leads to smaller numbers of execution time, I/O and transitions when the bug is detected in the seeded bug version.

VM	Multiplication choices	Reordering window	Fault seeded	Fault found	Time [s]	Heap [MB]	#I/O	#Transitions
JVM	–	–	yes	no	0.12	1	14	–
	–	–	no	no	0.12	1	14	–
JPF	1	1	yes	no	0.29	77	14	15
			no	no	0.29	77	14	15
		2	yes	no	2.38	77	22	31
			no	no	2.38	77	22	31
	2	1	yes	**yes**	0.29	77	14	18
			no	**no**	0.29	77	14	19
		2	yes	**yes**	2.34	77	16	25
			no	**no**	2.87	77	64	233
	1,2	1	yes	**yes**	0.42	77	21	45
			no	**no**	0.44	77	22	55
		2	yes	**yes**	2.96	77	59	234
			no	**no**	4.50	110	236	1253

Table 4: Results for checking the file transfer client for 2 files with 2 and 3 packets, using different configurations of packet perturbation.

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 transfer application only fails in a specific I/O schedule, stochastic approaches that randomly generate unreliable UDP I/O behavior [13, 23, 33, 40] (see Section 5.2) may miss the bug.

5 Related Work

Although the *testing* of applications that use the UDP protocol is largely documented [13, 23, 33, 40], there is little work on the *verification* of UDP-based applications. Research related to our work covers 1) the application of software model checking for verifying networked applications [2, 8, 22, 25, 38, 39, 41, 43] and actor-based systems [21], 2) classical techniques for testing UDP-based applications [13, 23, 33, 40], and 3) the analysis of packet reordering on IP networks [9, 30, 44].

5.1 Software Model Checking of Distributed Applications

This section summarizes approaches that apply software model checking for verifying networked applications at varying levels of granularity. There are two major approaches to software model checking of communicating peer processes:

Centralized approach: merging multiple peer processes into a single multi-threaded process and verifying it;

Modular approach: verifying a single process at a time while executing the remaining processes externally.

In this work, we have adopted the latter approach. Centralization-based approaches take a global view of all processes, resulting a more complete exploration of the system's state space. However, it does not scale well in the number of processes. The modular approach offers better scalability but is not complete in general [22].

5.1.1 Centralization-based Approaches

Stoller and Liu [41] introduced the concept of *centralization* 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 TCP sockets [2, 25]. A similar approach analyzes the complete state space of all processes by extending JPF itself [38, 39] rather than pre-processing the SUT.

De la Cámara et al. [10] propose a tool for the transformation of distributed C programs to the input language of the model checker SPIN [16]. The generated program model is combined with a static, manually constructed model of TCP sockets and verified by Spin. UDP support is mentioned as "future work" but has, to the best of our knowledge, not addressed by the authors since then.

The approaches above abstract from the *physical* behavior of communicating processes by simulating the exchange of messages using queues in shared memory. While there is work that adopts a centralization-based approach for the simulation of transmission failures [5] and packet perturbation [43], low-level network behavior, such as packet loss or I/O exceptions resulting from network failures, is usually not in the focus of these approaches.

Barlas and Bultan [8] propose stubs which are classes simulating the environment. The system is executed on single JVM, assuming an "ideal" network not causing packet perturbation. This approach limits the testing of UDP applications since disregarding unreliable I/O results in poor coverage. Conceptually, our approach is related to the use of stubs [8], but we execute unmodified peer processes [22], rather than user-defined stubs, to obtain the input data for the SUT.

In actor-based systems, multiple autonomous agents communicate by exchanging messages asynchronously. The distributed nature of agents in actor-based systems results in non-deterministic processing of messages since in-order delivery of messages is not guaranteed unless constrained in some actor language. Lauterburg [21] proposes a framework for systematically testing actor-based systems by using JPF for exploring different message schedules. Similar to our approach, JPF is used to explore the state space of the SUT exhaustively and to control the way messages are exchanged across the network. However, the application domain is different: Actor-based systems are targeted while our focus is on UDP-based applications.

The first work on checking UDP-based applications with JPF is centralization- and stub-based [43]. Similar to our approach, JPF's choice generator mechanism (cf. Section 3.2.3) is leveraged for exploring transmission outcomes with packet loss and reordering. Our approach is different in the following aspects:

- *Modularity*: Instead of centralizing all communicating process into a single multi-threaded process, we apply an I/O cache for the verification of one process at a time.
- *Physical* behavior: net-iocache supports network exceptions and timeouts, required for recognizing and reacting on packet loss. Such physical behavior is hard to reproduce in a centralized approach replacing network I/O with in-memory message exchange.
- *Multiple channels*: net-iocache supports the simultaneous communication on more than one UDP socket (Section 4.1), as well as combined TCP and UDP communication (Section 4.2).
- *Duplication* of packets is supported in addition to packet loss and reordering.
- *Configurability*: Based on a formal model of UDP transmissions, the perturbation generation is configurable in the following dimensions:
    1. Type of perturbation: packet loss / duplication / reordering;
    2. Range of generated duplication and reordering;
    3. Direction: send / receive / both.

Especially the modularity and configurability are key aspects for the scalability (Section 4.1) and practical applicability of our approach.

5.1.2 Modularization-based Approaches

Except net-iocache, the JPF extension jpf-nhandler [38] enables the verification of distributed systems in a modular way to some extent. However, it does not support the backtracking of sockets and the synchronization of the remote peers with a backtracked state of the SUT (see Section 3.3).

5.2 Classical Testing of UDP-based Applications

Work that tests networked applications by *stochastically* manipulating network I/O is mentioned in this section under the term *classical testing approaches of UDP-based applications.* These approaches emulate networks with different physical properties and do not generate cases of packet perturbation exhaustively.

Farchi et al. [13] propose to instrument Java bytecode related to the UDP API to introduce a layer for creating “automatic noise” which subsumes delay, packet loss, duplication, and reordering. In their approach, each packet is randomly selected to be subject to noise with an equal probability.

The network emulator netem [23] and its extensions [33, 40] 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, our approach explores outcomes of UDP-based communication systematically, resulting in higher achievable coverage and reproducibility. In addition, software model checking supports the systematic exploration of thread schedules and ranges of input data. This is particularly relevant for networked applications which are often multi-threaded to raise responsiveness and performance.

5.3 Analysis of Packet Reordering Across Networks

Work mentioned in this section is related 1) to the investigation of how frequent packet reordering and other forms of perturbation occur in real networks, and 2) to the analysis of the runtime required for reconstructing the original data from reordered packets. Both motivate the relevance of our work.

[44] reports on an experiment, monitoring the packet transmission between two endpoints across a dozen of hops on the Internet. The results suggest that the occurrence of packet reordering is rather frequent. Moreover, [9] shows that the probability of reordering is even higher in wireless networks.

[30] describes patterns of packet reordering and analyzes their impact on streaming applications w. r. t. the recovery from packet reordering under different settings of re-sequencing buffers. For describing the impact of re-sequencing, they consider two metrics: *reordering density*, defined as the distribution of displacement of packets from their original position, and *reordering buffer occupancy density* which is the degree of occupancy of a buffer used for re-sequencing out-of-order packets. They conclude that reordering has a significant negative effect on the performance of streaming applications, and should be handled by the application-level protocol with the same priority as packet loss.

6 Conclusion and Future Work

We propose the adoption of software model checking for the verification of UDP-based applications, combining the exhaustive exploration of non-deterministic thread schedules with the systematic generation of UDP I/O outcomes including packet loss, duplication, and reordering.

The approach has been implemented in a redesign of the JPF extension net-iocache, which now decouples the caching and communication mechanisms. The exhaustive generation of UDP I/O outcomes may result in a state space explosion, which we curb by allowing the user to configure the type and amount of injected packet perturbation.

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 the presented formal model of UDP transmissions. We also plan to check the compliance of our implementation of the `DatagramSocket` API with the standard Java library, using the Modbat [1] tool for model-based testing, similar to past work [4]. Finally, we consider integrating net-iocache with jpf-nas [39], which implements a centralization-based approach to the verification of distributed systems.

Acknowledgements

This work was supported by JSPS KAKENHI Grants Number 23240003, 23300004, and 26280019. The authors thank Yoriyuki Yamagata and Lei Ma for their helpful comments.

References

[1] 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)*, volume 8244 of *LNCS*, pages 112–128, Haifa, Israel, 2013. Springer.

[2] 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)*, pages 177–188, Tokyo, Japan, 2006.

[3] 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)*, volume 19 of *LNBIP*, pages 22–40, Zurich, Switzerland, 2008. Springer.

[4] Cyrille Artho, Masami Hagiya, Richard Potter, Yoshinori Tanabe, Franz Weitl, and Mitsuharu Yamamoto. Software model checking for distributed systems with selector-based, non-blocking communication. In *Proc. 28th Int. Conf. on Automated Software Engineering (ASE 2013)*, pages 169–179. IEEE, 2013.

[5] Cyrille Artho, Christian Sommer, and Shinichi 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*, pages 219–228, Washington, DC, USA, 2007. IEEE Computer Society.

[6] Cyrille Valentin Artho, Armin Biere, Masami Hagiya, Eric Platon, Martina Seidl, Yoshinori Tanabe, and Mitsuharu Yamamoto. Modbat: A model-based API tester for event-driven systems. In *Proceedings of the 9th International Haifa Verification Conference (HVC 2013)*, volume 8244 of *LNCS*, pages 112–128, Haifa, Israel, 2013. Springer.

[7] ATIS: Alliance for Telecommunications Industry Solutions. IPTV exploratory group report and recommendation to the TOPS council. Available at http://www.atis.org/tops/IEG/ATIS_IPTV_EG_RPT_final.pdf, accessed: 30 Apr 2015, 2005.

[8] Elliot D. Barlas and Tevfik Bultan. NetStub: A framework for verification of distributed Java applications. In *Proc. 22nd Int. Conf. on Automated Software Engineering (ASE 2007)*, pages 24–33, Georgia, USA, 2007.

[9] Arthur C., Girma D., Harle D., and Lehane A. The effects of packet reordering in a wireless multimedia environment. In *Wireless Communication Systems, 2004, 1st International Symposium on*, pages 453–457. IEEE, 2004.

[10] P. de la Cámara, M. M. Gallardo, P. Merino, and D. Sanán. Model checking software with well-defined APIs: The socket case. In *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems*, FMICS '05, pages 17–26, Lisbon, Portugal, 2005. ACM.

[11] R. Droms. Dynamic host configuration protocol. IETF RFC 2131, 1997. Available at http://www.ietf.org/rfc/rfc2131, accessed: 13th Feb 2015.

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

[13] Eithan Farchi, Yoel Krasny, and Yarden Nir. Automatic simulation of network problems in UDP-based Java programs. In *Proc. 18th International Parallel and Distributed Processing Symposium.* IEEE, 2004.

[14] S. Feuerhahn, M. Zillgith, C. Wittwer, and C. Wietfeld. Comparison of the communication protocols DLMS/COSEM, SML and IEC 61850 for smart metering applications. In *Proceedings of the Second IEEE International Conference on Smart Grid Communications (SmartGridComm 2011)*, pages 410–415, Brussels, Belgium, 2011. IEEE.

[15] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[16] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[17] Charlie Hosner. OpenVPN and the SSL VPN revolution. SANS Institute, InfoSec Reading Room, 2004. Available at https://openvpn.net/index.php/open-source/articles.html, accessed: 27 Apr 2015.

[18] C. Huitema. Real time control protocol (RTCP) attribute in session description protocol (SDP). IETF RFC 3605, 2003. Available at http://tools.ietf.org/html/rfc3605, accessed: 13th Feb 2015.

[19] Apple Computer, Inc. Apple remote desktop administrators guide version 2.0. Available at http://images.apple.com/remotedesktop/pdf/ARD_Admin_Guide.pdf, accessed: 30 Apr 2015, 2004.

[20] F. Junqueira and B. Reed. *ZooKeeper: Distributed Process Coordination.* O'Reilly, 2013.

[21] Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul Agha. A framework for state-space exploration of Java-based actor programs. In *ASE*, pages 468–479, 2009.

[22] 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*, 40(5):483–501, 2014.

[23] Linux Foundation. Network emulation with netem. `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`. Accessed: 7th Oct 2014.

[24] Donald W. Loveland. *Automated theorem proving: A logical basis (Fundamental studies in computer science).* Elsevier, 1978.

[25] 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, 2013. IEEE.

[26] Douglas Mauro and Kevin Schmidt. *Essential SNMP.* O'Reilly Media, second edition, 2005.

[27] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[28] Jean-Yves Migeon. The MIT Kerberos administrators how-to guide. Available at http://www.kerberos.org/software/adminkerberos.pdf, accessed: 30 Apr 2015, 2008.

[29] P. Mockapetris. Domain names — implementation and specification. IETF RFC 1035, 1987. Available at http://www.ietf.org/rfc/rfc1035, accessed: 13th Feb 2015.

[30] Jayasumana A. Narasiodeyar R. Improvement in packet-reordering with limited re-sequencing buffers: An analysis. In *Local Computer Networks (LCN), 2013 IEEE 38th Conference on*, pages 453–457. IEEE, 2013.

[31] Microsoft Developer Network. Remote desktop protocol. Available at https://msdn.microsoft.com/en-us/library/aa383015.aspx, accessed: 30 Apr 2015.

[32] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[33] P. Reinecke, M. Drager, and K. Wolter. Netemcg — IP packet-loss injection using a continuous-time gilbert model. Technical Report TR-B-11-05, Freie Universitt Berlin, Germany, 2011.

[34] K Robertson, K Miller, M White, and A Tweedly. Starburst multicast file transfer protocol (MFTP) specification. *IETF-DRAFT*, 1998. Available at http://tools.ietf.org/html/draft-miller-mftp-spec-03, accessed: 12th Feb 2015.

[35] Jim Roskind. QUIC: Multiplexed stream transport over UDP. *Google working design document*, 2013.

[36] H. Schulzrinne. RTP: A transport protocol for real-time applications. IETF RFC 3550, 2003. Available at http://tools.ietf.org/html/rfc3550, accessed: 13th Feb 2015.

[37] N. Sebih, F. Weitl, C. Artho, M. Hagiya, M. Yamamoto, and Y. Tanabe. Software model checking of UDP-based distributed applications. In *Proc. Second International Symposium on Computing and Networking (CANDAR'14)*, pages 96–105, Shizuoka, Japan, 2014. IEEE.

[38] Nastaran Shafiei and Franck Van 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.

[39] Nastaran Shafiei and Peter C. Mehlitz. Extending JPF to verify distributed systems. *ACM SIGSOFT Software Engineering Notes*, 39(1):1–5, 2014.

[40] 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, 2010. Springer Berlin Heidelberg.

[41] Scott D. Stoller and Yanhong A. Liu. Transformations for model checking distributed Java programs. In *Proc. 8th Int. SPIN Workshop (SPIN 2001)*, pages 192–199, NY, USA, 2001. Springer-Verlag New York, Inc.

[42] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.

[43] William Rathje, Brad Richards. A framework for model checking UDP network programs with Java Pathfinder. *HILT '14 Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology*, 2014.

[44] Piet Van Mieghem Xiaoming Zhou. Reordering of IP packets in internet. In *Passive and Active Network Measurement, 5th International Workshop, PAM 2004*, pages 237–246, Antibes Juan-les-Pins, France, 2004. Springer.

[45] Min Zhang, Maurizio Dusi, Wolfgang John, and Changjia Chen. Analysis of UDP traffic usage on internet backbone links. In *Applications and the Internet, 2009. SAINT'09. Ninth Annual International Symposium on*, pages 280–281. IEEE, 2009.