Cardinality of UDP Transmission Outcomes

Franz Weitl[1], Nazim Sebih[3], Cyrille Artho[2], Masami Hagiya[3], Yoshinori Tanabe[4], Yoriyuki Yamagata[2], and Mitsuharu Yamamoto[1]

[1] Chiba University, Chiba, Japan
[2] AIST/RISEC, Amagasaki/Tsukuba, Japan
[3] The University of Tokyo, Tokyo, Japan
[4] Tsurumi University, Yokohama, Japan

Abstract. This paper examines the cost of testing network applications using the User Datagram Protocol (UDP). Such applications must deal with packet loss, duplication, and reordering. Ideally, a UDP application should be tested against all possible outcomes of unreliable UDP transmissions. Their number, however, grows at least exponentially in the number of transmitted packets.

To estimate the cost of the exhaustive testing of UDP applications, we determine the number of UDP transmission outcomes analytically. Based on this combinatorial analysis, we derive a sound, complete, and optimal algorithm for generating outcomes of unreliable UDP transmissions. The algorithm is implemented in the net-iocache extension of the software model checker Java Pathfinder (JPF).

Experimental results confirm the consistency of the implementation with the analytical results. In addition, we found that JPF's state matching reduces the explored state space significantly and ensures the practicability of the approach despite of its exponential complexity.

Keywords: User Datagram Protocol, Software Model Checking, Java Pathfinder, Combinatorial Analysis

1 Introduction

Modern software often involves both multi-threading and network communication. Testing such systems is complex due to non-determinism in thread scheduling and network behavior. When applying the User Datagram Protocol (UDP), the application must be tested against non-deterministic outcomes of network input/output (I/O) including packet loss, duplication, and reordering.

Despite of its unreliability, UDP is favorable over the Transmission Control Protocol (TCP) for applications that require low latency and high throughput. These include real-time and multimedia applications such as gaming and media streaming [11, 22], but also high performance computing [13], widely used application-level protocols such as DNS [17], DHCP [5], and the new protocol QUIC [21] for web applications. Studies [32] report on a significant and increasing portion of UDP traffic on the Internet.

Testing a distributed application against all possible outcomes of UDP I/O is challenging because of the explosion of cases when combining packet loss, duplication, and reordering. Network emulators [7,15,19,26] that use stochastic methods for the injection of such *packet perturbation*, avoid a combinatorial explosion but cannot guarantee the coverage of all combinations. Recent work [23,31] proposes the application of software model checking with Java Pathfinder (JPF) [29] for testing UDP applications *exhaustively* against possible outcomes of UDP I/O but it is unclear which problem sizes these exhaustive methods scale up to.

In this paper, we analyze the practical feasibility of model checking UDP applications. We describe unreliable UDP I/O in a formal model and analyze the number of possible outcomes for a sequence of n transmitted packets. The formal model and its analysis yield a sound, complete, and optimal algorithm of generating outcome sets which is a formal and generally applicable version of the algorithms presented in previous work [23]. In experiments, we determine its cost in terms of runtime and memory consumption and compare the number of generated cases with the analytically derived cardinality results. A major finding is that the runtime grows less than the analytical results suggest because JPF recognizes visited states and prunes the exploration of the state space.

These results encourage the application of software model checking for UDP applications despite of its exponential complexity. The availability of efficient formal methods promotes the use of UDP for a broader range of applications, including dependable systems. The contributions of this paper are:

Formal Analysis: We formalize the set of UDP transmission outcomes for n packets and analyze its cardinality. This is an indicator of the computational cost of exhaustively testing UDP applications.

General Algorithm: We derive a new generally applicable algorithm for generating outcome sets which is sound, complete, and optimal. In contrast to previous algorithms [23], it can be implemented independently from JPF.

Evaluation: In experiments, we compare the analytical results with the number of cases generated by the JPF extension net-iocache, and evaluate the impact JPF's state matching on the runtime.

This paper is structured as follows. We introduce relevant concepts of JPF and its extension net-iocache for networked systems in Section 2. Section 3 defines outcome sets of unreliable UDP transmissions and proves their cardinality while Section 4 presents algorithms for generating them. We report on experimental results in Section 5 and discuss related work in Section 6 before concluding the paper in Section 7.

2 Background

Java Pathfinder (JPF) [10,29] is a custom Java Virtual Machine (JVM) written in Java. It runs on top of a host JVM (Fig. 1). The application verified by JPF is called the system under test (SUT). Net-iocache [3, 14] extends JPF towards major parts of the `java.net` application programmig interface (API): It

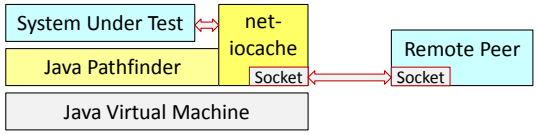

Fig. 1. Java Pathfinder and its extension net-iocache for network communication.

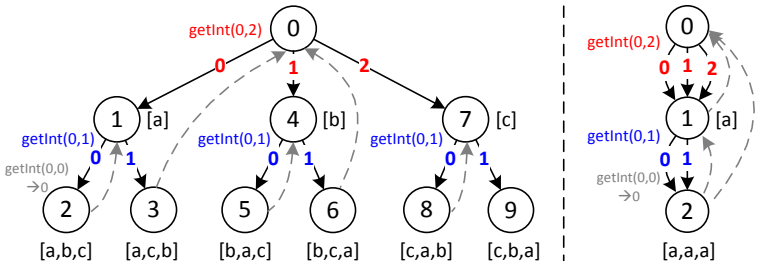

Fig. 2. State space exploration for input sequence `a,b,c` (left) and `a,a,a` (right) with JPF v8.0 rev 25; Dotted arrows: backtracking.

Table 1. Explored state space for different input sequences.

input	cases	branches	states	transitions
`a,b,c`	6	6	10	9
`a,a,a`	6	4	3	5

intercepts method calls of the SUT to classes such as `java.net.DatagramSocket` and forwards network I/O to the remote peers (Fig. 1 center). This way, instances of packet loss, duplication, or reordering can be injected transparently.

For non-deterministic operations such as thread scheduling or random number generation, JPF creates a *choice generator* and explores the rest of the SUT for each of the possible choices on a separate execution branch. JPF offers an application programming interface (API) for creating custom choice generators. Net-iocache uses this API for the exhaustive exploration of non-deterministic outcomes of UDP I/O [23]. E. g., permutations can be generated as follows:

```
1 List<Character> l=new ArrayList<Character>(Arrays.asList('a','b','c'));
  for(int i=0; i<perm.length; i++) {
    int max=l.size()-1;
    perm[i]=l.remove(Verify.getInt(0, max));
5 }
  System.out.print(Arrays.toString(perm)+" ");
```

The program stores the character sequence `'a','b','c'` in a list `l` (line 1) and moves it to an array `perm` (lines 2–5). `Verify.getInt(0,max)` in line 4 creates a data choice generator with choices `0,1,...,max`. When executing on a standard JVM, `Verify.getInt` returns a randomly chosen value in `[0,max]` and the program outputs a single permutation of `a,b,c`, for instance `[c,a,b]`. In contrast, when executing the same program on JPF, it outputs all permutations

`[a,b,c] [a,c,b] [b,a,c] [b,c,a] [c,a,b] [c,b,a]`. This is because JPF executes the SUT for all possible return values of each call to `Verify.getInt` in a depth-first-search manner (Fig. 2 lhs). For the input sequence `a,b,c`, each alternative choice results in a new program state, numbered in the order of their first visit (0: initial state, 9: last visited state). When reaching one of the 6 terminal states $\{2, 3, 5, 6, 8, 9\}$, JPF backtracks the SUT to a previous state with open choices. Note that the result of `Verify.getInt(0,0)` is deterministic. JPF merges it into the same transition as the preceding invocation `Verify.getInt(0,1)` (Fig. 2 lhs bottom).

When the arguments of method `Arrays.asList` in line 1 are changed to "`'a','a','a'`", different return values of `Verify.getInt` lead to the same program state (Fig. 2 rhs). By default, JPF recognizes previously visited states by *state matching* and prunes the search as follows (dotted arrows indicate backtracking): $0 \rightarrow 1 \rightarrow 2 \dashrightarrow 1 \rightarrow 2$ (visited) $\dashrightarrow 0 \rightarrow 1$ (visited) $\dashrightarrow 0 \rightarrow 1$ (visited). Note that only 4 execution branches and 5 forward transitions are executed instead of 6 branches and 9 transitions in the scenario of Fig. 2 lhs. Table 1 summarizes the size and structure of the explored state space for each of the two input sequences `a,b,c` and `a,a,a`. Column 'cases' refers to the number of permutations of length 3, while 'branches' refers to the number of combined choices generated by calls of `Verify.getInt`. If state matching detects visited states, the number of explored 'branches' can be smaller than the number of 'cases'. State matching can be disabled in the JPF settings via property `vm.storage.class` to save memory. If enabled, it leads to a significant speed up in our experiments (Section 5).

3 Formal Analysis of Unreliable UDP Behavior

When a message consisting of a sequence of n packets is sent by UDP, *which messages* possibly arrive at the destination? *How many possibilities* are there, taking arbitrary combinations of packet loss, duplication and reordering into account?

Consider a message being fragmented into the three packets (p_1, p_2, p_3) put onto the network subsequently. Since each packet may get lost, duplicated, and/or reordered, the packet sequences, which possibly arrive at the destination, include

ϵ	empty sequence, all packets lost
(p_1, p_2, p_3)	normal delivery, no loss/duplication/reordering
(p_1, p_3, p_3)	p_2 lost, p_3 duplicated, no reordering
$(p_2, p_3, p_2, p_1, p_3)$	p_2, p_3 duplicated, reordered
...	

How many such messages are there? Let us assume first that packets are duplicated at most once and do not get reordered. Then there are three possibilities for each individual packet: 1) loss, 2) delivery once, 3) delivery twice, resulting in 3^n combinations for n packets, i.e., 27 in the given case of $n = 3$. This means

that even in scenarios without reordering, the number of transmission outcomes grows already exponentially in the number of transmitted packets.

We will show that the number of transmission outcomes increases up to 271 for messages of 3 packets (Table 2), if cases of reordering are considered in addition. Their number depends on the network capacity which is the maximum number of packets the network can hold at a certain time. For instance, the number of transmission outcomes for messages of 3 packets drops to 135 on a network with a capacity of 2 packets and to 27 on a network with capacity 1 which does not permit any reordering (Table 2).

Why is it important to know the number of transmission outcomes precisely? Obviously it is an indicator of the cost of testing a UDP application exhaustively. More importantly, the cardinality analysis reveals the structure of outcome sets and yields an algorithm for generating them which is sound, complete, and optimal by construction.

3.1 Unreliable UDP Transmissions

We denote the set of *natural numbers including 0* as $\mathbb{N}$. $\mathbb{N}_1 =_{\text{def}} \mathbb{N} \setminus \{0\}$ denotes the set of *positive* natural numbers; $[n, m] =_{\text{def}} \{i \in \mathbb{N} : n \leq i \leq m\}$ denotes a *closed interval* in $\mathbb{N}$; $\mathcal{P}(A) =_{\text{def}} \{S : S \subseteq A\}$ denotes the *powerset* of set A; $A \uplus B$ denotes the *union of disjoint sets* A, B, i. e., $A \uplus B = A \cup B$ and $A \cap B = \emptyset$.

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 $(p_1, ..., p_n)$. ϵ denotes the empty sequence for $n = 0$.*

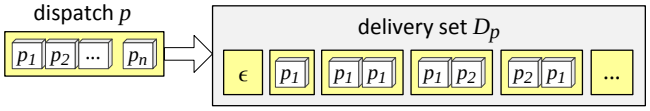

Fig. 3. Set of possible deliveries for a sequence of n dispatched packets $(p_1, ..., p_n)$.

In our model we fix a sequence of n unique packets $p \in P^n$, forwarded to the network (*dispatch*), and define the possible UDP transmission outcomes of p as *delivery set* (Fig. 3):

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

- *p is a* dispatch *iff $p_i = p_j$ implies $i = j$ for all $i, j \in [1, n]$.*
- *$D_p =_{\text{def}} \bigcup_{m \in \mathbb{N}} \{p_i : i \in [1, n]\}^m$ denotes the* set of deliveries *of dispatch p.*
- *The* dispatch order *is: $p_i < p_j \Leftrightarrow_{\text{def}} i < j$ for all $i, j \in [1, n]$.*

Each element of D_p is a sequence of packets of p with arbitrary order and number of instances: $(p_1, p_2, p_1) \in D_{(p_1,p_2)}$ and $\epsilon \in D_{(p_1,p_2)}$ but $(p_1, p_2, p_3) \notin D_{(p_1,p_2)}$ with $p_1, p_2, p_3 \in P$ being distinct packets.

The network has a limited capacity; it can hold at most c packets at a time. After a packet p_i is delivered, at most $c-1$ *late* packets can be delivered that have been sent before p_i. This limits the delivery set for p as follows:

Definition 3 (Capacity-Bounded Deliveries).

Let D_p be the set of deliveries of a dispatch $p \in P^n$. Let $c \in \mathbb{N}_1$ be the maximum number of packets the network can hold at a given time.

For a delivery $q \in D_p$ with length m and $i \in [1, m-1]$, let $L_{q,i} =_{\text{def}} \{q_j : j > i \wedge q_j < q_i\}$ denote the set of packets which are late *in q w. r. t. q_i. Then*

$$D_{p,c} =_{\text{def}} \{(q_1, ..., q_m) \in D_p : \forall i \in [1, m-1].\ |L_{(q_1,...,q_m),i}| < c\}$$

is the set of capacity-c-bounded deliveries *of p.*

As an example, consider the delivery $q = (p_2, p_3, p_1, p_3, p_2)$ of dispatch $p = (p_1, p_2, p_3)$. Then $L_{q,1} = \{p_1\}$, $L_{q,2} = \{p_1, p_2\}$, $L_{q,3} = \emptyset$, and $L_{q,4} = \{p_2\}$. Thus $q \in D_{p,c}$ if and only if $c > |L_{q,2}| = 2$. Note that $D_{p,c} = D_p$ if $c \geq n$. Furthermore, a network with capacity 1 does not permit reordering. For instance, $(p_1, p_1, p_2) \in D_{(p_1,p_2),1}$ but $(p_1, p_2, p_1) \notin D_{(p_1,p_2),1}$.

In the example above one may argue that a network capacity of 3 is still not sufficient for delivering $(p_2, p_3, p_1, p_3, p_2)$ because after the first delivery of p_3 there are three more packets delivered which must have been on the network at the time p_3 is delivered. Definition 3 is based on the assumption that a packet is not necessarily duplicated at dispatch time but at any time while it is on the network. The latest possible time is just the time of delivery. This most general assumption regarding duplication maximizes the cases of reordering permitted by a given network capacity in our model. For instance, a network with capacity 3 can generate the delivery $(p_2, p_3, p_1, p_3, p_2)$ as follows:

Event	**Packets on the network**	**Delivered packets**
p_1, p_2, p_3 dispatched	$\{p_1, p_2, p_3\}$	$()$
duplicate of p_2 delivered	$\{p_1, p_2, p_3\}$	(p_2)
duplicate of p_3 delivered	$\{p_1, p_2, p_3\}$	(p_2, p_3)
p_1 delivered	$\{p_2, p_3\}$	(p_2, p_3, p_1)
p_3 delivered	$\{p_2\}$	(p_2, p_3, p_1, p_3)
p_2 delivered	$\emptyset$	$(p_2, p_3, p_1, p_3, p_2)$

Delivery sets of non-empty dispatches are infinite because deliveries may contain arbitrarily many instances of dispatched packets. We identify finite subsets by constraining the number of times each dispatched packet may appear in a delivery, using a set of *multiplicity choices*:

Definition 4 (Multiplicity-Bounded Deliveries).

Let $M \subset \mathbb{N}$ be a non-empty, finite set of natural numbers, called multiplicity choices. *Let D_p be the delivery set of dispatch $p \in P^n$. Then*

$$D_{p,M} =_{\text{def}} \{(q_1, ..., q_m) \in D_p : \forall i \in [1, n].\ |\{j \in [1, m] : q_j = p_i\}| \in M\}$$

is the set of multiplicity-M-bounded deliveries *of* p.

For instance, $\{1, 2\}$ is the set of multiplicity choices that permits each packet to be delivered once or twice. Hence $(p_1, p_2, p_1) \in D_{(p_1,p_2),\{1,2\}}$ but $(p_1, p_2, p_1, p_1) \notin D_{(p_1,p_2),\{1,2\}}$ and $(p_1, p_1) \notin D_{(p_1,p_2),\{1,2\}}$. In general, we consider the deliveries that are both multiplicity- and capacity-bounded:

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

$$D_{p,M,c} =_{\text{def}} D_{p,M} \cap D_{p,c}$$

is the set of multiplicity-M-and-capacity-c-bounded deliveries *of* p.

3.2 Cardinality of Unreliable UDP Transmissions

We analyze the cardinality of the delivery set $D_{p,M,c}$ by splitting it into partitions whose cardinality can be determined easier. This partitioning also provides the formal ground for a sound, complete, and optimal algorithm for generating delivery sets (Section 4).

Delivery sets are partitioned along the two independent dimensions of variation: 1) the number of delivered instances of each dispatched packet, called *multiplicity vector* and 2) reordering as permitted by the network's capacity.

For instance, $(p_2, p_4, p_2, p_1, p_4)$ is a delivery of (p_1, p_2, p_3, p_4) with multiplicity vector $(1, 2, 0, 2)$, meaning that p_1 is delivered exactly once, p_2 and p_4 are delivered exactly twice, and p_3 is not delivered. Other instances with this multiplicity vector are obtained by reordering, e.g., $(p_1, p_2, p_2, p_4, p_4)$, $(p_1, p_2, p_4, p_2, p_4), ...$ For determining the number of such permutations with repetition, we can apply known results of combinatorics.

Formally, we divide $D_{p,M,c}$ into partitions using *multiplicity vectors* as follows:

Definition 6 (Multiplicity-Vector-Bounded Delivery Sets).
Let $p \in P^n$ be a dispatch. Then $\mu \in \mathbb{N}^n$ is a multiplicity vector *for p and*

$$D_{p,\mu} =_{\text{def}} \{(q_1, ..., q_m) \in D_p : \forall i \in [1, n].\ |\{j \in [1, m] : q_j = p_i\}| = \mu_i\}$$

is the set of multiplicity-vector-μ-bounded deliveries *of* p.

For $c \in \mathbb{N}_1$, $D_{p,\mu,c} =_{\text{def}} D_{p,\mu} \cap D_{p,c}$ is the set of multiplicity-vector-μ-and-capacity-c-bounded deliveries *of* p.

A multiplicity vector μ defines for each individual packet p_i of a dispatch $p \in P^n$, how often it appears in a delivery of $D_{p,\mu}$. For instance, the multiplicity vector $(2, 1)$ permits such deliveries of dispatch (p_1, p_2) where p_1 appears twice and p_2 once. Thus $(p_1, p_2, p_1) \in D_{(p_1,p_2),(2,1)}$ but $(p_2, p_1, p_2) \notin D_{(p_1,p_2),(2,1)}$.

Multiplicity vectors partition the set of multiplicity-M-and-capacity-c-bounded deliveries $D_{p,M,c}$ into pairwise disjoint sets. By Definitions 4 and 6, it holds for $\mu \in M^n$ and $\mu' \in M^n \setminus \{\mu\} : D_{p,\mu,c} \cap D_{p,\mu',c} = \emptyset$ and $\bigcup_{\mu \in M^n} D_{p,\mu,c} = D_{p,M,c}$. This gives the following Lemma:

Lemma 1 (Partitioning of Delivery Set).
For a dispatch $p \in P^n$, a non-empty, finite set of multiplicity choices $M \subset \mathbb{N}$, and a network capacity $c \in \mathbb{N}_1$ it holds:

$$D_{p,M,c} = \biguplus_{\mu \in M^n} D_{p,\mu,c} \tag{1}$$

$$|D_{p,M,c}| = \sum_{\mu \in M^n} |D_{p,\mu,c}| \tag{2}$$

Next we derive the cardinality of $D_{p,\mu,c}$, using the following operations:

Definition 7 (Vector Operations).
Let $p \in P^n$ be a dispatch, $c \in \mathbb{N}_1$ a network capacity, and $\mu \in \mathbb{N}^n$ a multiplicity vector. Then

- $|\mu| =_{\text{def}} |\{i \in [1,n] : \mu_i \neq 0\}|$ *denotes the number of packets that appear at least once in any delivery $q \in D_{p,\mu,c}$.*
- $\mathbf{u}_i$ *denotes the i-th* unit vector *in $\mathbb{N}^n$ for $i \in [1,n]$. I. e., with $x = \mathbf{u}_i$ it holds: $x_i = 1$ and $x_j = 0$ for all $j \in [1,n] \setminus \{i\}$.*
- $\mu - \mathbf{u}_i$ *denotes the vector subtraction of $\mathbf{u}_i$ from μ. I. e., with $x = \mu - \mathbf{u}_i$ it holds: $x_i = \mu_i - 1$ and $x_j = \mu_j$ for all $j \in [1,n] \setminus \{i\}$.*
- $F_{\mu,c} =_{\text{def}} \{i \in [1,n] : \mu_i > 0 \wedge |\{j \in [1,i] : \mu_j > 0\}| \leq c\}$ *denotes the* first *c indices where μ has a value greater than zero. These are the indices of the first c packets of a dispatch p which appear at least once in any delivery $q \in D_{p,\mu,c}$.*

Lemma 2 (Partitioning of Multiplicity-Vector-Bounded Deliveries). *For a dispatch $p \in P^n$, capacity $c \in \mathbb{N}_1$, and multiplicity vector $\mu \in \mathbb{N}^n$ it holds:*

$$D_{p,\mu,c} = \begin{cases} \{\epsilon\} & \text{if } |\mu| = 0 \\ \biguplus_{i \in F_{\mu,c}} \{p_i\} \times D_{p,\mu-\mathbf{u}_i,c} & \text{if } |\mu| > 0 \end{cases} \tag{3}$$

Proof (Sketch).
$D_{p,\mu,c} = \{\epsilon\}$ for $|\mu| = 0$ follows directly from Definitions 2 and 6.

Assume $|\mu| > 0$. On a network with capacity c, the first packet q_1 of a delivery $q \in D_{p,\mu,c}$ of length $m \in \mathbb{N}$ is one of the first c packets of dispatch p which appear at least once in q, i. e., $q_1 = p_i$ for some $i \in F_{\mu,c}$. Packet p_i appears $\mu_i - 1$ times in the remaining delivery sequence $(q_2, ..., q_m)$ Thus the multiplicity vector of $(q_2, ..., q_m)$ is $\mu - \mathbf{u}_i$ and we get Equation (3) for $|\mu| > 0$. □

Proposition 1 (Cardinality of Multiplicity-Vector-Bounded Deliveries).
For a dispatch $p \in P^n$, capacity $c \in \mathbb{N}_1$, and multiplicity vector $\mu \in \mathbb{N}^n$ with $|\mu| > 0$ it holds:

$$|D_{p,\mu,c}| = \sum_{i \in F_{\mu,c}} |D_{p,\mu-\mathbf{u}_i,c}| \tag{4}$$

For $|\mu| \leq c$ it holds:

$$|D_{p,\mu,c}| = |D_{p,\mu}| = \frac{(\sum_{i=1}^n \mu_i)!}{\prod_{i=1}^n \mu_i!} \tag{5}$$

Proof.

Equation (4) is direct consequence of Lemma 2.

Equation (5) is shown as follows. For $|\mu| \leq c$ we get: $D_{p,\mu,c} = D_{p,\mu}$ since reordering is not limited by c if less than c packets are delivered.

$D_{p,\mu}$ is the set of permutations of n packets where each packet p_i with $i \in [1, n]$ appears μ_i times (multiset permutation [4]). Its cardinality is given by the multinomial coefficient $\binom{m}{\mu_1,\ldots,\mu_n}$ with $m = \sum_{i=1}^{n} \mu_i$ [4, 9]. We get:

$$|D_{p,\mu,c}| = |D_{p,\mu}| = \binom{\sum_{i=1}^{n} \mu_i}{\mu_1, \ldots, \mu_n} = \frac{(\sum_{i=1}^{n} \mu_i)!}{\prod_{i=1}^{n} \mu_i!} \quad \square$$

Lemma 1 and Proposition 1 enable the calculation of $|D_{p,M,c}|$ with $p \in P^n$, by unfolding the recursive Equation (4) until $|\mu| \leq c$ and then applying Equation 5. Table 2 displays the numbers for $M = \{0, 1, 2\}$ (packet loss/normal delivery/duplication), $n \in [1, 5]$, and $c \in [1, 6]$.

Table 2. Cardinality of delivery sets $D_{p,\{0,1,2\},c}$ with $p \in P^n, n \in [1, 5]$ and $c \in [1, 6]$; Numbers in blue are referred in the beginning of Section 3 and in Section 5.

$n \setminus c$	1	2	3	4	5	6
1	3	3	3	3	3	3
2	9	19	19	19	19	19
3	27	135	271	271	271	271
4	81	955	3825	7365	7365	7365
5	243	6711	51331	176011	326011	326011

4 Generating UDP Transmission Outcomes

According to Lemmata 1 and 2, the delivery set $D_{p,M,c}$ for a given dispatch $p \in P^n$, a non-empty, finite set of multiplicity choices $M \subset \mathbb{N}$, and a network capacity $c \in \mathbb{N}_1$ is partitioned as

$$D_{p,M,c} = \biguplus_{\mu \in M^n} D_{p,\mu,c}$$

$$D_{p,\mu,c} = \begin{cases} \{\epsilon\} & \text{if } |\mu| = 0 \\ \biguplus_{i \in F_{\mu,c}} \{p_i\} \times D_{p,\mu-\mathbf{u}_i,c} & \text{if } |\mu| > 0 \end{cases}$$

Algorithm 1 is a direct operational reformulation of these equations. This ensures its soundness, completeness, and optimality in the sense that each element in $D_{p,M,c}$ is calculated exactly once.

Function delivery of Algorithm 2 returns an arbitrary element of the delivery set $D_{p,M,c}$. Function **chooseOneOf**, similar to JPF's `Verify.getInt` (see Section 2), performs a non-deterministic choice, returning an arbitrary element

of a non-empty set. The combination of all non-deterministic choices in Algorithm 2 yields the delivery set $D_{p,M,c}$. Parameters M and c of function delivery are configuration settings chosen by the user according to the test goals for a given SUT [23, 24].

```
Function deliveries(p, M, c)
    n ← arity(p);
    D ← ∅;
    for μ ∈ M^n do
        D ← D ⊎ delivsRec(p, μ, c);
    return D;

Function delivsRec(p, μ, c)
    if |μ| = 0 then
        return {ϵ};
    F ← getFirst(μ, c);
    D ← ∅;
    for i ∈ F do
        μ' ← μ;
        μ'_i ← μ'_i − 1;
        D' ← delivsRec(p, μ', c);
        D ← D ⊎ ({p_i} × D');
    return D;

Function getFirst(μ, c)
    F ← ∅;
    i ← 1;
    while i ≤ arity(μ) ∧ |F| < c do
        if μ_i > 0 then
            |F| ← |F| ⊎ {i};
        i ← i + 1;
    return F;
```

Algorithm 1: Delivery set generation.

```
Function delivery(p, M, c)
    n ← arity(p);

    for i ∈ [1, n] do
        μ_i ← chooseOneOf(M);
    return delivRec(p, μ, c);

Function delivRec(p, μ, c)
    if |μ| = 0 then
        return ϵ;
    F ← getFirst(μ, c);

    i ← chooseOneOf(F);
    μ' ← μ;
    μ'_i ← μ'_i − 1;
    (q_1, ..., q_m) ← delivRec(p, μ', c);

    return (p_i, q_1, ..., q_m);
```

Algorithm 2: Non-deterministic generation of a single delivery.

5 Experimental Results

We implemented an adapted version of Algorithm 2 in net-iocache [23]: It generates packet perturbation for *individually* sent and received packets rather than for packet *sequences*.

In a scenario inspired by the UDP-based file transfer protocols TFTP [27] and MFTP [20], we determine the number of cases generated by net-iocache and compare them with the analytical results on the cardinality of delivery sets (Proposition 1). In addition, we evaluate the impact of JPF's state matching (Section 2) on the performance. The source repository of net-iocache v2 [30] comprises this and other experiments.

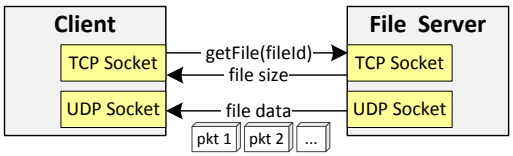

Fig. 4. Components of the file transfer application.

Fig. 4 shows the setting: A *client* connects to a TCP port of a *file server* for exchanging control information and listens on a UDP port for receiving files. The server adds a sequence number to each UDP packet, allowing the client to detect missing or duplicated packets, and to restore their original order. The server does not read files from the disk but synthesizes them on demand in such a way that each packet of each file is distinct. This maximizes the number of program states of the client (cf. scenario in Fig. 2). The client checks the validity of the received file content but does not store it to the file system. This avoids effects of file I/O on the runtime behavior. Dropped packets are not retransmitted to keep the number of packets sent by the server independent from the generated packet perturbation.

We analyze the runtime behavior of JPF when checking the client, receiving files with increasing number of packets (packet size: 512 bytes), for UDP transmissions with possible packet loss, duplication and reordering, according to multiplicity choices $M = \{0, 1, 2\}$ and capacity $c = 2$.

JPF does not detect any errors and thus explores the entire state space of the SUT. Both the client and file server were executed on the same 8 core Mac Pro workstation with 24 GB of memory running Ubuntu 14.04.2 LTS (64 Bit), Java RTE 1.8.0_45-b14, JPF v8.0 (rev 25), and net-iocache v2 (rev 76) [30].

Table 3. File transfer client explored by JPF for **one** file with $n \in [1, 12]$ packets and delivery set $D_{p,\{0,1,2\},2}$, permitting packet loss, duplication, and reordering.

		no state matching			state matching			speed-up
packets	cases	branches	time[s]	mem[MB]	branches	time[s]	mem[MB]	factor
1	3	3	0.3	362	3	0.3	362	1.00
2	19	19	0.5	362	19	0.4	362	1.25
4	955	955	3.4	457	303	0.8	362	4.25
8	2,305,819	2,305,819	5437.3	1,782	17,383	11.4	1,021	476.96
16	$13.9 \cdot 10^{12}$	–	–	–	$12.2 \cdot 10^{6}$	6693.7	1,782	–

Table 3 shows the runtime results when transferring one file with an increasing number of packets. Column 'cases' refers to the cardinality of the delivery set $D_{p,\{0,1,2\},2}$ (second data column of Table 2) while 'branches' refers to number of combined choices actually explored by JPF (cf. Section 2, especially Table 1). When state matching is disabled, the number of cases is identical with

the number of branches JPF explores. This confirms the consistency of the implementation with the analytical results for the cardinality of delivery sets in Section 3.2. Enabled state matching, however, reduces the number of explored execution branches significantly and enables the exhaustive exploration of much larger problems than the analytical results suggest.

Equivalent states detected by JPF's state matching arise from the reaction of the SUT on packet duplication and reordering. Duplicated packets are discarded immediately [23] and do not lead to a new program state. Similarly, the compensation of packet reordering eventually leads to the same program state for all generated packet permutations. A similar speed up by state matching can be expected for applications such as multimedia streaming that cope with duplicated and reordered packets in this way.

6 Related Work

In previous work [23, 24], we created a new version of net-iocache [30] for the exhaustive exploration of UDP transmission outcomes with JPF and conducted first experiments to confirm the feasibility and usefulness of the approach. This paper describes UDP transmissions and their enumeration formally and analyzes its cardinality. The proposed algorithm for generating the set of possible outcomes of unreliable UDP transmissions extends existing algorithms for enumerating permutations [12] towards a limited reorder window according to the assumed network capacity. The non-deterministic version of the algorithm can be considered as a variant of the Fisher-Yates shuffle algorithm [6, 8] for generating random permutations, extended in two aspects: 1) Instead of choosing each element *exactly once*, each element (packet) is chosen a *number of times* according to the chosen number of duplications; 2) Instead of choosing an *arbitrary* element from the set of not yet chosen elements, only one of the remaining *first c* elements (packets) is chosen in each iteration to account for the network capacity c.

Rathje and Richards [31] use JPF for exploring non-deterministic outcomes of UDP I/O. They apply a *centralization*- and stub-based approach: All communicating peers are transformed into a single multi-threaded program and network I/O is replaced by inter-thread communication using message queues. Packet loss and reordering is generated but packet duplication is not covered. The adopted approach is not entirely automatic: A small implementation effort is required for each individual SUT. Stoller and Liu [28] coined the term *centralization* for merging multiple processes into one. In their work, Java RMI method invocations are replaced by local method calls. This has been extended to TCP sockets [2, 16]. A similar approach analyzes the complete state space of all processes by extending JPF itself [25] rather than pre-processing the SUT.

In contrast to centralization, net-iocache adopts a modular approach [3, 14]: A single peer is selected as SUT and explored by JPF while the other peers run as remote processes outside of JPF. Net-iocache stores and replays network I/O in a cache to synchronize the backtracked SUT with the remote peers. The

modular approach leads to smaller number of concurrent threads in the SUT, reducing the state space and increasing scalability. In general, however, only a part of the state space of the distributed system is covered. For an in-depth discussion of the differences between centralization and net-iocache, we refer the reader to previous work [14].

Instead of software model checking, stochastic methods have been applied for the testing of UDP applications: Farchi et al. [7] 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 [15] and its extensions [19,26] are Linux modules that inject stochastically packet delays, loss, duplication, reordering, and IP packet corruption to simulate non-deterministic unreliable UDP I/O. Stochastic methods are more scalable but cannot guarantee complete coverage.

The reordering of network packets has been described formally and the impact of re-sequencing on the *performance* of streaming applications has been evaluated [18]. Two metrics are considered: *reordering density*, defining the distribution of the 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. To the best of our knowledge, the number of outcomes of unreliable UDP I/O has not been addressed in previous work.

Work on verifying programs with unreliable channels [1] shows that the *reachability problem* as well as *safety* and *eventuality* properties become decidable for communicating *infinite* state systems when lossy instead of lossless channels are used. In our work, we address the verification of *finite* state systems by exhaustively enumerating the outcomes of non-deterministic UDP I/O. The implementation of the proposed algorithm in the software model checker JPF enables the direct checking of Java programs without modeling effort, but it cannot be applied to models of infinite state systems.

7 Conclusion

Based on a formal model of UDP's unreliable transmission behavior, we analyzed the number of transmission outcomes and derived a sound, complete, and optimal algorithm for generating them. The algorithm is implemented in the JPF extension net-iocache. In experiments, the behavior of net-iocache is consistent with the analytical results: It generates the same number of cases as predicted by the formal analysis. We observed in addition, that JPF's state matching reduces the state space significantly which enables the exhaustive exploration of scenarios with trillions of cases.

Future work addresses the following issues: 1) By mapping multiplicity-and-capacity-bounded delivery sets onto known problems in combinatorics, it may be possible to derive a *non*-recursive precise formula and/or tight approximations of their cardinality. 2) Additional experiments would help to evaluate the effectiveness and scalability of the approach for a broader range of applications.

3) Since techniques such as state matching cannot solve the inherent combinational complexity of exhaustive techniques, the combination of software model checking with other, more scalable methods such as runtime verification, is an important issue of our future work.

Acknowledgements

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

References

1. Abdulla, P.A., Jonsson, B.: Verifying programs with unreliable channels. Information and Computation 127(2), 91–101 (1996)
2. Artho, C., Garoche, P.: Accurate centralization for applying model checking on networked applications. In: Proc. 21st Int. Conf. on Automated Software Engineering (ASE 2006). pp. 177–188. Tokyo, Japan (2006)
3. Artho, C., Leungwattanakit, W., Hagiya, M., Tanabe, Y.: Efficient model checking of networked applications. In: Proc. 46th Int. Conf. on Objects, Models, Components, Patterns (TOOLS EUROPE 2008). LNBIP, vol. 11, pp. 22–40. Springer (2008)
4. Bona, M.: Combinatorics of Permutations. CRC Press, second edition edn. (2012)
5. Droms, R.: Dynamic host configuration protocol. IETF RFC 2131 (1997), available at http://www.ietf.org/rfc/rfc2131, accessed: 13th Feb 2015
6. Durstenfeld, R.: Algorithm 235: Random permutation. Communications of the ACM 7(7), 420– (1964)
7. Farchi, E., Krasny, Y., Nir, Y.: Automatic simulation of network problems in UDP-based Java programs. In: Proc. 18th International Parallel and Distributed Processing Symposium. IEEE (2004)
8. Fisher, R.A., Yates, F.: Statistical tables for biological, agricultural and medical research. Oliver & Boyd, London, 3rd edn. (1948), pp. 26–27
9. Hall, M.: Combinatorial theory. Wiley (1986)
10. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. International Journal on Software Tools for Technology Transfer 2(4), 366–381 (2000)
11. Huitema, C.: 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
12. Ives, F.M.: Permutation enumeration: four new permutation algorithms. Communications of the ACM 19(2), 68–72 (1976)
13. Junqueira, F., Reed, B.: ZooKeeper: Distributed Process Coordination. O'Reilly (2013)
14. Leungwattanakit, W., Artho, C., Hagiya, M., Tanabe, Y., Yamamoto, M., Takahashi, K.: Modular software model checking for distributed systems. IEEE Transactions on Software Engineering 40(5), 483–501 (2014)
15. Linux Foundation: Network emulation with netem. http://www.linuxfoundation.org/collaborate/workgroups/networking /netem, accessed: 7th Oct 2014

16. Ma, L., Artho, C., Sato, H.: Analyzing distributed Java applications by automatic centralization. In: Proc. 2nd IEEE Workshop on Tools in Process. IEEE, Kyoto, Japan (2013)
17. Mockapetris, P.: Domain names — implementation and specification. IETF RFC 1035 (1987), available at http://www.ietf.org/rfc/rfc1035, accessed: 13th Feb 2015
18. Narasiodeyar R., J.A.: Improvement in packet-reordering with limited re-sequencing buffers: An analysis. In: Local Computer Networks (LCN), 2013 IEEE 38th Conference on. pp. 453–457. IEEE (2013)
19. Reinecke, P., Drager, M., Wolter, K.: Netemcg — IP packet-loss injection using a continuous-time Gilbert model. Tech. Rep. TR-B-11-05, Freie Universität Berlin, Germany (2011)
20. Robertson, K., Miller, K., White, M., Tweedly, A.: 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
21. Roskind, J.: QUIC: Multiplexed stream transport over UDP. Google working design document (2013)
22. Schulzrinne, H.: RTP: A transport protocol for real-time applications. IETF RFC 3550 (2003), available at http://tools.ietf.org/html/rfc3550, accessed: 13th Feb 2015
23. Sebih, N., Weitl, F., Artho, C., Hagiya, M., Yamamoto, M., Tanabe, Y.: Software model checking of UDP-based distributed applications. In: Proc. Second International Symposium on Computing and Networking (CANDAR'14). pp. 96–105. IEEE, Shizuoka, Japan (2014)
24. Sebih, N., Weitl, F., Artho, C., Hagiya, M., Yamamoto, M., Tanabe, Y.: Software model checking of UDP-based distributed applications. International Journal of Networking and Computing (IJNC) 5(2), 373–402 (2015)
25. Shafiei, N., Mehlitz, P.C.: Extending JPF to verify distributed systems. ACM SIGSOFT Software Engineering Notes 39(1), 1–5 (2014)
26. Sliwinski, J., Beben, A., Krawiec, P.: EmPath: Tool to emulate packet transfer characteristics in IP network. In: Proceedings of the Second International Workshop on Traffic Monitoring and Analysis (TMA 2010). pp. 46–58. Springer (2010)
27. Sollins, K.: The TFTP protocol (revision 2). IETF RFC 1350 (1992), available at http://tools.ietf.org/html/rfc1350, accessed: 1th May 2015
28. Stoller, S.D., Liu, Y.A.: Transformations for model checking distributed Java programs. In: Proc. 8th Int. SPIN Workshop (SPIN 2001). pp. 192–199. Springer-Verlag New York, Inc., NY, USA (2001)
29. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering Journal 10(2), 203–232 (2003)
30. Weitl, F., Sebih, N., Artho, C.: jpf-net-iocache v2 — source code repository. https://bitbucket.org/weitl/jpf-net-iocache, accessed: 15th Apr 2015
31. 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)
32. Zhang, M., Dusi, M., John, W., Chen, C.: Analysis of UDP traffic usage on Internet backbone links. In: Proceedings of the 9th Annual International Symposium on Applications and the Internet (SAINT 2009). pp. 280–281. IEEE (2009)