Model Checking Networked Programs in the Presence of Transmission Failures

Cyrille Artho
National Institute of Informatics, Tokyo, Japan

Christian Sommer
ETH Zürich, Zurich, Switzerland

Shinichi Honiden
National Institute of Informatics, Tokyo, Japan

Abstract

Software model checkers work directly on single-process programs, but not on multiple processes. Conversion of processes into threads, combined with a network model, allows for model checking distributed applications, but does not cover potential communication failures. This paper contributes a fault model for model checking networked programs. If a naïve fault model is used, spurious deadlocks may appear, because certain processes are terminated before they can complete a necessary action. Such spurious deadlocks have to be suppressed, as implemented in our model checker extension. Our approach found several faults in existing applications, and scales well because exceptions generated by our tool can be checked individually.

1. Introduction

Model checking [7] explores the entire behavior of a system by investigating each reachable system state. Recently, model checking has been applied directly to software. Unlike in classical model checking, the actual implementation, and not just the design, is model checked. Java [14, 20] is a popular object-oriented, multi-threaded programming language. Verification of Java programs has become increasingly important. Several model checkers for Java-based programs have been created [4, 9, 34]. Existing software model checkers can explore only a single process and are not applicable to networked applications, where several processes interact. Many non-trivial programs that are in use today use network communication.

Process centralization is a technique that allows model checking of distributed applications: Processes [32] are converted into *threads* [14, 32] and merged into a single process [29]. Networked applications can then run inside one multi-threaded process. Previous work has addressed modeling of TCP/IP communication [3] but has not covered potential network failures. Unlike other approaches, centralization uses a faithful transformation of all processes involved and does not use an abstraction of other processes.

The Java programming language uses exceptions to signal failure of a library or system call [14]. The ideas in this paper are thus applicable to any other programming language supporting exceptions, such as C++ [30], Eiffel [21], or C# [22]. When an exception is thrown, the current stack frame is cleared and its content replaced with a single instance of type `Exception`. In Java, failures of network operations result in exceptions of type `IOException` or `ConnectException` [6, 12].

Our program model consists of applications using TCP/IP networking. Target language is Java, because possible failures are easily identifiable through exception signatures. We model library calls to network operations as open operations that can generate an exception non-deterministically. This is similar to other approaches [5, 8, 11, 19]. However, we combine this approach with process centralization, which allows to model check several applications interacting with each other [3]. Due to the large state space, exhaustive treatment of exceptions is not always possible, and may force the choice of a slight under-approximation of the program behavior. Based on insights gained from this work, our contributions are as follows:

1. We show that modeling potential network failures independently is an effective, scalable way of enhancing the model checking of distributed applications.
2. We show how a failure in one process may trigger an exception in another process.
3. We introduce an automated tool that instruments potential failures. Model checking the resulting program found several faults that were not detected by other means.
4. We show the problem of inter-process deadlocks arising due to processes being prematurely terminated by

an exception. Some, but not all, of these deadlocks are spurious; we show a solution to this problem.

This paper is organized as follows: Section 2 gives the necessary background about network failures and the exception mechanism in Java. Section 3 describes how fault injection is implemented. Initial experiments revealed the need for special treatment of inter-process deadlocks; this run-time component is described in Section 4. The results of our experiments are given in Section 5. Sections 6 and 7 describe related and future work and Section 8 concludes.

2. Background

An *exception*, as commonly used in many programming languages [14, 21, 22, 30], indicates an extraordinary condition in the program, such as the unavailability of a resource. In Java, a method call that fails may "throw" an exception by constructing a new instance of `java.lang.Exception` or a subtype thereof, and using a `throw` statement to "return" this exception to the caller [14]. At the call site, the exception will override normal control flow. The caller may install an exception *handler* by using the `try`/`catch` statement. A `try` block includes a sequence of operations that may fail. Upon failure, remaining instructions of the `try` block are skipped, the current method stack frame is replaced by a stack frame containing only the new exception, and control is transferred to the exception handler, indicated in Java by the corresponding `catch` block.

The usage and semantics of exceptions covers a wide range of behaviors. In Java, exceptions are used to signal the unavailability of a resource (e.g., when a file is not found or cannot be written), failure of a communication (e.g., when a socket connection is closed), when data does not have the expected format, or for programming errors such as accessing an array at an illegal index. Two fundamentally different types of exceptions can be distinguished: *Unchecked* exceptions and *checked* exceptions. Unchecked exceptions are of type `RuntimeException` and do not have to be declared in a method. They typically concern programming errors, such as array bounds overflows, and can be tested through conventional means, by white box testing.[1] On the other hand, checked exceptions have to be declared by a method that may throw them. Failure of external operations results in such checked exceptions [6, 12]. Such external failures cannot be tested easily, as the operations leading to a failure occur in external library operations, which cannot be controlled by the application. For instance, a network connection outage is hard to simulate. This work focuses on such external failures.

[1] This involves writing a test case where the code in question is called with incorrect parameters.

Code instrumentation injects additional code into an application, adding extra behavior to it, while not changing the original behavior or only changing it in a very limited way. It corresponds to a generic form of aspect-oriented programming [17], which organizes code instrumentation into a finite set of operations. *Program steering* [18] overrides normal execution flow, typically altering program behavior using application-specific properties [18], or as schedule perturbation [28], which covers non-determinism in thread schedules. *Fault injection* [2, 16] refers to influencing program behavior by simulation of failures in hardware or software. In model checking, such faults can be injected non-deterministically, covering both the successful and the failed case [5, 8]. In our case, we used the Java PathFinder (JPF) model checker [34] to analyze our programs, as JPF currently is the only openly available Java model checker supporting user-defined non-deterministic outcomes.

Software model checkers execute a software application directly, with little or no abstraction. Such model checkers use a concrete initial state, provided by a test case. Therefore, model checking can be seen as program testing with an exhaustive exploration of non-deterministic choices. Such choices include all possible interleavings of threads. Multiple threads share the address space of a process [32], requiring proper synchronization to avoid access conflicts. A distributed application consists of several processes communicating through means such as TCP/IP networking. Because writing multi-threaded programs is difficult, software model checkers have received much attention in the last years as a means of finding defects in concurrent software. Unlike classical testing, model checkers do not have to rely on the environment to generate a particular schedule that reveals a fault in the program.

Unfortunately, almost all software model checkers can only model check a single process [4, 9, 34]. Alternatives provide an interesting direction for future research but do not yet scale to anything but very small applications [24]. Therefore, model checking networked applications provides two fundamental problems: The fact that multiple processes have to be model checked, and the problem of treating network communication. The first problem can be solved by *centralization*, which converts processes into threads and transforms other aspects of the program such that the original semantics is preserved [3, 29]. These transformations ensure that each process can run independently and maintain a different address space for each process. Therefore, centralization embodies a reversible, isomorphic transformation: Each state in the resulting single-process program corresponds to a state in the original multi-process program. Network communication has been addressed by a recent model [3]. This model uses inter-thread signals (`wait`/`notify` in Java) to simulate blocking system calls, and inter-thread pipes to model the communication channel itself.

The above-mentioned work [3] does not cover transmission failures, which are modeled in this paper. Our model checking process works as follows: (1) The source code is compiled, (2) the centralization tool merges all processes, (3) faults are injected into the application, (4) the resulting application is model checked using a test case as the initial state of a model checking run. This contribution of this paper covers steps 3 and 4, as described in Sections 3 and 4.

3. Fault injection

Prior to model checking a centralized application, faults that simulate possible transmission failures are injected into the code. The first challenge in fault injection is the identification of the methods where exceptions should be generated. We implemented a tool that analyzes given programs for calls to these methods. Before each such call, code that uses `Verify.randomBool` from JPF is inserted . This simulates both the successful case, where no special action is taken and the following library call succeeds, and the failure case. In the failure case, a new instance of type `IOException` is constructed and thrown. If a corresponding exception handler is present, it is then triggered by that exception; otherwise, the exception propagates up the call stack. Therefore, code instrumentation corresponds to insertion of the following code sequence before each method call in question:

```
if (Verify.randomBool()) {
  throw new IOException();
}
```

This example does not include a message string. Message strings represent a human-readable explanation, and are used for displaying diagnostic messages [2]. Our implementation includes a fixed message, distinguishing injected exceptions from conventional ones for human analysis.

Almost any method may throw exceptions. Certain exceptions, such as exceptions referring to character encoding support, can be tested through conventional white-box testing and should not be instrumented.[2] Furthermore, we did not model failures of type `SocketException`, which concern low-level TCP methods that were not used by our example applications. Table 1 shows all methods of the relevant classes in packages `java.net` and `java.io` that can throw an `IOException`. Such exceptions are used to signal the occurrence of an I/O error or similar problems [31]. Certain methods, such as constructors, are declared to throw possible exceptions, but not all constructor variants actually implement them. Such exceptions were excluded in our model. Finally, failures that do not involve non-determinism can be tested conventionally (through white-box testing) or, in the case of `bind`, by using the same port twice, and are not relevant for the kind of faults we analyze.

We focus on exceptions that occur after communication failures, which may result in one or several exceptions [6, 12]. In our tool, we used the SERP bytecode instrumentation library [36] to instrument all these methods (or a subset of choice). Choosing a subset greatly improves efficiency of the model checking process, as shown by experiments. On the other hand, it constitutes an under-approximation, as multiple failures in different method calls are no longer accounted for. This may miss complex errors involving several distinct I/O failures. Therefore, the optimization of instrumenting only certain methods should only be used if coverage of multiple errors is too expensive for a given application.

[2] This proposition assumes that treatment of malformed input is entirely thread-local and of no consequence to the global program state. Input handling can then be tested for one thread by conventional unit testing. If this is not the case, more exceptions can be included for fault injection.

4. Inter-process deadlocks

The previous section describes how possible network faults are simulated using non-determinism. Unfortunately, some of these simulated failures lead to spurious or trivial inter-process deadlocks, which mask real faults, because spurious deadlocks are seen as a failure state by the model checker. Such deadlocks therefore have to be ignored. This section describes how spurious deadlocks are recognized and suppressed automatically. JPF considers the following situation as a deadlock:

> "[JPF checks] the absence of states of the system where no thread can execute: this does not include states where threads are waiting for a timeout. If the system reaches one of these states, it will stay in that state indefinitely: this situation is improperly called a deadlock." [25]

In other words, such a deadlock occurs if all threads are waiting for a lock or are suspended inside a `wait` call. Let T be the set of all threads of a process. For each thread t, function isAlive returns *true* if a thread has been started and not yet terminated [14]. Similarly, let function isWaiting return *true* if a thread is either inside a `wait` or `join` method call, or waiting on a lock. Therefore, a deadlock as above can be defined formally as

$$\text{deadlock} : \forall t \in T : \neg\text{isAlive}(t) \vee \text{isWaiting}(t).$$

4.1. Spurious inter-process deadlocks

The problem is that such deadlocks can also be reported across centralized processes. Such reports can include trivial or spurious deadlocks where a server is blocked inside an operation waiting for a client to connect. Such deadlocks would not be considered as deadlocks in a normal

Class	Method	Fails upon	Remark
`Socket`	constructor	I/O error	exceptions subsumed by plain constructor + `connect`
	`bind`	bind failure/address in use	can be tested conventionally
	`connect`	error during connection	
	`getInputStream`	socket closed/not connected	
	`getOutputStream`	I/O error/not connected	
	`close`	I/O error	
	`shutdownInput/Output`	I/O error	not used by example apps
	`setSocketImplFactory`	I/O error	not used by example apps
`ServerSocket`	constructor	I/O error	exception subsumed by `bind`
	`bind`	bind failure/address in use	can be tested conventionally
	`accept`	I/O error	
	`close`	I/O error	
	`implAccept`	I/O error	not used by example apps
	`getSoTimeout`	I/O error	different exception type for (client) socket; not used by examples
	`setSocketFactory`	I/O error	not used by example apps
`Input-StreamReader`	`read`	I/O error	
	`close`	I/O error	
`Output-StreamWriter`	`write`	I/O error	
	`flush`	I/O error	
	`close`	I/O error	

Table 1. Possible exceptions in Java library methods that are related to I/O errors.

production environment, because of the special nature of a server process, which normally runs inside an endless loop. A server process in this context is defined as a process that accepts incoming requests [33].

Consider the case where an `IOException` prevents a client from connecting to the server. In a test setup, the server is programmed to serve a certain number of clients. If one of these clients cannot complete its connection operation, the server will wait forever for the final client. JPF will report this situation as a deadlock even though the fact that a server waits indefinitely for clients constitutes its normal mode of operation. Only for testing, where the number of connections is finite, the server is expected to terminate. Therefore, a deadlock report where a server process is still waiting for incoming requests is spurious.

For JPF, we have implemented a custom search class that is capable of suppressing these deadlocks. A spurious deadlock occurs if:

1. JPF detects a deadlock because no thread can execute, as defined above.
2. At least one thread waits for a connection and is blocked in `ServerSocket.accept`.

Let inAccept return *true* if a thread is blocked in `ServerSocket.accept`. Then, a true deadlock can be defined by ignoring such spurious deadlocks:

$$\text{trueDeadlock} : \text{deadlock} \wedge \forall t \in T : \neg\text{inAccept}(t)$$

This rule can be generalized to programs using different communication models. In order for the server main loop to terminate, a particular operation by another (client) process is necessary to allow a blocking system call on the server to return. If a failure on the client side prevents this operation from happening, then a deadlock is reported. Under normal operation, the number of clients is potentially infinite, and another process would eventually perform the operation under question. In testing, the number of clients is finite, and therefore deadlocks that arise in such situations are spurious. Such scenarios can be discarded safely.

4.2. True inter-process deadlocks

True deadlocks correspond to situations where even a successful client connection (which allows the server to continue) leads to a deadlock. Such inter-process deadlocks occur when several threads are dependent on the action of another thread. Some of these threads may belong to the same centralized process. Such deadlocks correspond to genuine faults in the program.

Figure 1 shows a scenario where JPF reported a deadlock despite the suppression mechanism described above. The reason is a recursive dependency: The server main thread waits until all worker threads have terminated, such that it can close the open port. In the faulty scenario, one of the worker threads blocks inside a `read` call and waits forever for a response from a client. This response never arrives due to a fault in the client (such as an infinite loop). Because the

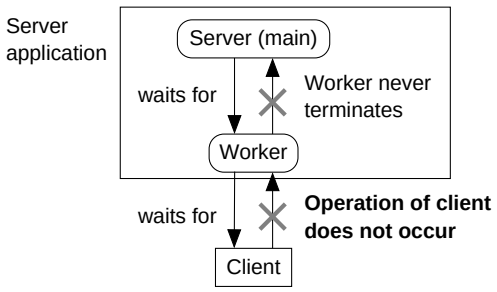

Figure 1. An inter-process deadlock that involves recursive dependencies.

worker thread never terminates, the server main thread is also prevented from terminating.

This deadlock corresponds to a situation where a blocking call never returns. Reliable programs, such as servers, should use a timeout mechanism to prevent a deadlock in such a situation. In the given chat server version, this mechanism was not used. Therefore, this scenario corresponds to a real error.

5. Experiments

The goal of the experiments was to evaluate our deadlock suppression algorithm, and to determine the overhead caused by exception instrumentation. Four categories of operations were chosen, representing two pairs of operations that occur either once per connection or once per message (see Table 2). Experiments were executed using JPF 3.0a [25, 34] on a dual-processor PowerPC G5 (2.7 GHz, 8 GB RAM, 2 GB RAM per Java process, 512 KB of L2 cache per CPU), running Mac OS 10.4.7.[3] For model checking the centralized applications, we first instrumented I/O methods for simulating network failures. The instrumented code, together with the run-time libraries for networking and suppression of spurious deadlocks, was executed in the JPF model checker in order to evaluate the outcome of all possible communication interleavings and failures. The default properties of JPF were verified: uncaught exceptions, assertion failures, and deadlocks.

5.1. Example applications

As realistic applications require heavy manual abstractions, only two examples were available: an echo server and a chat server (including test clients for each). The echo server returns the input received to the client. The chat server sends the input of one client back to all clients, including the one that sent the input. All applications were supplied with two test clients. The echo client sends three lines and expects the same lines back. The chat client sends and reads a single line. While the echo server is a simple application, the architecture of the chat server is fairly complex, involving a main server thread to accept connections and one worker thread per connection. Worker threads use shared data structures to send a message to all other clients (see Figure 2). This architecture is comparable to modern web servers [1].

[3] Attempts were also made to use a recent version (Jan. 2007) of JPF 4, but a regression defect in JPF 4 made it unable to handle one of the case studies that JPF 3.0a was able to run.

Method	Abbr.	Java methods
open	o	`Socket.connect`, `ServerSocket.accept`
close	c	`Socket.close`, `ServerSocket.close`
write	w	`OutputStreamWriter.write`, `flush`
read	r	`InputStreamReader.read`

Table 2. Instrumented methods.

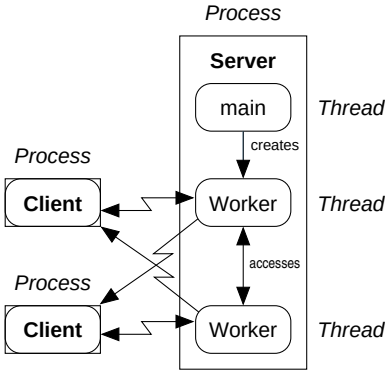

Figure 2. Chat server architecture.

5.2. Overhead measured

Experiments measured the overhead for both applications when one or several of each category of method calls were instrumented to simulate possible failures. Table 3 shows the run times when model checking the echo application. The first four columns indicate which method calls were instrumented for a particular model checking run. The time measured (in seconds, and relative to the uninstrumented case) is shown in the next two columns. Finally, the last four columns show the overhead of a particular type of fault injection relative to the total set of injected faults. In other words, it lists the additional overhead of the argument to ovh(). For instance, ovh(o) refers to the overhead for "open". In the fifth data entry, only "open" is instrumented, and the overhead given by ovh(o) corresponds to the total overhead (factor 1.13). Four rows below, both "open" and "read" are instrumented. The entry in ovh(o) corresponds to

o	c	w	r	time [s]	overhead	ovh(o)	ovh(c)	ovh(w)	ovh(r)
				8.71	1.00				
			1	68.02	7.81				7.81
		1		65.49	7.52			7.52	
	1			19.93	2.29		2.29		
1				9.82	1.13	1.13			
		1	1	156.33	17.95			2.30	2.39
	1		1	128.36	14.74		1.89		6.44
	1	1		133.35	15.31		2.04	6.69	
1			1	69.89	8.02	1.03			7.12
1		1		66.00	7.58	1.01		6.72	
1	1			32.47	3.73	1.63	3.31		
	1	1	1	292.88	33.63		1.87	2.28	2.20
1		1	1	160.07	18.38	1.02		2.29	2.43
1	1		1	304.52	34.96	2.37	4.36		9.38
1	1	1		332.89	38.22	2.50	5.04	10.25	
1	1	1	1	742.19	85.01	2.53	4.64	2.44	2.23

Table 3. Experiments with the echo application. Instrumented method calls are indicated by a "1".

o	c	w	r	time [s]	overhead	ovh(o)	ovh(c)	ovh(w)	ovh(r)
				217.21	1.00				
			1	1787.11	8.23				8.23
		1		780.90	3.60			3.60	
	1			532.89	2.45		2.45		
1				391.39	1.80	1.80			
		1	1	3240.17	14.92			1.81	4.15
	1		1	2824.72	13.00		1.58		5.30
	1	1		1133.77	5.22		1.45	2.13	
1			1	3149.58	14.50	1.76			8.05
1		1		1350.14	6.22	1.73		3.45	
1	1			906.53	4.17	1.70	2.32		
	1	1	1	4338.23	19.95		1.34	1.53	3.82
1		1	1	5426.49	24.98	1.67		1.72	4.02
1	1		1	4745.83	21.85	1.68	1.51		5.24
1	1	1		1975.65	9.10	1.74	1.46	2.18	
1	1	1	1	7698.17	35.44	1.78	1.42	1.62	3.90

Table 4. Experiments with the chat application. Instrumented method calls are indicated by a "1".

the additional overhead of instrumenting "open" compared to instrumenting "read" only. This additional overhead is very small (1.03), because the overhead for instrumenting "read" is already quite high (7.81) compared to the combined overhead of 8.02. Measurements exhibited some variations, so small differences between different tests have no significance.

The key observation from Table 3 is that instrumenting each call individually, and model checking each version separately, is much more efficient than instrumenting all method calls at once, without having missed any errors in our examples. The first "sequential" approach model checks the program four times, with a total overhead of 18.75, while the total overhead of the analysis with all methods instrumented amounts to 85.

Similar results were obtained when model checking the instrumented chat application (see Table 4). The difference between open/close and read/write is relatively small, because only a single message is sent by each client, resulting in few messages being sent across the entire network. Checking each possible failure individually has an overhead of factor 16.08, which is much more efficient than model checking with all failures enabled (overhead: 35.44). Of course, this optimization misses failures that only arise when several network failures occur. However, we have not found any extra faults under these circumstances. This suggests that failures resulting from combined I/O problems are rare in practice.

The impact of injected faults becomes smaller for faulty applications, because there tend to be many execution traces leading to a system failure. Therefore, an error trace is found quickly, after exploring only a small percentage of the state space, as illustrated by Table 5. Even with multiple instrumented failures, the fact that a faulty path is found quickly out-balances the overhead introduced by fault injection. The model checker can even find faults in instances where it would normally run out of its 2 GB of memory. Note that versions where the read call was instrumented did not execute successfully due to a faulty partial-order reduction in JPF. JPF 3.0a sometimes fails to break up transitions inside an infinite loop within an application thread, leading to an infinite loop in JPF itself.

5.3. Faults found in the given applications

Initially, JPF reported spurious deadlocks, which were suppressed by the algorithm described in Section 4. This prompted our definition of spurious deadlocks and the corresponding extension to JPF. Subsequently, three genuine faults were found in the given applications, as shown in the overview in Table 6. The first fault occurs in the same thread as where it was triggered. However, it has consequences for other processes, as will be explained below. The second fault caused another exception in the client process. The third fault resulted in a complex deadlock between several processes and threads, as outlined in Section 4.2.

Figure 3 illustrates the first fault. In the chat server, a worker thread keeps on reading the input until the input is closed or an exception occurs. Each line read is sent to all chat clients. After a chat client quits, it is removed from the global set of chat clients (such that no other worker thread will send data to a closed socket), and its connection is

	2 clients							3 clients						
	o	c	w	time [s]	overhead	# of trans. in counterex.		o	c	w	time [s]	overhead	# of trans. in counterex.
1 served				217.21	1.00	–	1 served				12523.57	1.00	–
			1	5.64	0.03	55				1	41.47	0.00	63
		1	1	8.71	0.04	55			1	1	84.23	0.01	63
	1		1	7.71	0.03	58		1		1	130.03	0.01	67
	1	1	1	14.45	0.07	58		1	1	1	345.46	0.03	67
2 served				27284.65	1.00	–	2 served				out of memory after 80 hours		
			1	1338.91	0.05	136				1	7559.18	n/a	141
		1	1	5783.89	0.21	138			1	1	51928.85	n/a	143
	1		1	1366.68	0.05	140		1		1	15342.20	n/a	146
	1	1	1	5925.03	0.22	142		1	1	1	107681.30	n/a	148

Table 5. Results when using the faulty chat application, which deadlocks when a write call fails on the server side. Instrumented method calls are indicated by a "1". Tests have been run for two and three clients, out of which one or two were served at the same time.

Methods where faults were injected	Application failures
open (`ServerSocket.accept`, `Socket.connect`)	–
close (`Socket` and `ServerSocket`)	Chat: Double removal of worker thread in exception handler of chat server
write (`OutputStreamWriter`)	Echo: `NullPointerException` in client after failed send in server
	Chat: Possible deadlock in worker thread
read (`InputStreamReader`)	–

Table 6. Faults found in the echo and chat server applications.

```
 1 BufferedReader in;
   try {
     in = new BufferedReader(new
       InputStreamReader(sock.getInputStream()));
 5   String s = null;
     // main loop
     while ((s = in.readLine()) != null) {
       chatServer.sendAll(n + ": " + s);
     }
10   // clean-up
     chatServer.remove(n);
     sock.close();
   } catch(IOException e) {
     System.out.println("Worker thread "+n+": "+e);
15   chatServer.remove(n);
   }
```

Figure 3. Double removal of a worker thread.

closed. Unfortunately, the initial implementation contained the following fault: During "clean-up", after a client has been removed from the working set, an exception may occur in `Socket.close` (at line 11). The exception handler will remove the client again (at line 15). Because the entire block of code is not atomic, this creates several problems:

1. Another worker thread may already have been started using the recently-freed slot. In that case, the exception handler of the dying worker thread would remove a different thread. The corresponding new chat client would never receive any response from the server. As no actual run-time error would occur, even model checking would not find this failure, unless a specification exists stating that a certain output from the server is expected.

2. The server maintains a counter for the number of active worker threads. Calling `remove` too often therefore triggers a premature shutdown of the chat server. Certain scenarios even caused the value of the counter to drop below zero. This triggered an assertion failure, which allowed us to find the fault. Fixing the fault involves moving the clean-up code from the `try` and `catch` blocks into a `finally` block.

The second fault found is an example for a *cascading exception*, an exception that provokes a run-time error in another process. Figure 4 illustrates this situation. Assume the client has successfully sent its `N` strings at lines 2 and 3. It then waits for the response of the server (lines 5–7). The server normally sends back its input line by line (lines 5–7 on the right-hand side of Figure 4). If one of these `write` operations fails, fewer lines than expected will be sent, and

Echo client

```
1 int i;
  for (i = 0; i < N; i++)
    out.writeln(i);

5 for (i = 0; i < N; i++)
    System.out.print("Received " +
      in.readLine());
    // in.readLine() may return null
```

Echo server

```
1 String line;



5 while ((line=in.readLine())!=null)

    out.writeln("Echo " + line);
```

Figure 4. `NullPointerException` **in the echo client at line 7 if the server cannot send data.**

method `readLine` will return `null` on the client side. The client code does not check against a `null` pointer, which results in a `NullPointerException` in the chat client.

This failure could also be discovered when model checking the client process in isolation, using a stub for method `readLine`. That stub would return a string or `null` non-deterministically. However, we think that the reason for the `NullPointerException` may not be apparent when model checking one process in isolation. Indeed, a programmer writing the stub or evaluating such an error trace may overlook this failure potential, or treat it as a false positive. Our method delivers a concrete failure scenario.

6. Related work

The classical application domain of model checking consists of the verification of algorithms and protocols [15]. More recently, model checking has been applied directly to software, sometimes even on concrete systems. Such model checkers include the Java PathFinder system [34], JNuke [4], and similar systems [5, 9, 13, 28].

Regardless of whether model checkers are applied to models or software, they suffer from the state space explosion problem: The size of the state space is exponential in the size of the system, which includes the number of threads and program points where thread switches can occur. System *abstraction* offers a way to reduce the state space by merging several concrete states into a single abstract state, thus simplifying behavior. In general, an abstract state allows for a wider behavior than the original set of concrete states, preserving any potential failure states [5, 9].

Most software model checkers analyze a single OS-level process. They cannot handle distributed systems [4, 5, 9, 13, 27, 34]. When using manual program abstraction, I/O operations can be replaced by stubs that mimic each possible operation. However, creation of such stubs is application-specific and can be quite labor-intensive for complex applications. Automation is either imprecise [8, 19] or requires a large model of the underlying functionality [5, 23]. Such an abstraction models all possible outcomes of an interaction with an external process without modeling the entire state and history of that external process. Therefore, traces of external processes are not available. This makes debugging an error trace much more difficult, and may affect performance and precision of the model, because an imprecisely modeled response may result in spurious successor states.

An alternative to stubs consists of lifting the power of a model checker from process level to OS level [24]. The effects of system calls are modeled by hand, allowing several processes to be model checked together without modifying the application code. The difference to our approach is that centralization transforms a multi-process system into a single-process one, while the other tool expands the scope of model checking to several processes. Furthermore, the OS-level model checker does not model communication failures, and has a slower performance than process-level model checkers.

Static analysis [10] uses abstractions similar to the ones used by model checking. Static analysis computes a fix point of all possible behaviors of the abstract program. As in model checking, non-deterministic decisions are explored exhaustively. For verification of correct resource deallocation, there exist static analysis tools that consider each possible exception or failure location [11, 35]. The same effect can be achieved by using a model checker on an abstract version of the program [19]. However, these techniques only cover a part of the program behavior, such as resource handling. For a more detailed analysis, code execution (by testing [2] or by model checking the concrete system involving all processes) is necessary. While efficient comprehensive fault injection mechanisms exist for unit testing [2], test-based fault injection only addresses non-determinism introduced by failed external operations, but cannot account for non-deterministic scheduling.

Centralization includes all processes. Model checking a centralized system therefore provides a comprehensive analysis. The extra information of external processes, which are included in the centralized version, makes an error trace more understandable. When using instrumentation to replay the error trace schedule [26], centralization even makes it possible to replay traces of all processes in a conventional debugger. Furthermore, violations of liveness properties may only appear for particular interleavings of processes (not just threads inside one process) [3] and

could be lost by modular checking. Therefore, centralization is a useful technique for such client/server applications. Centralization has first been proposed and implemented by Stoller et al. [29]. The original approach did not model certain fine points of the Java semantics with full accuracy, and did not cover generic TCP/IP communication. Recent work addressed this shortcoming by covering the Java semantics in more detail, and by adding a model for generic TCP/IP network communication [3]. The original TCP/IP model did not cover communication failures. Our work combines extended centralization with fault injection. It constitutes the first model for network operations that can actually transmit data between (centralized) Java processes, while covering potential network failures, and treating spurious deadlocks correctly.

7. Future work

Future work includes enhancements to our network library model and to the centralization process, in order to reflect more aspects of the Java semantics. Currently, timeout mechanisms are not supported by our model. The addition of shutdown semantics for the termination of a centralized process will also expand the scope of our model. Besides such functional extensions, we also consider optimizations that reduce the overhead in model checking.

Experiments have shown that checking for each kind of failure individually results in an improved performance. We are looking into splitting up instrumentation into smaller subsets to explore this further. For instance, failures could be simulated individually per class or method. Finally, when considering the dynamic program state, even more optimizations are possible. For instance, each possible exception occurrence may be limited to the first few iterations of a loop. Usually, the program state inside a loop does not change significantly, such that the outcome of the exception handler will not differ after one or many loop iterations. Similar reductions are possible per `try`/`catch` block, or per method.

8. Conclusions

By combining centralization with a network model for TCP/IP, networked Java applications can be model checked in any Java model checker. However, certain faults can be detected only if potential network failures are taken into account. Java PathFinder allowed us to model such failures as non-deterministic actions. Some network failures lead to application failures, and may even cause failures in other processes. However, while a model of all combined processes can yield a precise failure scenario, it also increases the state space significantly. Fortunately, most program faults can be observed as a consequence of a single network failure. This allows for checking each kind of failure individually, which is much more efficient than model checking combined failures. Finally, due to the multi-process nature of the applications considered, a model checker may detect spurious deadlocks, which result from failed connection attempts. Such deadlocks have to be suppressed in the model checker itself.

Acknowledgments Thanks go to Shinichi Nagano for the initial version of the chat server.

References

[1] The Apache Foundation, 2006. `http://www.apache.org/`.

[2] C. Artho, A. Biere, and S. Honiden. Enforcer – efficient failure injection. In *Proc. FM 2006*, Canada, 2006.

[3] C. Artho and P. Garoche. Accurate centralization for applying model checking on networked applications. In *Proc. ASE 2006*, Tokyo, Japan, 2006.

[4] C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient Dynamic Analysis for Java. In *Proc. CAV 2004*, volume 3114 of *LNCS*, pages 462–465, Boston, USA, 2004. Springer.

[5] T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proc. TACAS 2001*, volume 2031 of *LNCS*, pages 268–285, Genova, Italy, 2001. Springer.

[6] G. Candea, M. Delgado, M. Chen, and A. Fox. Automatic failure-path inference: A generic introspection technique for Internet applications. In *Proc. WIAPP 2003*, page 132, Washington, USA, 2003. IEEE Computer Society.

[7] E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.

[8] C. Colby, P. Godefroid, and L. Jagadeesan. Automatically closing open reactive programs. In *Proc. PLDI 1998*, pages 345–357, Montreal, Canada, 1998.

[9] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. ICSE 2000*, pages 439–448, Limerick, Ireland, 2000. ACM Press.

[10] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. POPL 1977*, pages 238–252, Los Angeles, USA, 1977. ACM Press.

[11] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *Proc. VMCAI 2004*, volume 2937 of *LNCS*, pages 191–210, Venice, Italy, 2004. Springer.

[12] C. Fu, R. Martin, K. Nagaraja, T. Nguyen, B. Ryder, and D. Wonnacott. Compiler-directed program-fault coverage for highly available Java internet services. In *Proc. DSN 2003*, pages 595–604, San Francisco, USA, 2003.

[13] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. POPL 1997*, pages 174–186, Paris, France, 1997. ACM Press.

[14] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.

[15] G. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2004.

[16] M. Hsueh, T. Tsai, and R. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. *LNCS*, 2072:327–355, 2001.

[18] M. Kim, I. Lee, U. Sammapun, J. Shin, and O. Sokolsky. Monitoring, checking, and steering of real-time systems. In *Proc. RV 2002*, volume 70 of *ENTCS*. Elsevier, 2002.

[19] X. Li, H. Hoover, and P. Rudnicki. Towards automatic exception safety verification. In *Proc. FM 2006*, volume 4085 of *LNCS*, pages 396–411, Hamilton, Canada, 2006. Springer.

[20] T. Lindholm and A. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.

[21] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, USA, 1992.

[22] Microsoft Corporation. *Microsoft Visual C# .NET Language Reference*. Microsoft Press, Redmond, USA, 2002.

[23] M. Musuvathi. *CMC: A model checker for network protocol implementations*. PhD thesis, Stanford University, 2004.

[24] Y. Nakagawa, R. Potter, M. Yamamoto, M. Hagiya, and K. Kato. Model checking of multi-process applications using SBUML and GDB. In *Proc. Workshop on Dependable Software: Tools and Methods*, pages 215–220, Yokohama, Japan, 2005.

[25] NASA. Java PathFinder user guide, 2006. `http://ti.arc.nasa.gov/ase/jpf/JPF-user-guide.html`.

[26] V. Schuppan, M. Baur, and A. Biere. JVM-independent replay in Java. In *Proc. RV 2004*, volume 113 of *ENTCS*, pages 85–104, Málaga, Spain, 2004. Elsevier.

[27] S. Stoller. Model-Checking Multi-threaded Distributed Java Programs. In *Proc. SPIN 2000*, volume 1885 of *LNCS*, pages 224–244, Stanford, USA, 2000. Springer.

[28] S. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proc. RV 2002*, volume 70(4) of *ENTCS*, pages 143–158, Copenhagen, Denmark, 2002. Elsevier.

[29] S. Stoller and Y. Liu. Transformations for model checking distributed Java programs. In *Proc. SPIN 2001*, volume 2057 of *LNCS*, pages 192–199. Springer, 2001.

[30] B. Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 1997.

[31] Sun Microsystems, Santa Clara, USA. *Java 2 Platform Standard Edition (J2SE) 1.5*, 2005. `http://java.sun.com/j2se/1.5.0/`.

[32] A. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.

[33] A. Tanenbaum. *Computer Networks*. Prentice-Hall, 2002.

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

[35] W. Weimer and G. Necula. Finding and preventing run-time error handling mistakes. In *Proc. OOPSLA 2004*, pages 419–431, Vancouver, Canada, 2004. ACM Press.

[36] A. White. SERP, an Open Source framework for manipulating Java bytecode, 2002. `http://serp.sourceforge.net/`.