Efficient Model Checking of Networked Applications

Cyrille Artho[1], Watcharin Leungwattanakit[2], Masami Hagiya[2], and Yoshinori Tanabe[3]

[1] Research Center for Information Security (RCIS), AIST, Tokyo, Japan,
c.artho@aist.go.jp
[2] University of Tokyo, Tokyo, Japan,
{watcharin,hagiya}@is.s.u-tokyo.ac.jp
[3] Research Center for Verification and Semantics (CVS), AIST, Tokyo, Japan,
tanabe.yoshinori@aist.go.jp

Abstract. Most applications today communicate with other processes over a network. Such applications are often multi-threaded. The non-determinism in the thread and communication schedules makes it desirable to model check such applications. When model checking such a networked application, a simple state space exploration scheme is not applicable, as the process being model checked would repeat communication operations when revisiting a given state after backtracking. We propose a solution that encapsulates such operations in a caching layer that is capable of hiding redundant communication operations from the environment. This approach is both more portable and more scalable than other approaches, as only a single process executes inside the model checker.

Key words: Software model checking, network communication, software testing, caching

1 Introduction

Networked software is complex. It is often implemented as a concurrent program, using threads [21] to handle multiple active communication channels. This introduces two dimensions of non-determinism: Both the thread schedule of the software, and the order in which incoming requests or messages arrive, cannot be controlled by the application. In software testing, a given test execution only covers one particular instance of all possible schedules. For exhaustive analysis, it is desirable to model check software, to ensure that no schedules cause a failure.

Model checking explores, as far as computational resources allow, the entire behavior of a system under test by investigating each reachable system state [9], accounting for non-determinism in external inputs, such as thread schedules. Recently, model checking has been applied directly to software [2, 5, 7, 10, 12, 13, 23]. However, conventional software model checking techniques are not applicable to networked programs. The problem is that state space exploration involves backtracking. After backtracking, the model checker will again execute certain parts of the program (and thus certain input/output operations). However, external processes, which are not under the control of the model checker, cannot be kept in synchronization with backtracking.

Backtracking would result in repeated communication operations, causing direct communication between the application being model checked and external processes to fail.

Our work proposes a solution to this problem. It covers all input/output (I/O) operations on streams and is applicable when I/O operations always produce the same data stream, regardless of the non-determinism of the schedule. While our solution is implemented for Java programs, the ideas are applicable to any software model checker or programming language supporting TCP-based networking [22]. A large number of programs uses TCP-based communication and is amenable to such verification. Previous work introduced the idea of caching I/O communication traces for model checking [4]. This paper extends that work and contributes the following achievements:

1. We show the necessity of matching requests to responses and introduce a solution for this problem. We also amend problems arising with more complex protocols that were not solved in the initial solution [4].
2. We introduce a full implementation of the caching approach, which is capable of model checking complex networked Java applications such as an HTTP server. Its performance is orders of magnitudes faster than previous work based on centralization of applications [1, 19].

This paper is organized as follows: An intuition for our algorithm is given in Section 2, while Section 3 formalizes our algorithm. The implementation is of our approach is described in Section 4, and experiments are given in Section 5. Section 6 describes related work. Section 7 concludes this paper, and future work is outlined in Section 8.

2 Intuition of the Caching Algorithm

2.1 Software Model Checking

Model checking of a multi-threaded program analyzes all non-deterministic decisions in a program. Non-determinism includes all possible interleavings between threads that can be generated by the thread scheduler or through possible delays in incoming communication. Alternative schedules are explored by storing the current program state in a *milestone*, and backtracking to such a milestone, running the program again from that state under a different schedule.

Figure 1 shows an example to illustrate the problem occurring with communication operations. The program consists of two threads, which print out their thread name. Consider all possible schedules for this simple program. When both threads are started, either T1 or T2 may execute first, depending on the schedule. In testing, only one of these two outcomes will actually occur; in model checking, both outcomes are explored.

Figure 2 illustrates the state space exploration of this simple program when executing it in a software model checker. Boxes depict the set of threads that can be scheduled for execution. Transitions are shown by arrows and labeled with the output generated in that transition. After thread initialization, the scheduler has to make a choice of which thread to run first. As both choices should be explored, the model checker saves the complete program state at this point in a milestone. Assume the state space exploration then picks T_1. After its message is printed, only T_2 remains. After execution of T_2,

Thread 1	Thread 2

`void run() {`
`  print "[T1]";`
`}`
	`void run() {`
	`  print "[T2]";`
	`}`

Fig. 1. Example program to illustrate backtracking in a software model checker.

both threads have finished, terminating the program. The model checker subsequently backtracks to a previously stored program state. After backtracking, the model checker explores the other possible schedule. The operations of T_1 and T_2 are executed again, in reverse order.

This simple example shows how the same operations are repeated after backtracking. As long as repeated operations target the program heap, their effect is usually consistent, as the entire program state is restored from a milestone. Therefore, repeated operations have the same effect on the restored memory as the original ones, as long as all components affected are captured by milestones. In this example, printing to the screen was repeated. While the state of the console cannot be backtracked by the model checker, this redundant output is usually ignored. However, input/output operations that affect other processes cannot be treated in this direct way.

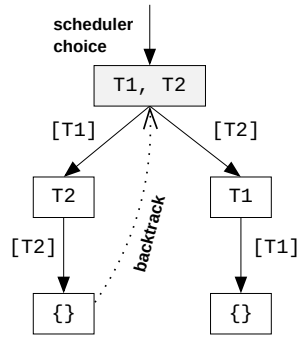


Fig. 2. State space exploration for the example program.

2.2 Handling Redundant Actions after Backtracking

Effects of input/output (I/O) operations cannot be captured in milestones, as the *environment* of the system outside the current process is affected. When model checking an application that is part of a distributed system using multiple processes, external processes are not backtracked during model checking. Thus, two problems arise:

1. The application will re-send data after backtracking. This will interfere with the correct functionality of external processes.
2. After backtracking, the application will expect external input again. However, an external process does not re-send previously transmitted data.

One possible solution to this problem is to lift the power of a model checker from process level to operating system (OS) level. This way, any I/O operation is under control of the model checker [17]. However, this approach suffers from scalability problems, as the combination of multiple processes yields a very large state space. The same scalability problem arises if one transforms several processes into a single process by a technique called *centralization* [19]. With a model for TCP, networked applications can be model checked, but the approach does not scale to large systems [1, 3].

Stub-based approaches replace external processes with a simplified representation [6, 8, 11, 16]. In such work, external processes are not executed, and the stub returns a (previously known) result mimicking the behavior of the real implementation.

Our approach differs in that it only executes a single process inside the model checker, and runs all the other applications externally in their real implementation. Figure 3 depicts the overall architecture of the system. Let "system under test" (SUT) denote the application executing inside the model checker. Execution of the SUT is therefore subject to backtracking. Redundant externally visible operations, such as input/output, have to be hidden from external processes. External processes are called *peers* and can implement either client or server functionality, as defined in [22]. In our solution, a special cache layer intercepts any network traffic. This cache layer represents the state of communication between the SUT and external processes at different points in time. After backtracking to an earlier program state, data previously received by the SUT is replayed by the cache when requested again. Data previously sent by the SUT is not sent again over the network; instead, it is compared to the data contained in the cache. The underlying assumption is that communication between processes is independent of the thread schedule. Therefore, the order in which I/O operations occur must be consistent for all possible thread interleavings. If this were not the case, behavior of the communication resource would be undefined. Whenever communication proceeds beyond previously cached information, new data is both physically transmitted over the network and also added to the cache.

As an example, a simple program involving two threads is given in Figure 4. Each thread first writes a message to its own (unique) communication channel and then reads

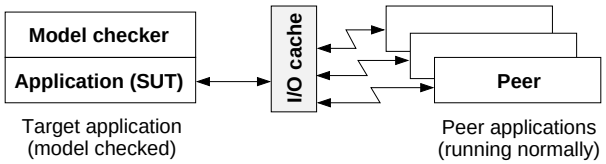


Fig. 3. Cache layer architecture.

Thread 1	Thread 2

`void run() {`
`  conn1.write("1");`
`  r1 = conn1.read();`
`}`
	`void run() {`
	`  conn2.write("2");`
	`  r2 = conn2.read();`
	`}`

Fig. 4. Example program communicating with peer processes.

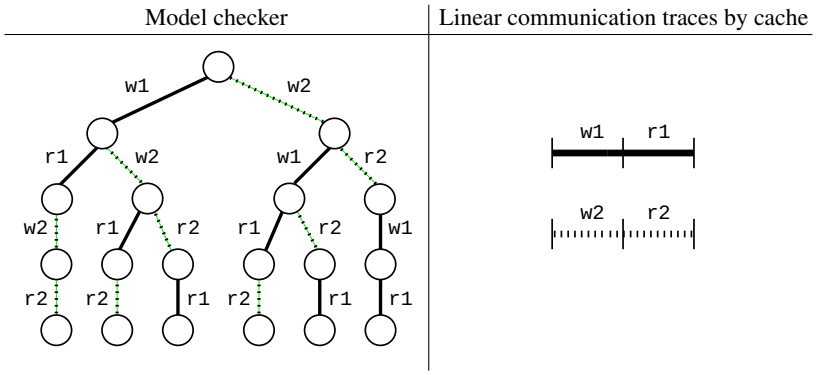


Fig. 5. State space exploration inside the model checker (left) and communication traces stored by the cache layer (right).

from it. Both communication channels interact with external processes (or with the same external process using two independent connections). Both threads run concurrently, so any interleaving of their operations is possible. Their combined state space is explored inside the model checker, as shown on the left side of Figure 5. In that figure, write and read operations are abbreviated by w_i and r_i, respectively, with i denoting the thread ID. As can be clearly seen, execution of all possible interleavings results in multiple backtracking operations, and each communication operation is repeated several times. However, in a given execution trace, each operation only occurs once. Operations within each thread are totally ordered, resulting in a partial order on I/O operations across all threads. Our cache layer takes advantage of this. The cache keeps track of I/O operations during the entire state space exploration and maintains a linearization of each communication trace. Each communication trace reflects the (total) order of messages within each communication channel. Identical physical communication operations of each execution trace are executed only once. The results of these operations is cached in a linear communication trace, as shown on the right side of Figure 5.

In our approach, all processes involved are executed; however, only a single process runs inside the model checker. Our approach therefore combines the (relative) scalability of running a single (usually multi-threaded [21]) process inside the model checker with the benefit of finding implementation errors when analyzing real applications. Indeed, peer processes may even run on external hosts and require features such as database access that a given model checker cannot support. As our approach only exhaustively searches the state space of *one* process at a time, it has to be applied once to each application: Each process is run once in the model checker, with its peers as external processes. Thanks to our cache layer, external processes do not have to be backtracked. In essence, our cache produces the same results as a perfect stub. The scalability improvement comes from the fact that the state space of one process is exponentially smaller than the state space of multiple processes. Our work differs from stub generation [6] in that we generate the corresponding data on the fly, without requiring a previous execution, and that we can handle peer processes running on other hosts or platforms. For a more detailed comparison, see Section 6.

2.3 Extension to More Complex Protocols

So far, the design described is sufficient for simple protocols consisting of one (atomic) request and response [4]. However, keeping track of data stream positions is not sufficient for protocols where several requests and responses are interleaved, and for requests that consist of multiple parts. For example, in HTTP, a GET request consists of two lines, one line containing a URL, and another empty line to mark a complete request.

As an example that will also serve to test the performance of the approach, we take a simple protocol where the server returns the *n*th character of the alphabet requested. A request consists of a number between 1 and 26, and a newline character (`\n`). Assume that a client running this protocol is model checked, and the server runs as a peer application (see Figure 6). The client consists of a request thread, sending the request in two steps to the server, and a response thread, which reads the (atomic) response when available.

Assume that in the first schedule, the model checker executes both request steps before the response is read. Both parts of the communication are then cached by the cache layer as described above. Now, the model checker tries a different schedule and backtracks to the state after the first half of the request, and executes the response thread. Clearly, the response should not (yet) be returned by the cache layer! Correctness of the cache layer can therefore only be ensured by polling the server after each request, in order to verify if a matching response exists for a complete request.

Previous work has implemented the basic I/O cache idea [4], but the design and implementation had several flaws.[1] This paper presents the first complete, fully working implementation, which has successfully been applied to complex software such as a concurrent HTTP server.

3 Formalization of the Caching Algorithm

A *request* is a message sent (written) to the peer, and a *response* is a message received (read) from the it. Our approach depends on two key assumptions:

[1] In the original design, communication channels were not always correctly assigned to threads. Furthermore, non-atomic messages could not be handled at all. Finally, the requirement to cache `close` operations as originally proposed [4] is redundant and can be dropped.

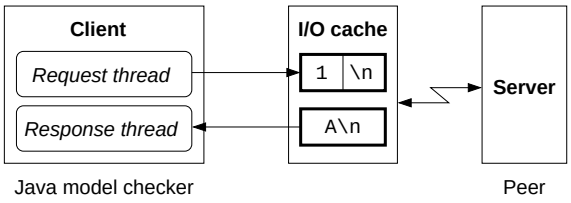


Fig. 6. A multi-threaded client requiring the cache to match requests to responses.

1. Deterministic peer responses: For each request sequence, there exists a corresponding unique response sequence.
2. Consistent application behavior: For each thread and each socket, the same requests are issued regardless of the thread schedule.

These assumptions allow for model checking a system when only one execution of peer processes is observed. Based on these two assumptions, we define our execution semantics of the state space exploration of input/output operations, which also verifies the second assumption.[2] The following definitions assume an assignment operator $:=$ as in computer programs, allowing for updates of variables and functions.

3.1 Stream Abstraction

A *data stream* s is a finite sequence of messages m: $s = \langle m_0, \ldots, m_{|s|-1} \rangle$. A *stream pointer* $sp_s = i$ refers to a particular index i in the message sequence of a given data stream s. A *communication trace* $t = \langle req, resp, \text{limit} \rangle$ consists of two data streams, a request stream *req* and a response stream *resp*, and a function $\text{limit}(sp_{req}) : sp_{resp}$. This function takes a request pointer and returns its corresponding response pointer.

Programs operate on a set T of communication traces t (which correspond to streams or sockets in a given programming language). One communication trace is associated to each socket. We augment the normal program state consisting of a global heap and several threads that each carry their own program counter and stack, by a pair of stream pointers $\langle sp_{req}, sp_{resp} \rangle$ for each communication trace t. This *extended program state* is managed by the model checker and subject to backtracking. Definitions below assume that stream pointers are changed by backtracking, while data in T remains unchanged.

In our approach, all communication traces have to be consistent with the first seen communication trace: In any possible program execution, there has to be one unique trace $\hat{t}$ such that for all thread schedules, $t = \hat{t}$ when the program terminates normally.[3] Consistency is checked by verifying message data of repeated requests against previously cached request data.

3.2 Execution Semantics

Model checking of a program using I/O is performed as follows: When progressing beyond previously cached messages, all operations are directly executed, using the functionality provided by the standard library. The result of that library function, using the correct set of parameters, will be denoted by `lib_`*xy*$(\ldots)$, where *xy* represents the original library function. Any error codes returned by library function will also be returned by our model. Our library model treats errors as if they occurred deterministically, e. g. as a result of invalid parameters. As our approach requires a deterministic response,

[2] As responses are cached, the first assumption cannot be verified at run-time. However, inconsistent peer behavior is detected if the peer itself is model checked in a separate analysis run.

[3] We omit treatment of input/output errors here. In such cases, program behavior diverges.

nondeterministic errors arising from communication failures of the underlying network cannot be covered. For brevity, we omit error handling here.[4]

Without loss of generality, we assume that each message has size 1. Multiple messages from a client may be required in order to elicit a server response. Conversely, at a given state, a response may consist of multiple messages of size 1. Operations always work on a given trace t, which is omitted in the following definitions for brevity.

Helper function `pollResponse` (see Algorithm 1) serves to check whether a given request produces a response from the peer. Responses are checked for and cached proactively, in order to correctly treat programs where responses are processed by an independent thread. Whenever a program reads from the same connection later on, cached response data will be used to determine the size of the response, i. e., the limit of the incoming message. Function `pollResponse` checks if data is available on the physical connection, and stores that data. It also updates function limit, which denotes the extent of the response received. This function is always defined up to the position where a request has been cached, i. e., up to $|req| - 1$.

Function `write` behaves in two possible ways, as shown in Algorithm 2: if previously recorded communication data extends beyond the current position, then current data is compared to previously written data. If no cached data exists at the current position, data is physically sent to the peer, and the peer is polled for a response. Reading data returns previously cached data, if a corresponding response had been cached by `pollResponse` after the last message was sent. If no data is available, function `read` blocks (see Algorithm 3).

When opening a connection (through `connect` or `accept` on the client and server side, respectively), the library function normally returns a socket object *sock* representing as a handle to that connection. As the behaviors of `connect` and `accept` are very similar, we will subsume both functions with `open` in this discussion. Our model library returns the same socket, but also maintains a communication trace object t for each socket. Subsequent communication via *sock* is cached in t . When backtracking to a point before the creation of *sock* (and t), *sock* is discarded, but t is merely marked as unused, such that re-execution of function `open` will retrieve the previously seen communication trace. Algorithm 4 summarizes this functionality. (Our implementation requires a consistent order in which sockets are created by different threads; relaxation of this criterion is subject to future work.) Closing a socket marks a communication trace t as unused, which allows it to be used again after backtracking, as shown in Algorithm 5.

3.3 Example Execution Scenario

Figure 7 shows an example state space exploration of the alphabet client described in Section 2. The first column depicts state space exploration, with the current state shown in black. The state of trace t and function limit in the current state are shown in the next two columns. In the protocol used by this example, newline characters that complete a request or response have been omitted for simplicity. The example client uses only one

[4] Approaches that backtrack all processes involved, such as centralization, can inject communication failures into the simulation, but suffer from poor scalability [3].

Algorithm 1 Function `pollResponse`. n is the size of the request.

$i := \text{limit}(sp_{req})$ (limit at current position)
while (data is available) do
  $resp_i :=$ `lib_read`$(\ldots)$
  increment i
$\text{limit}(sp_{req}+n) := i$ (limit at new position)

Algorithm 2 Function `write`; d is the payload to be written.

$i := sp_{req}$
if $i < |req|$ then (check against cached data)
  abort if $req_i \neq d$
else (physically write new data and cache it)
  call `lib_write`$(\ldots)$
  $req_i := d$
  call `pollResponse`$(n = 1)$
increment sp_{req}

Algorithm 3 Function `read`.

$i := sp_{resp}$
if $i = \text{limit}(sp_{req})$ then (no data available)
  suspend current thread until data available
  call `pollResponse`$(n = 0)$
increment sp_{resp}
return $resp_i$ (cached data)

Algorithm 4 Function `open`.

create new socket object *sock*
if unused communication trace t_{old} available then
  $t := t_{old}$
else
  open new physical connection for *sock*
  $t :=$ new communication trace
mark t as used
bind t to *sock* (subsequent operations on s will access communication trace t)
call `pollResponse`$(n = 0)$ (certain protocols return data without requiring a request)

Algorithm 5 Function `close`, operating on socket *sock* and its trace t.

$i = sp_{req}$
if $i < |req|$ then (premature close)
  abort
if $i = |req|$ then
  if physical connection for *sock* is open, close it
  mark t as unused

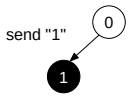
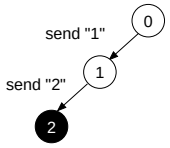
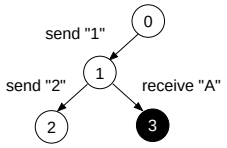
State space exploration	Trace t	limit	Remarks
	0 1 2 ↓ 1 A ↑	$0 \to 0$ $1 \to 1$	After an initial request "1", function `pollReponse` records server response "A". If the reader thread were scheduled now, it could read that response.
	0 1 2 ↓ 1 2 A B ↑	$0 \to 0$ $1 \to 1$ $2 \to 2$	A second request results in another response and another update of limit. If the reader thread were to access responses now, it would advance sp_{resp} twice, until the limit at the current request position (2) is reached.
	0 1 2 ↓ 1 2 A B ↑	$0 \to 0$ $1 \to 1$ $2 \to 2$	After backtracking to the state after the first request, sp_{req} is backtracked to 1. The reader thread is scheduled, and accesses response data. The persistent mapping in limit ensures that only the first response is returned by the cache.

Fig. 7. An example demonstrating the interaction between sp_{req}, sp_{resp}, and limit.

connection, so there exists only a single trace t. Trace t is illustrated as a two-row table with a request record (top), response record (bottom), and stream pointers pointing to a particular position in each data stream. Note that in this example, no data is returned after a connection is made; this results in an initial entry $0 \to 0$ for limit.

3.4 Limitations of Our Approach

Any program whose written messages fulfill the consistency criteria defined above can be model checked successfully using our approach. However, there are classes of programs that are normally considered to be valid, for which our criteria are too strict. This includes software that logs events to a file or network connection. For this discussion it is assumed that logging occurs by using methods `open`, `write`, and `close`. Assume further that actions of each thread can be interleaved with actions of other threads, which include logging.

If log entries of individual threads depend only on thread-local data, they are independent of each other. In such a case, different correct interleavings of log entries can occur without violating program correctness. If log data is sent over a single shared communication channel, occurrence of different message interleavings violates the criterion saying that written data at a specific position must be equal for all thread interleavings. Such programs can therefore not be model checked with our approach, unless some messages are exempted from the consistency check.

On a more general level, applications where communication depends on the global application state are not applicable to our approach. For instance, a chat server where the server sends a message back to all clients currently connected will violate our consistency criterion. In such a server, one connection is maintained per client. As the

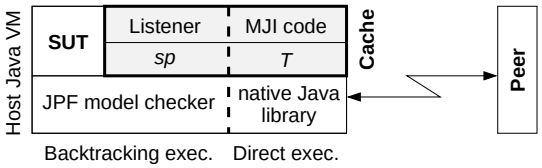


Fig. 8. The implementation architecture.

order in which incoming messages are processed differs between schedules, several interleavings of incoming messages are possible. As a consequence of this, the sequence of outgoing message varies as well across different schedules. The resulting inconsistency prevents our approach from being applicable. Applications where communication varies across schedules can still be model checked with our approach by abstraction over messages data, utilizing the same sequence of messages for each thread. In the chat server case, all messages are replaced by a string constant to allow our approach to proceed.

A large number of programs fulfills our requirement of deterministic communication traces. This includes any service-oriented architecture where several clients are served concurrently, but independently. Programs that violate the given invariant can still be model checked by using application centralization [1, 3]. Centralization can also cope with slow responses, which occur when the network is slow or a response requires extensive computation on the peer.

In the caching approach, polling assumes that a response is ready shortly after a request has been sent. In our implementation, a short delay is induced in the SUT to wait for the response. Still, it cannot be guaranteed that a response is always received when logically possible. This shortcoming could only be overcome by inspection of the server, to determine whether a request has been fully processed. This could be implemented by code instrumentation in all peer systems, and is subject to future work.

4 Implementation

The I/O cache implements the Java library API for network I/O [15]. It fully controls messages exchanged between the system under test (SUT) in the model checker and external processes. Implementation details depend on the model checker and the way it allows access to internal events. We chose Java PathFinder (JPF) [23], a system to verify executable Java bytecode programs, as our model checker. It was chosen because it is openly available and features a stable API for extending model checker actions.

JPF is an execution framework for software verification purposes. Its latest version supports user extensions via two extension mechanisms, *listeners* and the *Model Java Interface*. During the execution of JPF, its internal state changes after each executed instruction. A listener subscribes to certain internal events, such as instruction execution or backtracking. It is notified by the JPF when such an event happens. The listener uses this information to verify application properties or customize search patterns. In our work, the listener notifies the cache layer when a state transition takes place.

The Model Java Interface (MJI) separates the Java virtual machine (VM) of JPF from the underlying Java VM of the host computer system, called *host VM*. The difference between JPF and host VM involves the backtracking function of the JPF. When an instruction is executed by the host VM, no backtracking occurs. An instruction executed by JPF is model checked with specified properties and subject to backtracking.

The architecture is summarized by Figure 8. The dashed vertical line shows which part of the cache code (shown by a solid box) execute within JPF, and which part execute directly on the host Java VM. In Java, the network library is implemented in package *java.net.* Key classes include *Socket*, *InetAddress* and *InetSocketAddress.* These classes contain native methods that physically communicate over the network. For these methods, backtracking is not possible. However, JPF allows users to replace standard library classes with user-defined classes. This extension is implemented through the MJI mechanism by creating two classes, called *model class* and *native peer* class, for each replaced standard class.

When the SUT is model checked by JPF, user-defined classes are executed instead of the standard library code. The native peer class, however, is still executed on the host JVM. Note that the model class can propagate methods where backtracking is not needed to the corresponding native peer class. User-defined classes can also utilize the MJI mechanism in order to avoid backtracking. By replacing the original Socket class with our customized version, communication between the SUT and its environment is redirected to the cache layer. The cache layer returns input and output communication channels to the client and behaves like the real peer.

5 Experiments

For evaluation of our approach, we conducted a number of experiments. To facilitate automation, these experiments were performed on a single computer. In principle, peer applications could also be run on a different host when using our I/O caching approach. While this may be necessary to model check software working together with third-party clients or services, having all processes under our control greatly facilitated the setup.

For model checking, startup of the SUT and the remote application have to be synchronized. Otherwise, the client may attempt to contact the server before it is ready. Premature client startup can be avoided in two ways:

1. Extra control code could be added to the client, ensuring that the server is ready. For instance, the client could retry a communication attempt in the event of failure.
2. Starting the client is delayed after starting the server. This allows the server to initialize itself and be ready for communication.

The second approach may start the client prematurely, if the delay is too small. In practice, it has the advantage that the SUT does not have to be modified, and proved to work quite well. If the client is started too early, this can be seen immediately from the client log. Restarting the client later therefore fixes this issue. Automation of this could be achieved by using operating system utilities such as `trace`, `strace`, or `truss` [14], to supervise system calls.

The Java PathFinder model checker [23], version 4 revision 353, was used for our experiment. This version features great performance improvements over version 3. Unfortunately, revision 354 introduced a new bug that prevented us from running the centralized test applications on newer revisions.[5]

5.1 Example Applications

Table 1 gives an overview of the examples used as benchmarks. The *daytime client* connects to a server, which sends a fixed string back to the client.[6] While the server is single-threaded, the client launches concurrent requests. In this case, the client is more complex than the sequential server, and was the focus of model checking.

Jget [18] is a multi-threaded download client, which issues a number of concurrent partial download requests in addition to the main request. Depending on which task finishes first, Jget either uses the entire file downloaded by the main thread, or it assembles the file from the pieces returned by the partial downloads. Essentially, the worker threads are in a (controlled) race condition against the main thread. This creates the challenge of ensuring that the complete file is received when the program shuts down. In order to allow for the necessary concurrency and partial downloads, we augmented an existing example web server [20] with ability to serve parts of a file. The resulting system proved too complex for model checking with JPF, so it was abstracted to a slightly simpler system. The abstract system has all strings of HTTP reduced to very short literals, eliminating the necessity of string parsing.

The *alphabet client* generates two threads per connection: a producer and a consumer thread. They communicate with the *alphabet server*. The server expects a string containing a number, terminated by a newline character, and returns the corresponding character of the alphabet. In this case, both the client and the server are multi-threaded, and were model checked.

The chat server, described in more detail in [1], sends the input of one client back to all clients, including the one that sent the input. The original chat client transmits its ID at the beginning of each message. This ID causes a mismatch in the cached server input when the order of server worker threads is reversed after backtracking. Therefore, the client ID was stripped from the transmitted messages when using our cache approach. (Code that builds a compound string using that ID was left in the system to maintain the same complexity for comparison purposes.)

5.2 Results

All experiments were run on an Intel Core 2 Duo Mac 2.33 GHz with 2 GB of RAM, running Mac OS 10.4.11. JPF was given 1 GB of memory, a limit that was never exhausted, and a time limit of two hours. The standard properties used by JPF were verified: We checked against deadlocks, uncaught exceptions, and assertion violations. The

[5] While we have reported that bug more than a year ago, we were unable to produce a small test case that reproduces it. The large size and run-time of the test producing a failure has made it impossible to determine the exact location of the problem.

[6] For the purpose of model checking, the "date" used was hard-coded in the replacement class for `java.util.Date`.

Table 1. Example applications used.

Application	Description
Daytime client	Returns the current time (RFC 867).
Jget client, version 0.4.1	Multi-threaded, multi-connection HTTP client.
Web server (for Jget)	Multi-threaded, multi-connection HTTP server.
Alphabet client/server	Returns the nth character of the alphabet.
Chat server	Sends messages of one client to all clients.

original version of Jget (0.4.1) included a few initial bugs in the calculation of ranges, and an inefficient design that relied on busy-waiting. These initial flaws were fixed prior to further analysis. When model checking the final abstracted version of Jget, we found that Jget erroneously reported an incomplete download, even though the entire file was received. Furthermore, the abstract version of the web server may exhibit a (handled) null pointer exception due to sloppy error handling in our own extension for download ranges. Investigation of the problems in Jget (whether due to a bug in the original application or in the abstraction) remains subject to future work; therefore, we did not investigate other settings for Jget.

No program contained a critical error that would have terminated the state space search by JPF and resulted in an error message. JPF therefore investigated the full state space, allowing a comparison of the size of both models. Where possible, we compared the results of our approach to model checking all the clients and the server processes using centralization [1]. Note that in our new caching approach, the client and server are analyzed separately, while in centralization, they are analyzed together. Our approach therefore requires at least two model checker runs, but allows for analyzing much larger programs, as the state space explosion of combined processes is avoided. However, our approach sacrifices analysis of all possible combinations of client/server behaviors for efficiency. This is most obvious for the Jget/web server pair, where only one possible schedule on the client side is exhibited when model checking the server.

Table 2 shows the results of all our experiments. The daytime server and the chat client do not exhibit any concurrency; therefore, an analysis by JPF was not necessary, as it would not reveal any results that cannot be obtained by ordinary testing. The table is divided into three parts: A description of the test case, and the results of the centralization and caching approaches. The test setup includes the name of the application, the number of connections or threads used, and a scale parameter for the last two cases. For the alphabet server, the number of messages per connection was varied. For the chat server, limiting the number of concurrently accepted clients was another way to curb the state space.

The second and third part of the table show the results for the two different approaches. The I/O caching approach is analyzing the client and server separately, with the peer process(es) running outside the model checker. As the peer processes consume next to no resources compared to the model checker, only the time spent in the model checker is shown. For completed runs, the number of program states analyzed by the model checker is shown as well. “New” states refer to distinct program states; “revisited” states refer to redundant states that resulted in backtracking.

Table 2. Results of our experiments. The table lists the applications with the number of connections or threads used first. The "scale" parameter refers to the number of messages per connection in the alphabet server, and the number of concurrent connections accepted in the chat server case. Results using the centralization approach are compared to analyzing the client and the server side separately, using our I/O caching approach.

Appl.	# conn./ threads	Scale param.	Centralization			I/O caching approach					
						Client			Server		
			Time	States		Time	States		Time	States	
			[mm:ss]	new	revisited	[mm:ss]	new	revisited	[mm:ss]	new	revisited
daytime	2	n/a	0:57	33123	71608	0:01	204	229	Server implementation is not concurrent.		
	3		55:39	715941	4726181	0:03	2140	3983			
	4		> 2 h			0:38	25744	67638			
	5					10:16	356122	1216424			
	6					> 2 h					
Jget	2	n/a	> 2 h			0:50	31015	50640	0:14	7636	14250
alphabet	2	1	112:24	JPF error		0:04	3631	8711	0:02	102	96
		2	> 2 h			0:11	11386	27978	0:02	226	214
		3				0:25	27211	67871	0:03	394	372
		4				0:50	54246	136362	0:03	610	574
		5				1:28	96253	243045	0:03	874	820
	3	1	> 2 h			4:42	226646	904025	0:03	1072	2069
		2				30:14	1383304	5764677	0:06	4011	7587
		3				113:38	5017624	21325338	0:10	9347	17347
		4				> 2 h			0:18	17947	32951
		5							0:28	30579	55755
	4	1	> 2 h			> 2 h			0:16	12014	65646
		2							1:16	70308	200088
		3							3:39	208228	579576
		4							8:26	483812	1331400
		5							16:31	967092	2643048
	5	1	> 2 h			> 2 h			3:14	134700	537111
		2							26:06	1185717	4516299
		3							97:46	4407369	16449707
		4							> 2 h		
	6	1	> 2 h			> 2 h			42:25	1487902	7432964
		2							> 2 h		
	7	1	> 2 h			> 2 h			> 2 h		
chat	2	1	2:12	50284	143439	Client implementation is not concurrent.			0:03	112	81
		2	77:08	714830	2668937				0:08	4100	4935
	3	1	89:28	1819345	6786956				0:03	112	81
		2	> 2 h						0:08	4100	4935
		3							3:52	115036	221174
	4	1	> 2 h						0:03	112	81
		2							0:08	4100	4935
		3							3:56	115036	221174
		4							> 2 h		

Finally, we have also verified that the I/O caching approach finds synchronization problems present in faulty versions of the chat server that were investigated in earlier work [1]. These errors can be found if two or more clients are present. With both I/O caching and centralization, JPF immediately finds a schedule exhibiting a data race. However, the I/O cache allows for more in-depth analysis of the revised chat server, where the absence of faults can be confirmed for up to three clients with our new approach.

5.3 Summary

Our experiments in Table 2 show that model checking with our I/O caching approach is orders of magnitudes faster than model checking using centralization. Our new approach is capable of analyzing interesting and complex system such as concurrent client/server implementations with up to at least three concurrent connections. The fact that concurrency problems can be ruled out at that scale gives good confidence that they are also absent for a larger number of connections, even though no formal proof for this exists. We therefore think that our approach constitutes a very important breakthrough in scalability for model checking networked software.

The reason for this improved scalability is that our approach executes only a single process inside the model checker, analyzing all interleavings of its threads. This avoids analysis of the product of all state spaces of all processes. When analyzing a complete system consisting of multiple processes, each process should be analyzed in turn in the model checker using our cache, with other processes running as peers. The resulting complexity of all analysis runs will correspond to the sum of the state spaces of each process. This is vastly smaller than the product thereof. Our caching approach inherently cannot analyze all possible schedules of peer processes, making it less sound than aggregation-based approaches such as centralization [1]. However, this sacrifice allows scalability to systems that were previously out of reach, making it more useful in practice.

6 Related Work

Software model checkers [2, 5, 7, 10, 12, 13, 23] store the full program state (or differences to a previously stored state) for backtracking. They are typically implemented as explicit-state model checkers. Milestone creation and backtracking operations occur many times during state space exploration. This causes operations to be executed several times when a set of schedules is explored. Such exploration does not treat communication behavior accurately, as described in the introduction.

One solution is to model I/O operations as as stubs. In this approach, communication operations are modeled by shared memory, semaphores, and channels. Peer processes are included in the resulting system [8, 16] or modeled by a (possibly abstracted) environment process or stub [5, 7, 11]. Abstraction in the environment ensures scalability but can lead to false positives requiring refinement of the model. In most tools, stubs for communication operations and environment processes are provided manually, at a level of abstraction suitable to the problem at hand [8, 11, 16]. The process of generating the optimal abstraction can be automated but is of course still constrained by computational resources [5, 7]. Our approach is fully automated, requiring no abstraction. However, applicability relies on the equality of communication traces between schedules, a property that is checked at run-time.

A general solution to model checking multiple communicating processes is to lift the power of a model checker to operating system (OS) level. This way, the effect of I/O operations are visible inside the model checker. An existing system that indeed stores and restores full OS states is based on user-mode Linux [17]. That model checker uses

the GNU debugger to store states and intercept system calls. The effects of system calls are modeled by hand, but applications can be model checked together without modifying the application code. In that approach, the combined state space of all processes is explored. Our approach analyzes a single process at a time inside a model checker, while running other processes normally. Our approach is therefore more scalable but requires programs to fulfill certain restrictions. On the technical side, OS-level model checkers intercept communication at device level, where the network device itself is wrapped. We intercept communication at library call level.

External processes could be backtracked in tandem with the system under test, for instance, by restarting them [8, 13]. In existing implementations, one central scheduler controls and backtracks several processes, effectively implementing a multi-process model checker [13, 16]. Like all approaches controlling multiple processes inside the model checker, it incurs a massive state space explosion.

In an alternative approach, multiple processes are analyzed in a single-process model checker after applying program transformation. *Centralization* transforms multiple processes into threads, creating a single-process application [19]. This allows several processes to run in the same model checker, but does not solve the problem of modeling inter-process communication (input/output). Recent work modeled network communication in the centralized model where all processes are executed inside the model checker [1, 3]. Communication and backtracking of centralized processes all occur in a single model checker. Other work has implemented this approach in a similar way, but sacrificed full automation in favor of manual instrumentation of communication operations [6]. That tool has another mode in which it can run, allowing for replacing peer processes with stubs. In this approach, a program that just repeats previously observed communication contents is used as peer. Recent work has gone into automating this process by a tool that captures communication in a corresponding stub program [6]. When using stubs from a previous recorded communication, the assumptions mentioned in Section 3 also have to hold. In contrast to stub usage, our approach eliminates the need for an intermediary stub program. Our approach records communication and replays it on the fly, in one module. Furthermore, it even allows model checking of applications where external processes are not running on a platform that the model checkers supports.

Our approach curbs state space explosion by only running a single process inside the model checker. It builds on previous work [4] that introduced the idea of I/O caching. Previous work has several shortcomings and flaws that prevented the idea from working on realistic examples. First, it did not always correctly associate sockets to traces, and traces to threads. Second, it lacked the crucial idea of proactive response caching. Because of this, the implementation showed problems on more complex examples. Our work is the first one to complete to formalization and implementation of the I/O caching idea [4] proposed earlier.

7 Conclusions

When model checking communicating programs, processes outside the model checker are affected by communication but not subject to backtracking. Executing different

branches of a non-deterministic decision is not applicable to external communication. With traditional approaches for model checking software, input/output operations had to be subsumed by stubs, or multiple processes had to be executed inside the model checker. The former is difficult to automate, while the latter suffers from scalability problems.

We defined special caching semantics for stream-based I/O, which includes network communication. This generates the corresponding behavior of a stub on the fly, during model checking. If program behavior is independent of the execution schedule, such a program can be model checked using our cache layer semantics. Unlike most previous work, we can handle implementations using standard network libraries without any manual intervention, while eliminating some scalability issues of some related approaches. We also have a fully working and very scalable implementation of our algorithm for the Java PathFinder model checker, and we could successfully model check several complex applications where multiple clients interact with in parallel with a server.

8 Future Work

Future work includes possible relaxations of the completeness criteria defined, regarding the order of I/O operations and socket creations. Specifically, certain interleaved write actions on the same communication channel should be allowed, such as log entries.

Current work focuses on model checking applications communicating by TCP. This protocol is reliable in the sense that message order is preserved, and messages are not lost. However, we think that the main concept can be modified and be applied to I/O failures, where communication is interrupted. This would also make it possible to model check applications communicating by lightweight but unreliable protocols such as the User Datagram Protocol (UDP). We will also work on the issue of slow responses, by implementing a tool that instruments the peer application in order to signal readiness for new requests to the model checker. Finally, it remains to be seen how far our approach, which has so far been tried on service-oriented client-server systems, is applicable to peer-to-peer systems or multicast protocols.

References

1. C. Artho and P. Garoche. Accurate centralization for applying model checking on networked applications. In *Proc. 21st Intl. Conf. on Automated Software Engineering (ASE 2006)*, pages 177–188, Tokyo, Japan, 2006. IEEE Computer Society.
2. C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient Dynamic Analysis for Java. In *Proc. 16th Intl. Conf. on Computer Aided Verification (CAV 2004)*, volume 3114 of *LNCS*, pages 462–465, Boston, USA, 2004. Springer.
3. C. Artho, C. Sommer, and S. Honiden. Model checking networked programs in the presence of transmission failures. In *Proc. 1st Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE 2007)*, pages 219–228, Shanghai, China, 2007. IEEE Computer Society.

4. C. Artho, B. Zweimüller, A. Biere, E. Shibayama, and S. Honiden. Efficient model checking of applications with input/output. *Post-proceedings of 11th Int'l Conf. on Computer Aided Systems Theory (Eurocast 2007)*, 4739:515–522, 2007.
5. T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proc. 7th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 268–285, Genova, Italy, 2001. Springer.
6. E. Barlas and T. Bultan. Netstub: a framework for verification of distributed Java applications. In *Proc. 22nd Intl. Conf. on Automated Software Engineering (ASE 2007)*, pages 24–33, Atlanta, USA, 2007. ACM.
7. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, 2004.
8. S. Chandra, P. Godefroid, and C. Palm. Software model checking in practice: an industrial case study. In *Proc. 24th Intl. Conf. on Software Engineering (ICSE 2002)*, pages 431–441, New York, USA, 2002. ACM.
9. E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.
10. 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. 22nd Intl. Conf. on Software Engineering (ICSE 2000)*, pages 439–448, Limerick, Ireland, 2000. ACM Press.
11. J. Dingel. Computer-assisted assume/guarantee reasoning with VeriSoft. In *Proc. 25th Intl. Conf. on Software Engineering (ICSE 2003)*, pages 138–148, Washington, USA, 2003. IEEE Computer Society.
12. M. Dwyer, J. Hatcliff, M. Hoosier, and Robby. Building your own software model checker using the Bogor extensible model checking framework. In *17th Int'l Conf. on Computer Aided Verification (CAV 2005)*, volume 3576 of *LNCS*, pages 148–152, Edinburgh, UK, 2005. Springer.
13. P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th ACM Symposium on Principles of Programming Languages (POPL 1997)*, pages 174–186, Paris, France, 1997. ACM Press.
14. I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. In *Proc. 6th Usenix Security Symposium (SSYM 1996)*, pages 1–13, San Jose, USA, 1996. USENIX Association.
15. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.
16. M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: a pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, 2002.
17. Y. Nakagawa, R. Potter, M. Yamamoto, M. Hagiya, and K. Kato. Model checking of multiprocess applications using SBUML and GDB. In *Proc. Workshop on Dependable Software: Tools and Methods*, pages 215–220, Yokohama, Japan, 2005.
18. S. Paredes. Jget, 2006. `http://www.cec.uchile.cl/~sparedes/jget/`.
19. S. Stoller and Y. Liu. Transformations for model checking distributed Java programs. In *Proc. 8th Intl. SPIN Workshop (SPIN 2001)*, volume 2057 of *LNCS*, pages 192–199. Springer, 2001.
20. Sun Microsystems, Santa Clara, USA. *A simple, multi-threaded HTTP server*, 2008. `http://java.sun.com/developer/technicalArticles/Networking/Webserver/`.
21. A. Tanenbaum. *Modern operating systems*. Prentice-Hall, 1992.
22. A. Tanenbaum. *Computer Networks*. Prentice-Hall, 2002.
23. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.