Journal of Advanced Research in Applied Sciences and Engineering Technology

Journal homepage:
https://semarakilmu.com.my/journals/index.php/applied_sciences_eng_tech/index
ISSN: 2462-1943

Software Model Checking Distributed Applications: A Hybrid Approach

Hing Ratana[1], Sharifah Mashita Syed-Mohamad[2,*], Chan Huah Yong[1]

[1] School of Computer Sciences, Universiti Sains Malaysia, 11800 Gelugor, Penang, Malaysia
[2] Faculty of Ocean Engineering Technology and Informatics, Universiti Malaysia Terengganu, 21030 Kuala Nerus, Terengganu, Malaysia

ARTICLE INFO

Article history:
Received 6 June 2023
Received in revised form 1 March 2024
Accepted 13 April 2024
Available online 22 May 2024

Keywords:
Software model checking; distributed system; Java PathFinder (JPF); Interprocess communication (IPC); backtracking; state-space explosion

ABSTRACT

Developing reliable distributed systems poses many challenges such as concurrency, failure handling, and scalability. It is due to the non-deterministic execution of threads within processes, and communication means. Formal verification methods, such as model checking, have been used to ensure the reliability of safety-critical systems. This technique systematically explores the complete behavior of the system under test (SUT), investigating each reachable state with different thread schedules. Recent software model-checking tools, employing cache and centralization, have been applied to distributed systems. The caching technique can only check one process at a time, while the centralization technique verifies all processes simultaneously. In the centralization technique, two "ArrayByteQueue" buffers are utilized to store communication and process data byte-by-byte. However, during the backtracking process, the read and write operations, involving data insertion and removal from the queue, become resource-intensive. As a consequence, existing interprocess communication (IPC) models encounter computational limitations and experience a rapid state space explosion. To address these challenges, our work proposes the remodeling of IPC models by introducing a request and response tree structure to store communication data. Additionally, we employ pointers to navigate through the data during the backtracking process. Through experimental evaluations, the proposed implementation choices have demonstrated significantly improved performance across various metrics. By incorporating the request and response tree, we enhance the efficiency of storing communication data, while the use of pointers optimizes navigation during backtracking. This remodeling of IPC models shows promise in mitigating computational limitations and state space explosion, thereby enhancing the model-checking process in distributed systems. Our research contributes to advancing the field of model checking in distributed systems and offers potential solutions to the challenges associated with resource-intensive read and write operations during the model-checking process.

1. Introduction

Modern societies nowadays rely on software in all aspects of daily living and interacting with each other, including safety-critical systems, electric cars, smart home devices, and so on. Software or

* *Corresponding author.*
E-mail address: s.mashita@umt.edu.my (Sharifah Mashita Syed-Mohamad)

https://doi.org/10.37934/araset.45.2.152167

applications such as Google Search, Facebook, YouTube, Netflix, Shopee, Lazada, Grab, Foodpanda, and many more are distributed. A distributed system consists of a group of nodes. Those nodes can be generically referred to as physical smartphones, personal computers, smart devices, televisions, and so on which cooperate by exchanging messages over communication links to achieve some tasks [1].

This article focuses on software model checking of distributed systems. Catching bugs as early as possible. Big companies like Google have spent most of their cost focusing on finding and fixing, processes, and procedures to debug incidents in Google's distributed systems [2]. Many large software systems consist of multiple processes spread across multiple computer processors, executing independently. While this provides the advantages of increased performance and scalability it also makes such systems much harder to test due to partial failure and asynchrony. Partial failure refers to the components in distributed applications that can fail along the way, resulting in incomplete results or data. Asynchrony is the indeterminateness of ordering and timing within a distributed system that often leads to solutions with a high degree of complexity. Avoiding distributed system bugs also requires reasoning about the integration between nodes and must tolerate the failure of the underlying hardware. In addition, the probability of human error in either design, implementation, or operation also contributes to system bugs. Therefore, developing a reliable distributed system is a very challenging task.

Formal methods, particularly model-checking, can produce rigorous and automated reliability proofs for hardware and software systems. The area has focused on distributed systems for two main reasons [3]. First, distributed programs are error-pruned, because programmers must consider all possible effects induced by different scheduling of events. Second, testing, which is widely used for certifying sequential programs, tends to have low coverage in distributed settings, because bugs are usually difficult to reproduce. They may happen under very specific thread schedules, and the likelihood of taking such corner-case schedules may be very low. Therefore, automated verification techniques represent crucial support in the development of reliable distributed applications.

Model checking is a technique to detect property violations in a concurrent system by exploring every possible execution path [4]. Accordingly, every possible state of the system is checked against given properties. This technique is very useful for the quality assurance of safety-critical systems and core algorithms/protocols of large systems. Model checking was originally developed for hardware verification, but the concept of state space exploration has been applied to a wide range of software systems as well.

In the traditional software model-checking process, a system to be verified is abstracted into an input language supported by the model checker, a software tool that performs model-checking [5]. The model checker then generates a directed graph that represents the state space of the system. It traverses the graph and checks if the desired properties hold at every state. This activity is referred to as state space exploration. After verification of the model, the system is usually implemented in a programming language such as C or Java, based on the verified model.

Although model checking originally requires the system to be checked written in formal languages, e.g., PROMELA, recent work in the community applies model checking directly to source codes; this activity is often called software model checking. Direct verification of real codes increases confidence in software safety. The fact that the system design meets a specification does not imply that the implementation does. Many concurrency-related bugs are still introduced by programming mistakes during the implementation phase, such as race conditions, deadlocks, and assert violations. Verification in the design phase cannot guarantee the absence of bugs in the final deliveries. The software model-checking community focuses on the analysis of real implementations, written in mainstream programming languages such as Java or C [6].

This article focuses on utilizing software model checking in the context of distributed systems using Java. The research aims to enhance existing techniques by combining the cache approach with centralization. The centralization technique verifies all processes within the distributed systems simultaneously [7]. This technique employs two 'ArrayByteQueue' buffers to store communication and process data at a byte-by-byte level. However, the write and read operations, which involve extracting data from the queue during the model-checking process, pose significant resource-intensive challenges due to the backtracking process. This leads to a rapid expansion of the state space and computational limitations in existing interprocess communication (IPC) models. Therefore, our study proposes a novel remodeling of the existing IPC models by incorporating a request and response tree for storing communication data and employing pointers to navigate through the data during the backtracking process. The paper is organized into the following sections: Section 2 provides an overview of recent advancements in model-checking distributed systems, including caching and centralization techniques. Section 3 presents our methodology and the proposed solution. Section 4 offers insights into the experiments conducted and the results obtained from the proposed method. Finally, in Section 5, we conclude our work.

2. Literature Review

Initially, model checking was developed as a means to verify algorithms, protocols, and system models. However, its scope has expanded to include direct verification of software systems or implementations. With the increasing complexity of code compared to models, there has been a growing interest in model-checking networked applications. This paper focuses on explaining the techniques relevant to model-checking networked applications.

Model checking involves exhaustively and systematically examining the behavior of a system under test (SUT) by analyzing each reachable state for different thread schedules. The SUT undergoes backtracking during the model-checking process. In the context of model-checking network applications, the SUT often engages in repeated messaging (input/output operations) with external processes. However, these external processes, which are not controlled by the model checker, lack coordination with the SUT's backtracking mechanism, resulting in a breakdown of direct communication between the SUT and the external processes.

Among model checkers, Java PathFinder stands out as the only one capable of verifying distributed systems, as demonstrated in the 2019 software verification competition contribution [8,9]. Various approaches, such as cache and centralization, have been proposed to tackle the challenges of model-checking distributed systems [7,10-17].

2.1 Cache Approach

The cache approach model checks a single process inside the model checker tool once at a time and runs all the other processes externally in their native environment. A process is a self-contained execution environment and has its resources such as memory and other resources, whereas threads run within a process and share the process runtime resources. In the cache-based approach, the SUT and peers denote the single process inside the model checker and the external processes, respectively. The SUT executed by the model checker is subjected to backtracking, while external processes run normally.

Figure 1 illustrates the overall architecture of the cache approach for model-checking network applications. The model checker executes the single-process SUT in exhaustive ways thus making repeated requests. The special cache layer intercepts all the communications between the SUT and

its peers. It represents the state of communication at different points in time. Data previously received by the SUT is held in the cache and if the SUT makes the same request, the response will be sent from the cache rather than incur all the processing involved in resending the request to the peers.

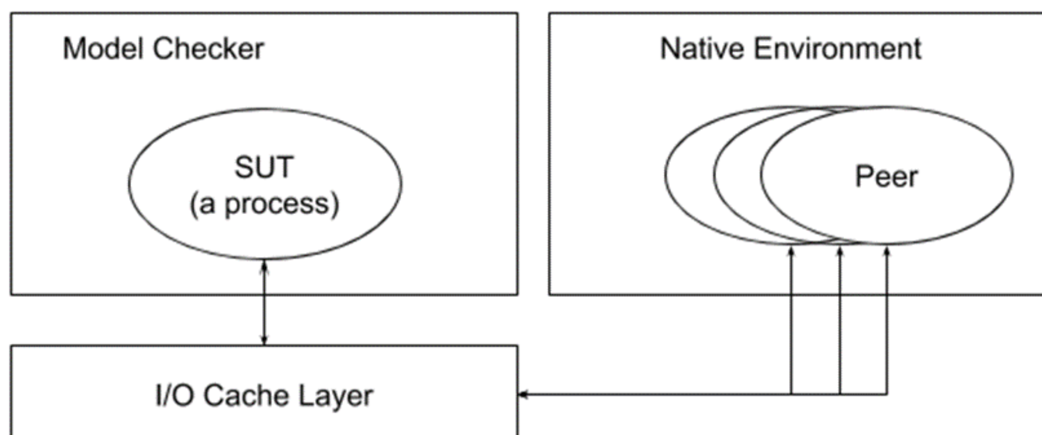

Fig. 1. Overall architecture of cache approach

The main challenge of this approach is the synchronization between the single-process SUT and its peers since the SUT is subjected to backtracking by the model checker, and the model checker does not have any control over its peers. During model checking SUT, the SUT may resend data which might interrupt the correct behavior of the peers, and the peers may not send the correct data back to the SUT. A special cache layer has been developed to solve these problems. Existing cache-based techniques address this problem by introducing a special cache layer between the SUT and its peers for state synchronization.

Initial work by Artho *et al.,* [10] proposes a solution for model-checking network applications by developing a special caching layer for stream-based input/output (I/O). They introduce the idea of I/O caching via deterministic communication. The method is called linear-time cache. A linear cache models communication data using a pair of arrays called a request array and a response array. The linear data structure stores request and response messages, respectively. The initial solution works if the I/O operations of the SUT always produce the same data stream regardless of the indeterminacy of the thread schedules. The communication between the SUT and its environment must be independent of the thread schedules. For instance, if the client sends a sequence of characters to the server, the server is supposed to send the same sequence of characters back to the client, regardless of the thread schedules. If this is not the case, the behavior of the communication traces would be undefined.

The later work by Artho *et al.,* [11] extends the idea of caching I/O traces to a wider range of network applications by developing the tree data structure that allows diverging communication traces between different thread schedules. This concept is called branching-time cache. The technique captures the communication traces and stores them in a tree data structure. The major advantage of branching cache is to allow the non-determinism of thread schedules within the SUT, but it does not allow non-determinism within the peers. For this approach, the SUT at least can send sufficient different data from the previously observed ones.

To allow non-determinism within peers, the proposed work by Artho *et al.,* [11] combines branching-time cache with process checkpointing [12]. This is called the hybrid approach. Process checkpointing environment can run, pause, and replay the peers at any point in time. During model checking of SUT, the checkpointing idea can be incorporated when the SUT requires the

synchronization of data from the peers, at those points, checkpointing can play and replay the peers' states accordingly to the requests from the SUT. By doing this, this concept gives a broader range of model-checking network applications.

2.2 Centralization Approach

The centralization approach is to model-check all processes within a model checker. These techniques can be applied at the SUT Level, OS Level, and model checker level. Figure 2 shows the overall architecture of existing centralization techniques: (a) SUT level; (b) OS level; and (c) Model checker level. At the SUT level, the processes are transformed into one main process, so each process is mapped into a thread, and the model checker verifies the main process. At OS-level centralization, it does not involve transforming the SUT. Instead, all the processes are running on top of the virtualization tool, and the model checker tool is extended to capture the state of the virtualization tool for state-space exploration. Finally, the centralization approach can be applied at the model checker level. This technique is to extend the model checker so that the tool can capture multiple processes within the tool itself.

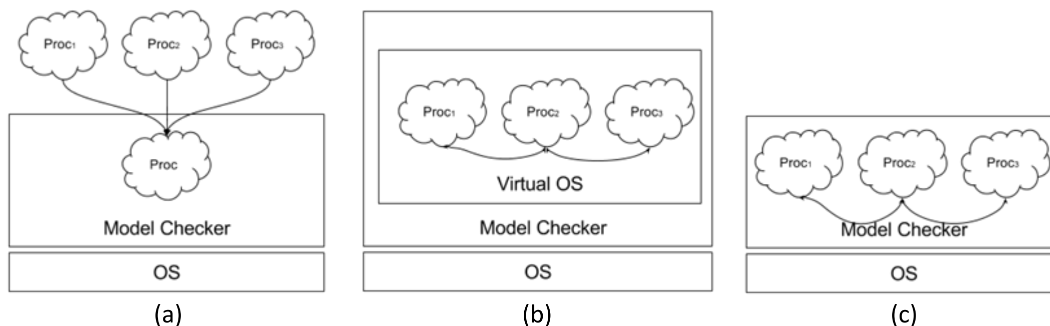

Fig. 2. Overall architecture of centralization approach; (a) SUT level, (b) OS level, (c) model check level

2.2.1 SUT level centralization

Initial work by Stoller and Liu [13] applying the centralization technique at the SUT level. They propose the concept of transforming processes into a single process by replacing remote method invocation (RMIs) with local ones that simulate RMIs. In addition, Stoller and Liu developed the CentralizedThread class that extends Thread and initializes an instance of field type integer to denote process id. By doing this, they can map each process into a thread, and each thread communicates with the other via the simulated local RMIs. Stoller and Liu initially introduced centralization at the system level, which involves merging all distributed Java application processes into one and replacing remote method invocations with simulated local ones. Java RMI, which enables the communication between Java processes, is used for objects in different JVMs to call each other's methods remotely.

Later work by Artho and Garoche [14] provides a more accurate transformation of processes into a single process, and they also address some of the limitations of previous work by Stoller and Liu. In contrast to previous work, Artho and Garoche perform bytecode instrumentation which applies to systems compatible with a newer version of Java, and, in addition, their technique is also applicable to applications that use sockets for communication.

Like the solution proposed by Stoller and Liu, the approach presented by Artho and Garoche replaces calls to methods that end the process, such as "System.exit(int)", by throwing the "java.lang.ThreadDeath" exception. However, their approach also doesn't offer a mechanism for

killing other threads within the terminating process. If all remaining threads are daemons, their solution suggests ignoring the failures in a dead process. They do provide a mechanism for freeing resources after processes terminate, but starting shutdown hooks requires manual modification of the centralized program. On the other hand, the RMI model proposed by Stoller and Liu cannot be extended to support communication via sockets, while the centralization technique of Artho and Garoche applies to applications that communicate through sockets. They adopt centralization at the SUT level, and their approach builds upon the work of Artho and Garoche while addressing some of its limitations. Specifically, they address the class version conflict issue, which arises when different processes use classes with the same name but different bytecode. Unlike other proposed methods, their approach doesn't require processes to use the same version of Java classes. Before implementing their centralization algorithm, they resolve class version conflicts between processes by renaming classes that have the same name but different bytecode. However, their approach does not resolve class version conflicts for Java system libraries, as it does not apply to native methods and code that utilize the reflection API.

Ma *et al.,* [15] approach provides a method to end processes by ending all their threads, which they accomplish by using the Java thread interruption mechanism. However, this requires adding code to the processes. Additionally, their approach does not handle starting shutdown hooks when a process terminates.

Finally, SUT-level centralization has been proposed by Barlas and Bultan [16]. They are mainly focusing on environment generation by introducing a framework called Netstub. The Netstub API requires users to manually develop how the environment should be generated to accommodate the SUT during model checking. In addition, Netstub also allows model checking a process at a time. The Netstub environment can generate network events that are perceived by the SUT.

2.2.2 OS level centralization

In centralization at the OS level, the processes are running on a virtualization tool; therefore, this approach does not require transforming the SUT. This approach requires the extension of the model checker's scope to capture the state of the virtualization tool. The major challenge for this technique is the state space explosion. Since the SUT processes are running on top of the virtualization tool and the model checker must cover all the processes including virtualization tool processes, this will lead to the exponential growth of states.

Nakagawa *et al.,* [17] develop a model-checking framework based on this approach. Their proposed framework can execute very close to the actual model-checking execution environment. They combine the user-mode Linux and the GNU debugger (GDB) to save and restore the entire Linux state. GBD can support several programming languages including Java. Processes are running on a virtualization tool and once non-determinism is detected within a process, the state of the OS and any possible execution paths are computed and explored by the tool.

2.2.3 Model checker level centralization

The major challenges with centralization at the model checker level are managing the state space within the model checker, modeling internal communication between local threads, and possible covering of language API and classes.

A recent centralization approach has been implemented at the model checker level in the initial work by Shafiei and Mehlitz [7]. They develop multi-process JVM for JPF which allows model checking of distributed Java applications. To address the problems of class confliction, static functions, and

static fields, the new multi-process in JPF modifies the class loaders in JPF. The processes are mapped as a group of threads. During the initialization, each new thread is created by the SUT automatically. To capture scheduling points inside JPF, new communication models have been developed based on network API calls. This technique has been implemented into a JPF extension called jpf-nas.

Shafiei and Mehiltz [7] proposed a centralization approach for model checking multiple communicating processes. This approach handles the challenge of separating data between processes by using a new class-loading model. The approach is implemented within the Java PathFinder (JPF) model checker, which allows for modeling inter-process communication (IPC) and exploring potential exceptional control flows caused by network failures. The state space of distributed applications is reduced using a partial order reduction (POR) algorithm, which is proven to preserve deadlocks. The approach also includes an automatic way to capture interactions between the system being verified and external resources.

3. Methodology

The centralization approach plays a crucial role in the model checker level and serves as the fundamental framework for JPF in model-checking distributed systems. However, for effective model-checking of these systems, it is necessary to have models that capture the communication between processes. Previous research has introduced a technique to capture communication data at a byte-by-byte level by utilizing two buffers for each connection [7]. These buffers store data that has been sent but not yet received by the other endpoint, employing a cyclic queue known as 'gov.nasa.jpf.util.ArrayByteQueue' to store raw bytes. However, the write and read operations, responsible for adding and removing data from the queue, impose significant resource demands due to the need for backtracking through explored states during the model-checking process. As a result, this leads to a rapid expansion of the state space and computational limitations within existing IPC models.

Our work proposes a revised approach to the existing IPC models by incorporating the cache approach. Instead of handling communication data at a byte-by-byte level, the proposed method focuses on multi-byte input and output. The communication data is stored in a request and response tree, acting as a cache. During the backtracking process, the request and response pointers navigate through the data, allowing for more efficient exploration and analysis.

This section discusses the results obtained from the surface pressure measurement study. The effects of angle of attack, Reynolds number, and leading edge bluntness are discussed in the next sub-section.

Figure 3 shows the overall architecture of the proposed remodeling of IPC. The revised IPC model is an extension of JPF that can model-check distributed systems. It is a combination of the cache and centralization features. It uses the multi-process VM from the JPF core, the connection manager, the connection, and the distributed scheduler. However, from the preliminary experiments, the read and write operations of the communication data byte-by-byte are not suitable for model-checking distributed systems since the states of the programs can grow exponentially and it is a resource-intensive operation during the backtracking process. The cache request and response tree is the most appropriate solution, as it only navigates request and response pointers during the backtracking process.

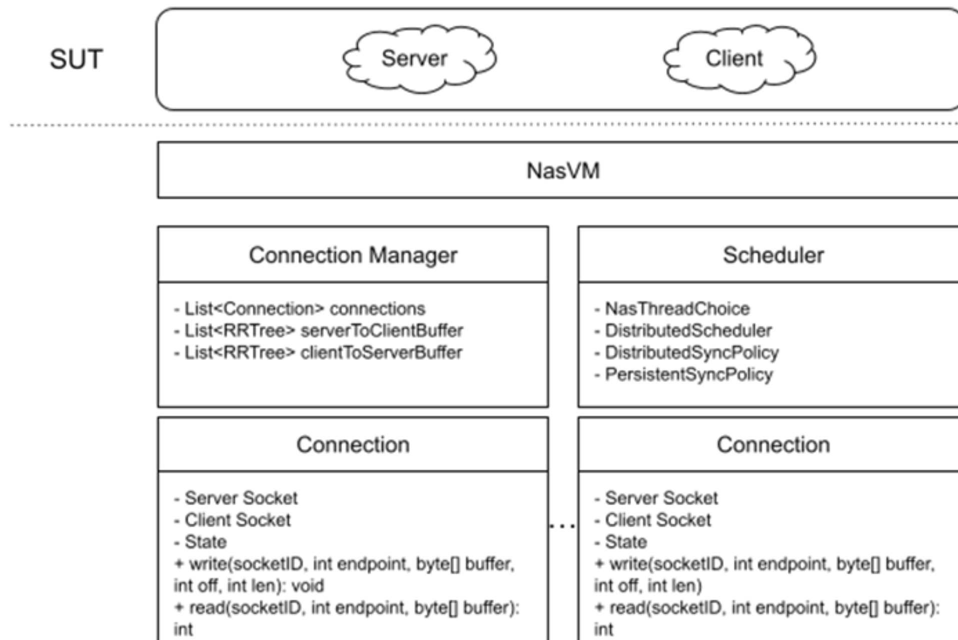

Fig. 3. Overall architecture of the proposed remodeling of IPC

The following describes the proposed redesign of IPC which includes important components such as the NasVM, the connection manager, the connection, and the distributed scheduler.

(i) NasVM

This class serves as the primary starting point. It begins by extending the multi-process VM from the JPF core system and then launches the connection manager component. Next, it starts the server process first and then the client process, and finally sets the global scheduling point for both processes.

(ii) Connection Manager

This component is created only once by the NasVM during the entire lifespan of the model checker. It is responsible for managing the connections between processes. One connection is established through two active sockets, for example, between the server and the client. The server typically starts by specifying an address and port number and then waits for the client to connect. If the client connects to the server successfully, both processes can then send and receive data from each other. When multiple threads or processes create more connections, the connection manager must manage multiple connections. The two most important variables are the buffers (communication data). In the previous centralization approach, the buffers are part of the connection object. In this work, the two buffer variables are created only once during the entire lifespan of the model checker because the data acts as a cache in the connection object. During the backtracking process, instead of creating and deleting data, the request and response pointers are used to navigate the request and response tree. The connection manager creates two buffers, which are represented as a list type of the request and response tree (RRTree) shown in Figure 3. One buffer stores the data sent from the server to the client, and the other buffer stores the data sent from the client to the server. To keep track of the states of the connections and buffers, the connection manager has to implement a listener from the JPF core system. This is to listen to the JPF core system when it goes through bytecode executions. To achieve this, the connection manager has to

implement a state extension client listener from the JPF core, which consists of three main functions: getStateExtension(), restore(), and registerListener(). The getStateExtension() function listens to the JPF core when it finds a new state, restore() is used when the JPF restores a state, and registerListener() registers the connection manager component to the state extension client.

(iii) Connection

The server socket object consists of a passive socket, in which the server process creates the server object and waits for the client to connect, after the successful connection, the server creates an active socket. The same thing happens to the client process, after the successful connection, the client creates an active socket. The connection manager considers this as one connection; hence, multiple server/client pairs will create multiple connections. The proposed design includes a connection object that includes a server socket, a client socket, the connection state, and read and write operations. However, the connection object cannot store the two buffers of the communication data due to the significant computation and state space explosion that occurs during the backtracking process when using the data structure. Therefore, in this design, the two buffers are left out. Instead, the read and write operations will generate two RRTrees for storing and receiving data, and utilize the request and response pointers to navigate the data. This approach helps to avoid the computational and state space issues associated with including the buffers in the connection object.

(iv) Distributed scheduler

JPF has a scheduler that is designed to observe the various ways in which threads are scheduled within a process. During the search for the state space of the SUT, JPF employs this scheduler to analyze and explore different sequences of concurrent transitions. These transitions could lead to varying behaviors of the process if executed in different orders. The JPF scheduler operates at a local level, which means it only deals with operations that are specific to a single process and do not involve communication between multiple processes. This approach is commonly referred to as the JPF local scheduler. For processes to communicate with one another, they must have access to a communication channel. However, the behavior of a distributed system can vary depending on the order in which processes access these channels. To account for the different orderings of concurrent transitions that involve interprocess communications, a mechanism is required. The IPC model tackles this issue by incorporating a global scheduler, which allows for the modeling of different possible orderings of communication events between processes.

3.1 The Caching Technique

Instead of storing the communication data in the “ArrayByteQueue”, the proposed method uses the request and response tree. The communication data is not deleted during the backtracking process, and it uses the request and response pointer to navigate through the data. Initially, the request and response tree (RR tree) consists solely of the root node, and when the request and response pointers are first established, they also point to this root node.

To illustrate the set of operations mentioned above, consider the following example. If the client sends “0”, the server replies “a”. After that the client sends “1”, and the server replies “b”. The first step is to create a new RR tree by invoking the main constructor and providing the socketID as a parameter. Since this is the first connection, the socketID equals zero. Then, to distinguish between a client or server connection, it is necessary to call setConnectionID(int endpoint). Assuming the client

end is 567 and the server end is 763. In the beginning, the tree only contains one node, which is the root node, and to ensure that the request and response pointers are set to the root node, the addPointers() function must be called. If the target application attempts to transmit the character "0", the client operates the addSendEvent("0"). This operation is also addReceiveEvent(“0”) to the server tree. After that, the server operates the addSendEvent(“a”). This operation will add the “a” as the request to the server tree, and the addReceiveEvent(“a”) method adds the response “a” to the client tree as shown in Figure 4.

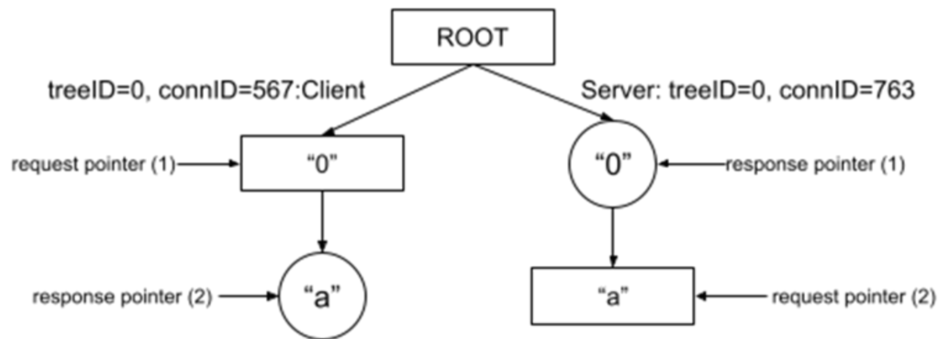

Fig. 4. RR tree illustration after the client sends “0” and the server replies “a”

If another message in the target program tries to send another message "1" to the server, the addSendEvent("1") method is executed, and the addReceiveEvent("1") method adds the response message to the server tree. Once the transmission is complete, the RR tree for both the client and the server tree is displayed in Figure 5.

The RR tree comprises a tree ID, connection ID, a collection of nodes, and two HashMaps for tracking send and receive pointers, which are represented by a HashMap<Integer, Integer>. To ensure that each connection has a unique tree, the tree ID is assigned by the socket ID. Furthermore, the connection ID is used to differentiate between a client and server connection. To ensure that the tree is not empty, a root node is created when the RR tree is first established. After that, it is essential to specify the connection ID, which is necessary for determining whether the tree is a client or a server tree. Finally, the send and receive pointers are generated and pointed to the root node.

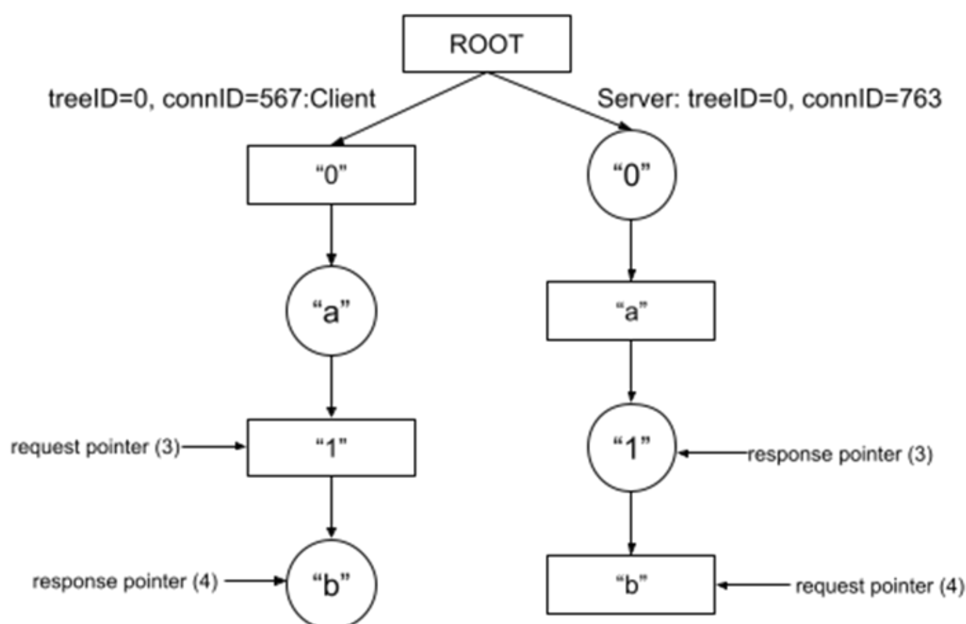

Fig. 5. RR tree illustration after the client sends “1” and the server replies “b”

4. Experiments and Results

This section presents the results of experimenting with the IPC remodeling approach on various Java networked programs. The results are compared with a previous study which utilized centralization at the model checker level [7]. The same Java networked programs used in the study by Artho *et al.,* [9] are used as the test subjects. Table 1 lists these programs and includes the name of the program, its size in terms of lines of code, and its architecture. The simplest program is Echo, which has single-threaded server and client processes, while the most complex program is Alphabet, which has multi-threaded server and client processes.

Table 1
Java networked applications used in the experiments

Application	Size (loc)	Architecture
Echo	63	Client/server
Daytime	56	Client/server
Chat	135	Client/server
Alphabet	104	Client/server

When it comes to verifying a complex application, it often involves multiple threads. The main focus is on two types of multi-threaded programs: those in which the primary thread generates all the worker threads and those in which the primary thread generates only the initial worker thread and then allows it to create additional worker threads. To manage both types of multi-threaded programs, the request and response tree can be utilized.

For each experiment, we compare the total number of states explored by the model checker, representing the state space size, the total number of bytecode instructions executed by the JPF's JVM throughout the entire model-checking process, the maximum depth of the JPF search tree in terms of the number of transitions explored by the model checker, the elapsed time which represents the total time spent, and maximum memory which shows the maximum Java heap size.

All the experiments are implemented using the computer with the configuration described as follows:

(a) Operating System: Windows 10 Pro 64-bit (10.0, Build 19044)
(b) Processor: Intel(R) Core(TM) i7-10870H CPU @ 2.20GHz (16 CPUs), ~2.2GHz
(c) Memory: 16GB
(d) Java Development Kit: 1.8.0_333
(e) Java PathFinder Core System: 8.0
(f) Eclipse IDE: 4.14.0

4.1 Experimental Settings

To differentiate between the proposed remodeling of IPC and the existing IPC, we use JPF-Nas-Hybrid (JNH) and JPF-Nas which represent the remodification and the original IPC, respectively. Both model checkers are an extension of JPF and are configured to run in multi-process mode, employing a multiprocess virtual machine and a distributed scheduler. The distributed Java applications' original code is used as input to the model checker. The applications under the centralization model checker require the server process to begin first; otherwise, an "IOException" arises. To guarantee that executions start with a specific process, the "vm.nas.initiating_target" property is set to the server process. Also, the "vm.process_finalizers" property is set to true to allow for finalizers, which are

necessary to capture certainly distributed application executions and clean up the communication channels. These two properties are used throughout the experiment.

To collect data on the decisions made during the model-checking process, we utilize a listener named "gov.nasa.jpf.listener.StateSpaceAnalyzer". This data is used to identify the primary factors that impact the size of the state space in both approaches. Additionally, we compare the scalability of the approach to the other approach based on variables such as the number of processes and the number of messages sent between processes. Finally, we provide details on the memory usage of each approach by reporting the maximum heap memory, in megabytes, that was consumed during the entire model-checking process.

Finally, for all the experiments, we only consider 20 minutes for each run, if any of the runs exceed more than 20 minutes, we report it as a state space explosion. Additionally, the existing approach can easily hit the CPU performance to 100 percent. In this case, we also report that the run reaches a state space explosion, and we will terminate the process execution.

4.2 Echo Application

The first experiment uses an application called Echo which includes a single server process and a single client process. Both processes are designed to operate using a single thread and rely on TCP sockets for communication. Once the two processes are connected, the client transmits a series of messages to the server, and the server responds by sending the same messages back to the client in the same sequence.

In this experiment, we model checks the Echo application by changing the number of threads on both the server and client. In each run, the variables "numClients" on the server code and "numThreads" on the client code are increased one by one until we reach the state space explosion.

Table 2 shows the experimental results for model checking Echo application. It shows that this approach is more effective as it leads to smaller state spaces, fewer bytecode instructions, and less complex search trees in all scenarios. The difference in state space size becomes more noticeable when there are more network interactions. This is largely due to the way we model communication channels our approach places communication buffers as the request and the response tree and keeps them in the connection manager and it does not involve read and write operations, unlike the array byte queue from the existing method.

Table 2
Execution results obtained from model checking Echo application

Thread		JPF-NAS-HYBRID			JPF-NAS		
Client	Server	Time (s)	States	Max Memory (MB)	Time (s)	States	Max Memory (MB)
1	1	0	60	243	0	66	243
2	2	0	741	243	0	1397	243
3	3	2	8767	432	8	27288	688
4	4	28	98544	432	158	483298	2416
5	5	411	1075796	782	N/A	N/A	N/A

4.3 Daytime Application

In our second experiment, we utilized the Daytime application, which consists of a server and one or more clients that communicate with each other. Both the server and client processes are single-threaded and utilize TCP sockets for communication. When a connection is established between the server and a client, the server generates a "java.util.Date" object that captures the current time and

sends its string representation to the client at that time. While a client is being served, any remaining clients are blocked until a connection with the server can be established.

In this experiment, we used our proposed model checker and JPF-Nas to perform a model check of the Daytime application. The complete source code for both the server and client can be located in Appendix A. The number of threads is incremented one by one in this scenario. If the execution time surpasses 20 minutes, we will terminate the execution.

The results of testing the Daytime application using JNH and JPF-Nas model checkers are displayed in Table 3. The table has three columns: the first one indicates the number of threads for the client and server, while the second and third columns present the time in seconds, the number of states, and the maximum memory used by the JNH and JPF-Nas model checkers, respectively. In every case, our proposed method results in a smaller state space, fewer bytecode instructions, and a shallower search graph. However, we can model check Daytime application up to 2 threads only due to the complexities of the application. If we set the number of client threads and the server threads to 3, the execution time will be longer than 20 minutes.

Table 3
Execution results obtained from model checking Daytime application

Thread		JPF-NAS-HYBRID			JPF-NAS		
Client	Server	Time (s)	States	Max Memory (MB)	Time (s)	States	Max Memory (MB)
1	1	0	62	243	0	100	243
2	2	7	9145	688	23	92598	991

4.4 Chat Application

Our next experiment involves the Chat application, which consists of a server that communicates with one or more clients using TCP sockets. The server is designed to handle multiple threads, while the clients only use a single thread. Client interactions are managed through a worker thread dedicated to handling each interaction. When a client connects to the server, the main server thread creates a worker thread to manage its communication. This allows multiple clients to be served simultaneously. Whenever a worker thread receives a message from a client, it sends it to all the other clients in the system using a shared array that stores references to all the worker threads. The worker thread then uses these references to obtain the sockets connecting the chat server to each client.

One of the input variables for Chat is the size of the array that stores references to the worker threads. The maximum number of workers that can serve clients at the same time is equal to the size of the array. To test Chat, we conducted experiments with varying numbers of client and server threads. Table 4 displays the execution times in seconds, the number of states, and the memory consumption obtained from applying both approaches.

Table 4
Execution results obtained from model checking Chat application

Thread		JPF-NAS-HYBRID			JPF-NAS		
Client	Server	Time (s)	States	Max Memory (MB)	Time (s)	States	Max Memory (MB)
1	1	0	497	243	0	251	243
2	2	61	330098	687	35	200148	1095

In this experiment, we discover that our approach leads to a larger state space and longer time. However, our proposed method still consumes less memory. If we look at the source code shown in Appendix A, each client sends the letter “H” to the server, and the server forwards the letter to all clients. Since the existing centralization processes data byte by byte in the array byte queue, the results lead to a smaller state space and lesser computation time compared to ours.

4.5 Alphabet Application

We conducted our last experiment using the Alphabet application, which involves a single server connected to one or more clients. Both the server and clients have multiple threads, and they use TCP sockets to communicate. Each client is assigned a string of digits, and their task is to get the server to convert the string into a string of letters. For example, the string "123" would become "ABC". To achieve this, each client creates multiple pairs of producer and consumer threads, with each pair responsible for transforming a portion of the client's string. The client determines the number of pairs and the size of the substrings to be transformed. Each producer and consumer pair connects independently to the server, and the server creates a worker thread to handle each connection. For each digit in the substring, the producer thread sends a byte representing the digit to the server, which then converts it to a letter and sends it to the consumer thread. The client and server can handle multiple connections simultaneously, and multiple communication channels can exist between them.

The results of experimenting with the Alphabet application using JNH and JPF-Nas model checkers are displayed in Table 5. The table has three columns: the first one indicates the number of threads for the client and server, while the second and third columns present the time in seconds, the number of states, and the maximum memory used by the JNH and JPF-Nas model checkers, respectively. In every case, the proposed method results in a smaller state space, fewer bytecode instructions, and a shallower search graph.

Table 5
Execution results obtained from model checking Alphabet application

Thread		JPF-NAS-HYBRID			JPF-NAS		
Client	Server	Time (s)	States	Max Memory (MB)	Time (s)	States	Max Memory (MB)
1	1	0	549	243	0	100	243
2	2	61	227387	687	82	301409	1691

To summarize, our experiments demonstrated the effectiveness of the proposed approach in model-checking networked applications with varying degrees of complexity. The networked programs and several matrices are compared similarly to the existing work [7,18]. Our proposed approach is found to be more efficient and better performance than JPF-NAS in terms of state space size, bytecode instructions, search tree, time, and memory complexity in most cases.

5. Conclusions and Future Work

Addressing the non-deterministic execution of threads, processes, and communication channels presents substantial hurdles in the development of dependable distributed systems. Our proposed approach introduces the request and response tree as a reliable storage mechanism for communication data. Additionally, our work processes data in multi-byte chunks, enabling efficient transmission between the client and server. We have also enhanced the RR tree to support multiple

connections, making our method scalable as the number of RR trees aligns with the number of sockets created by the server/client, with automatic incrementation. In our approach, the nodes in the tree represent the sent or received data, and during the backtracking process, we utilize request and response pointers to traverse the tree, replacing the need for traditional write and read operations. The results show the effectiveness of our work in providing a more accurate IPC model for capturing interactions between processes. Notably, our approach successfully detected a bug that went unnoticed when employing recent centralization techniques at the model checker level. Moreover, the comprehensive evaluation demonstrated that our implementation choices significantly improved the overall performance.

Future work in this area holds promising avenues for further enhancement. One potential direction is to explore the integration of advanced optimization techniques to mitigate the state space explosion and computational limitations associated with backtracking in existing IPC models. By addressing this avenue, we can pave the way for even more robust and efficient model-checking techniques for distributed systems, enabling the development of highly reliable and secure software systems.

Acknowledgment

The authors would like to thank the Prototype Development Research Grant Scheme (PRGS) (203.PKOMP.6740069) by the Ministry of Higher Education (MOHE) for funding the research project.

References

[1] Vitillo, Roberto. *Understanding Distributed Systems: What every developer should know about large distributed applications*. Roberto Vitillo, 2022.

[2] Chan, Charisma, and Beth Cooper. "Debugging Incidents in Google's Distributed Systems: How experts debug production issues in complex distributed systems." *Queue* 18, no. 2 (2020): 47-66. https://doi.org/10.1145/3400899.3404974

[3] Muscholl, Anca. "Automated Synthesis: a Distributed Viewpoint." In *37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[4] Clarke, Edmund M., Thomas A. Henzinger, and Helmut Veith. "Introduction to model checking." *Handbook of Model Checking* (2018): 1-26. https://doi.org/10.1007/978-3-319-10575-8_1

[5] Baier, Christel, and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[6] Beyer, Dirk, and Andreas Podelski. "Software model checking: 20 years and beyond." In *Principles of Systems Design: Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday*, pp. 554-582. Cham: Springer Nature Switzerland, 2022. https://doi.org/10.1007/978-3-031-22337-2_27

[7] Shafiei, Nastaran, and Peter Mehlitz. "Extending JPF to verify distributed systems." *ACM SIGSOFT Software Engineering Notes* 39, no. 1 (2014): 1-5. https://doi.org/10.1145/2557833.2560577

[8] Sherman, Elena, Yannic Noller, Cyrille Artho, Franck van Breugel, Anto Nanah Ji, John Kellerman, Parssa Khazra et al. "The Java Pathfinder Workshop 2022." *ACM SIGSOFT Software Engineering Notes* 48, no. 1 (2023): 19-21. https://doi.org/10.1145/3573074.3573080

[9] Artho, Cyrille, and Willem Visser. "Java Pathfinder at SV-COMP 2019 (competition contribution)." In *Tools and Algorithms for the Construction and Analysis of Systems: 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019*, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III 25, pp. 224-228. Springer International Publishing, 2019. https://doi.org/10.1007/978-3-030-17502-3_18

[10] Artho, Cyrille, Watcharin Leungwattanakit, Masami Hagiya, and Yoshinori Tanabe. "Efficient model checking of networked applications." In *Objects, Components, Models and Patterns: 46th International Conference, TOOLS EUROPE 2008*, Zurich, Switzerland, June 30-July 4, 2008. Proceedings 46, pp. 22-40. Springer Berlin Heidelberg, 2008. https://doi.org/10.1007/978-3-540-69824-1_3

[11] Artho, Cyrille, Watcharin Leungwattanakit, Masami Hagiya, Yoshinori Tanabe, and Mitsuharu Yamamoto. "Cache-based model checking of networked applications: From linear to branching time." In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pp. 447-458. IEEE, 2009. https://doi.org/10.1109/ASE.2009.43

[12] Leungwattanakit, Watcharin, Cyrille Artho, Masami Hagiya, Yoshinori Tanabe, and Mitsuharu Yamamoto. "Model checking distributed systems by combining caching and process checkpointing." In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pp. 103-112. IEEE, 2011. https://doi.org/10.1109/ASE.2011.6100043

[13] Stoller, Scott D., and Yanhong A. Liu. "Transformations for model checking distributed Java programs." In *International SPIN Workshop on Model Checking of Software*, pp. 192-199. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. https://doi.org/10.1007/3-540-45139-0_12

[14] Artho, Cyrille, and Pierre-Loic Garoche. "Accurate centralization for applying model checking on networked applications." In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pp. 177-188. IEEE, 2006. https://doi.org/10.1109/ASE.2006.10

[15] Ma, Lei, Cyrille Artho, and Hiroyuki Sato. "Analyzing distributed Java applications by automatic centralization." In *2013 IEEE 37th Annual Computer Software and Applications Conference Workshops*, pp. 691-696. IEEE, 2013. https://doi.org/10.1109/COMPSACW.2013.137

[16] Barlas, Elliot, and Tevfik Bultan. "NetStub: A framework for verification of distributed Java applications." In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 24-33. 2007. https://doi.org/10.1145/1321631.1321638

[17] Nakagawa, Yoshihito, Richard Potter, Mitsuharu Yamamoto, Masami Hagiya, and Kazuhiko Kato. "Model checking of multi-process applications using SBUML and GDB." In *Proc. Workshop on Dependable Software: Tools and Methods*, pp. 215-220. 2005.

[18] Leungwattanakit, Watcharin, Cyrille Artho, Masami Hagiya, Yoshinori Tanabe, Mitsuharu Yamamoto, and Koichi Takahashi. "Modular software model checking for distributed systems." *IEEE Transactions on Software Engineering* 40, no. 5 (2013): 483-501. https://doi.org/10.1109/TSE.2013.49