A Novel Approach on Centralizing Networked Applications

Cyrille Artho, National Institute of Informatics, Tokyo, Japan*

October 28, 2005

Abstract

Software model checkers can directly be applied to single-process programs, which typically are multi-threaded. However, multi-process applications cannot be model checked so easily. One approach is the manual merging of several processes into one, removing inter-process communication which is usually network-based. This process is very labor-intensive and a major obstacle towards model checking of client-server applications.

Previous work has addressed the merging of multiple applications but mostly omitted network communication. Remote procedure calls were simply inlined, creating similar results for simple cases but also removing much of the inherent complexities involved. Our approach is a fully transparent replacement of network communication, resulting in a program that is much closer to the original. This makes our approach suitable for testing, debugging, and software model checking.

1 Introduction

Model checking [2] tries to explore the entire behavior of a system under test (SUT) by investigating each reachable system state. Recently, model checking has been applied directly to software, sometimes even on concrete systems. Java [3] is a popular object-oriented, multi-threaded programming language. Verification of Java programs has become increasingly important. Several model checkers for Java-based programs have been created, e.g. [1, 6]. Existing software model checkers can only explore a single process and are not applicable to networked applications, where several processes interact. Most non-trivial programs which are in use today use network communication. Such programs are typically part of a larger system where several processes interact.

Process centralization is a possible solution*:* Processes are converted into *threads* and merged into a single application [5]. Networked applications can then run as one multi-threaded application. This approach is applicable if all programs to be merged are available in the same format and inter-program communication can be modeled accurately. The latter has not been addressed satisfactorily yet. Previous work inlined parts of one program in another one, modelling certain patterns of interaction using Remote Method Invocation (RMI) under Java [5]. Unfortunately, this implementation is very specific to certain RMI-based programs and cannot cover more general, TCP/IP-based communication. Our approach can fully replace the widely used socket mechanism. It is therefore more generic and closer to the original program than the previous inlining approach.

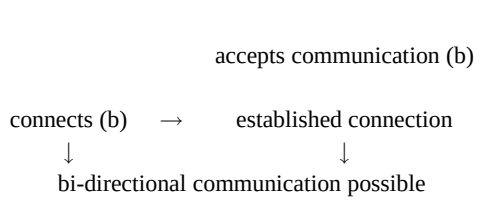

Figure 1: Client-server communication.

2 Network communication, RMI

Network communication can be modeled as an interaction of two peers, a *client* and a *server.* The server allows for incoming communication by accepting connections to a certain port. The client can subsequently connect to that port. After a connection is established, a bidirectional communication channel exists between the client and the server. Communication can then be performed in an asynchronous manner: Underlying transport mechanisms (commonly TCP/IP) ensure that sent messages arrive eventually (if a connection is available), but with some delay. This applies to messages in both directions. A connection can be closed by the client or the server, terminating communication.

Figure 1 illustrates this: The client has to wait until the server is available, and retry a connection attempt if necessary. When the server accepts a new connection, its execution blocks until a client has connected. A blocked state is shown as (b) in the figure. As soon as a client connects, the server is unblocked, and a connection is established by underlying system libraries. The corresponding connect call is in turn blocking for the client, unblocking as soon as the response from the server is received.

Remote Method Invocation (RMI) in Java builds in sockets and behaves in a similar way, as shown by Figure 2. RMI offers a naming mechanism which allows clients to find a server. After the lookup is performed,

*Funding: Special Coordination Funds for Promoting Science and Technology.

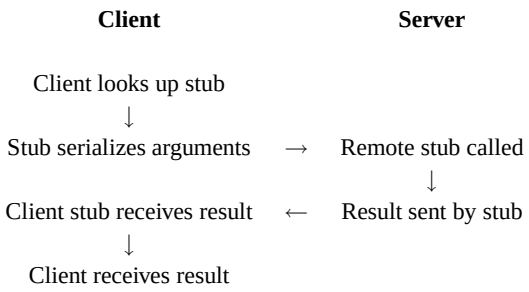

Figure 2: Remote Method Invocation.

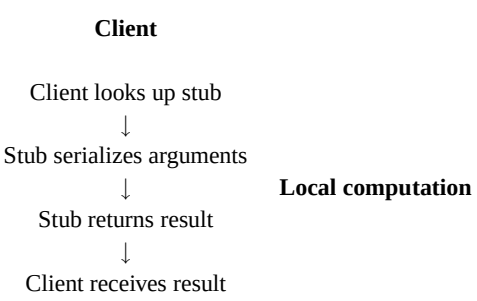

Figure 3: Inlined RMI call.

the client communicates with the server via *stubs*, which hide network communications and make (blocking) RMI calls appear to be local. Arguments for calls have to be serialized (converted into raw data) before transmission. The client blocks until it receives the response from the server. The stub receives the serialized result, unblocking execution, and relays the result to the calling client.

Previous work [5] has inlined the remote computation of the server, replacing a blocking remote call with a local call, as shown by Figure 3. This idea "cannibalizes" on the blocked client thread for simulating remote computation. In doing so, it greatly simplifies the original program, which is desirable for model checking. However, it also removes a lot of the structural complexity, which may be harmful because it may mask problems.

3 Our solution

We kept the idea of replacing multiple processes with multiple threads in a single process. The client and server applications are still merged into a single application as done before [5], but communication is treated in a totally different way. Due to the complex nature of true bidirectional communication in client-server programs, a simple inlining mechanism is no longer applicable.

General network communication was broken down into two steps: Connection establishment, and bidirectional communication. We used a two-step barrier [4] to model the blocking connection mechanism shown in Figure 1. In a first step, the server blocks during the `accept` call. When the client calls `connect`, the server is unblocked while the client blocks and waits for completion of the connection. This ensures that the sequence of each original application passing through blocking library calls is preserved in the centralized version. Upon connection, two unidirectional inter-thread pipes are set up, as available through `java.io.PipedInputStream` and `java.io.PipedOutputStream`. They model the underlying network communication normally provided by system libraries, replacing inter-process communication by inter-thread communication.

Due to the complex inter-thread interactions, development of a correct socket replacement was non-trivial. We used the JavaPathFinder model checker [6] to verify correctness of our implementation, which currently supports one connection per port but could be extended to multiple connections. We have also successfully used this socket replacement for running and model checking centralized example applications, such as an echo server.

4 Summary

Distributed programs include several processes and typically use network communication. For model checking, processes have to be replaced by threads and merged into a centralized application. Replacement of communication mechanisms is also necessary for non-trivial applications. While Remote Method Invocation can be replaced by inlining, this replacement is not very accurate and cannot be extended to arbitrary socket-based communication.

Our approach replaces the socket connection mechanism by barriers and the communication channels by two unidirectional, inter-thread pipes, as available by the Java library. We have successfully model checked our implementation using example client-server applications.

References

[1] C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient Dynamic Analysis for Java. In *Proc. CAV '04*, Boston, USA, 2004. Springer.

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

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

[4] D. Lea. *Concurrent Programming in Java, Second Edition*. Addison-Wesley, 1999.

[5] S. Stoller. Model-Checking Multi-threaded Distributed Java Programs. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 224–244. Springer, 2000.

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