Design of a Code Generator for ActiveRMI*

Meng-Chun Wueng, Fu-Fang Yang and Cheng-Zen Yang
Department of Computer Science and Engineering
Yuan Ze University
Chungli, Taiwan, R.O.C.
E-mail: {jun,jally,czyang}@syslab.cse.yzu.edu.tw

Abstract— **Java RMI provides a convenient way for programmers to design distributed applications in traditional networks. However, this paradigm suffers from a centralize bottleneck problem when many clients simultaneously send a large number of requests to the server. ActiveRMI improves the Java RMI performance by employing active networks technology with a code caching scheme. With ActiveRMI, the server bottleneck problem can be relieved, and the amount of network traffic between clients and RMI servers can be reduced. Consequently, the user response time is also highly reduced. However, the early version of ActiveRMI lacks programming tools to help programmers write code. In this paper, we report a code generator design called ARMIGen. Programmers can use ARMIGen to conveniently develop ActiveRMI applications and deploy them on active networks. We have evaluated the generation time and the execution time of the generated ActiveRMI applications. The preliminary experimental results show that the performance of the generated code is comparable to the performance of the hand-written code. The experiments also show that ARMIGen is an effective and efficient programming tool.**

Keywords— **ActiveRMI, Code Generation, Java RMI, ANTS, Active Networks.**

I. Introduction

Java RMI (Remote Method Invocation) provides a convenient way for programmers to design distributed applications because RMI encapsulates the transmission details. In RMI, the client stub and the server skeleton provide programming interfaces to the client and the RMI server. Therefore, the programmers need not to handle packets sending/receiving in detail. However, this programming paradigm suffers from a centralized bottleneck problem when many clients simultaneously send a large number of requests to the server. The server will deny or postpone further requests due to its heavy workload. Furthermore, the traffic in the path between the clients and the server is intensively congested. Therefore, not only the response time is lengthy, but also the network bandwidth along the path is completely consumed. In addition to the scalability problem, the traditional RMI paradigm also suffers from the single point failure problem when the server crashes or the network is partitioned.

To alleviate these RMI shortcomings, as discussed in [1], active networks [2, 3] provide a new network infrastructure that can improve the scalability, performance, and fault-tolerance of RMI. Since the routers (active routers) in active networks can execute user-customized code, if these intermediate active routers act as the agents of the remote RMI services and serve the RMI invocations immediately, the response time of RMI applications can be highly shortened. Furthermore, these active routers can provide services when the RMI server crashes. In [4, 5], Wueng and Yang have proposed a novel Java RMI mechanism called ActiveRMI that utilizes the programmability of active networks to speed up the invocation of remote RMI services.

ActiveRMI implements the RMI protocol on active networks. When an RMI request is issued, the intermediate active router nearest to the client intercepts it and checks the local code cache. If the service code has been already cached, the active router executes the service code immediately to satisfy the request. In [4], the preliminary experimental results show that ActiveRMI improves the response time up to 36% in the game-of-life benchmark [6].

ActiveRMI achieves three improvements. First, the user response time are reduced in ActiveRMI compared with the traditional Java RMI. Second, the server workload is shared with the active routers. The centralized bottleneck problem is alleviated. Third, the amount of network traffic between the clients and the RMI server is reduced because the active routers act as the agents to intercept client requests and satisfy remote services immediately.

Since the early version of ActiveRMI lacks enough programming tools to automate code translation, the design of a code generator is important for the ActiveRMI project. The ActiveRMI code generator is called ARMIGen (ActiveRMI Generator) [7]. We intend to achieve two goals in the ARMIGen design. First, the interface and usage of ARMIGen should be simple for programmers. Therefore, it can help programmers to develop ActiveRMI code very effectively and efficiently. They need not hand-code the details of ActiveRMI network operations. Second, the performance of the generated ActiveRMI code should be comparable to the hand-coded version. Otherwise, if the generated code sacrifices the system performance seriously, application development will be hindered. To achieve these goals, we introduce code templates to reduce the number of functions in which the programmer should write code. We also try to optimize these templates, so the generated code has acceptable performance.

The rest of the paper is organized as follows. Section 2 introduces the related work including ANTS and the recent progress in ActiveRMI. Section 3 describes the ARMIGen architecture design and its functionalities. Section 4 presents the results of two experiments to justify the correctness, the code generation performance, and the system execution performance. Finally, Section 5 concludes the paper.

*This research was supported by National Science Council of R.O.C. under grant NSC 92-2213-E-155-036.

II. ANTS AND ACTIVERMI

In ActiveRMI, each active router has a code cache to store remote RMI service code. The client stub is responsible for handling the underlying packet transmissions on the client side. Since ActiveRMI is developed on Janos [8, 9] and ANTS [10, 11], it implements an extended ANTS protocol to practice code caching. However, in the early development ActiveRMI has not provided enough programming tools to assist programmers in code writing.

ANTS is designed at MIT to provide an active network platform [10, 11] with two features. First, ANTS supports customized code execution to provide various network services. Second, ANTS supports new protocols establishment and deployment. In ANTS, capsules carry user code and data to achieve the features. ANTS uses the capsule hierarchy to deploy new network protocols.

In ANTS, a hierarchical model of capsules is specified to support various application requirements. A capsule is a basic unit carrying user code and data and includes well-known routines for common processing at each active node. A code group is a collection of correlative capsule types, and a protocol is a collection of correlative code groups. ANTS uses the capsules to deploy new protocols dynamically.

When a capsule arrives at an active router, the active router checks the protocol of this capsule. If the router does not have the protocol code, it sends a code-downloading request back to the previous active router. The capsules are then kept provisional until all the required code is prepared. If all required code segments are prepared, the capsules are awakened and ANTS continues the execution. If the previous active router does not response the code, the capsules will be discarded.

In ANTS, capsules and protocols are defined in Java files. An ANTS application composes of the following files: a main program, the definition of capsules and protocols, and the script files. The script files contain a configuration and a routing table, and a start file that executed by users. The configuration records a physical IP address, a logic IP address, and a network port. The logic IP address represents a specified domain of each ANTS application. The Janos virtual machine receives capsules from a fixed network port and dispatches these capsules to different domain according to the logic IP. The routing table records the physical IP addresses, the logic IP addresses, and the network ports of active routers. The active routers behave as tunnels because not all routers in current IP networks are active routers. Programmers should make sure that the capsules passing though specified active routers and they have the information of the active routers in the routing table. With the routing table, the capsules of the application pass though the specified active routers and program code can be loaded to the active routers with these capsules.

ActiveRMI [4, 5] is designed to utilize the benefits provided for active networks for Java RMI. There are RMI code caches on intermediate active routers to cache RMI service code. With the RMI code caches, the users repose time is reduced and the scalability is enhanced because RMI services cached on the intermediate active routers can satisfy the request as soon. In the ActiveRMI code caching mechanism, a manager called ARMIMan maintains the RMI service code execution and the cache.

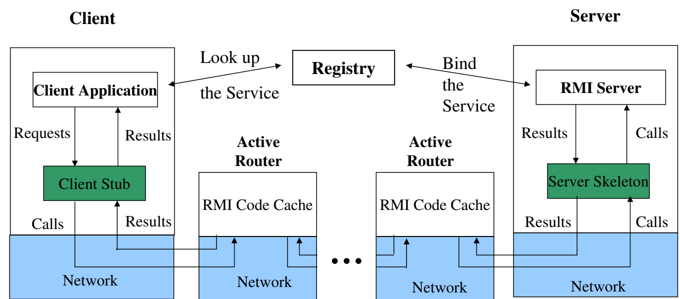

Fig. 1. The ActiveRMI infrastructure.

ActiveRMI applications use several capsules to cooperate with ARMIMan and the registry: `lookup` capsules, `registry` capsules, `argument` capsules, and `result` capsules. Figure 1 illustrates the ActiveRMI infrastructure. An RMI server uses `registry` capsules to bind RMI service information to the registry. The client sends `lookup` capsules to the registry to search for the RMI services. The registry processes the `lookup` capsules and then replies to the client with the registration information and the corresponding stub. The client can then invoke the stub to send `argument` capsules to the RMI server. When ARMIMan detects the `argument` capsules, it intercepts these capsules and provides RMI service code from the local RMI code caches. The stub then receives `result` capsules and replies the result to the client applications.

ARMIGen generates code for these capsules and summarizes their definitions in protocol files. Furthermore, ARMIGen inserts ActiveRMI function code to the program files of the client and the server and produces corresponding scripts files. To perform the security mechanism in ActiveRMI, ARMIGen processes corresponding actions to achieve ActiveRMI protocols. The code translation procedure of ARMIGen will be detailed further in Section 3.

III. CODE GENERATION DESIGN

The major functionality of ARMIGen is to automatically translate Java RMI code to ActiveRMI code. ARMIGen reads the code templates and the specifications, and generates the complete ActiveRMI application. This section provides the detailed design insights in ARMIGen. Several design issues are also discussed in this section.

A. The ARMIGen Architecture

The ARMIGen architecture is depicted in Figure 2. Programmers provide interface files and Java program code to ARMIGen. ARMIGen translates these files to produce ActiveRMI executable binary files. There are three components in ARMIGen: the parser, the code translator, and the back-end processor. The parser analyzes the interface files and Java program code, and check the syntax description to produce the meta-information for ActiveRMI code generation. The code translator reads the meta-information and then performs the

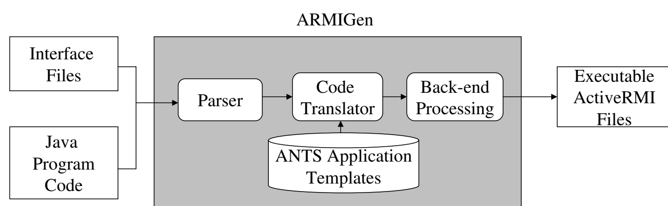

Fig. 2. The ARMIGen architecture.

generation. The backend processor then reads the ActiveRMI source code, compiles it, makes MD5 authentication, and packs the binary byte code into the Java Archive (jar) files.

B. The Parser

The parser has two main tasks: checking the syntax of the input files and summarizing the meta-information from the input files. The parser checks the syntax of the Java program code to ensure the validity of the generated ActiveRMI code. It processes the Java program code, the configuration, and the interface file to get the information of the remote service, the arguments, and the type of the return results. In ActiveRMI, the types of the arguments and the results are required to be primitive types, because the underlying ANTS platform marshals and unmarshals only the variables of primitive types. The configuration files are also processed in advance for setting up the environment of the ActiveRMI applications. Finally, the code translator generates the ActiveRMI source code with the configuration information.

C. The Code Translator

The code translator reads the meta-information and the ANTS application templates, and generates the ActiveRMI source files of scripts and Java code. It generates different files for clients and servers respectively. For client-side code generation, the code translator produces a configuration, a start file, a routing table, a main program, a lookup capsule file, and a lookup protocol file. For server-side code generation, it generates scripts and Java source code. The back-end processor then translates the Java source code to the executable binary file by invoking `antsjavac`.

D. The Back-end Processor

The back-end processor performs three main steps for final binary code generation: compiling the translated ActiveRMI Java source code, adding the MD5 authentication code, and archiving the byte code to the jar file. These steps are detailed as follows.

Since ActiveRMI applications use the ANTS libraries, the ActiveRMI source code must be compiled with the ANTS compiler called `antsjavac`. In this phase, ARMIGen invokes `antsjavac` to compile ActiveRMI source code files. After the compilation, ARMIGen generates the MD5 authentication code for remote service code. Because the remote services are executed at the intermediate active routers in ActiveRMI, a security mechanism is necessary to prevent malicious attacks. Currently, ActiveRMI uses MD5 to authenticate the remote service code. Therefore, ARMIGen

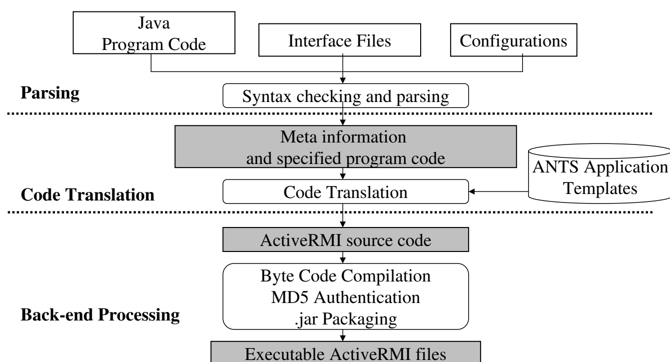

Fig. 3. Processing flow in ARMIGen translation.

```
ActiveRMI Path:/home/jally/ANTS/ants3;
Registry Physical IP:10.0.3.2;
Registry Logic IP:70.0.0.99;
Registry Port:7099;
ARCache Physical IP:10.0.2.2;
ARCache Logic IP:70.0.0.50;
ARCache Port:7050;
Application Physical IP:10.0.2.5;
Application Logic IP:70.0.0.10;
Application Port:7010;
```

Fig. 4. An example of the ARMIGen configuration file.

encodes every byte code file with an MD5 authentication signature. In the ActiveRMI implementation, jar files are used in service registration, code cache maintenance, and ActiveRMI invocation. The ActiveRMI registry will keep the jar file of the client stub code in its repository. When a client invokes the service code, the registry will send the jar file to the client. The code cache also maintains the jar files for building up execution environments without registry binding. The complete service code is also archived in a jar file for ActiveRMI execution.

E. Translation Process

The translation process in ARMIGen is depicted in Figure 3. To develop an ActiveRMI application, programmers should prepare three main files: the ActiveRMI source function code, the interface files, and the application environment configuration. Since ActiveRMI heavily relies on ANTS, two necessary ANTS methods, `run()` and `receive()`, are required in ActiveRMI programming. When coding the `run()` method, the programmer defines the initial operations at run time. The `receive()` method defines the operations when the ANTS application receives capsules. These two methods are also the only part of methods which the programmer needs to code.

An ARMIGen configuration records the settings of the ActiveRMI environment. To execute the ActiveRMI applications on an ANTS platform, the programmer needs to specify the logical IP addresses, the corresponding physical IP addresses, and the port numbers. Figure 4 is an example of the ARMIGen configuration. In this example, the registry has a physical IP 10.0.3.2, a logical IP 70.0.0.99, and the port number 7099. The first line is to specify the path of the application. Based on

this configuration, ANTS knows where to locate the services and where the active router resides. As shown in Figure 4, each configuration has four parts: the installation path of ActiveRMI, the physical IP address, the logical IP address, and the network port.

The correctness of the configuration is left for programmers. If the programmers fill up the settings of the configuration incorrectly, even ARMIGen can successfully generate the executable ActiveRMI files. However, the ActiveRMI application cannot find the RMI service. If the IP address and port numbers of the remote ActiveRMI registry and the intermediate active routers are not correctly set, ARMIGen will not be aware of the incorrectness.

The meta-information composes the arguments of the applications and the setting of the ActiveRMI environment. The arguments of applications include the remote service name, the types of the method arguments, and the types of the return results. The code translator merges the meta-information into the ANTS application templates, and then generates the ActiveRMI application source code. There are two kinds of ANTS application templates: the ANTS application code templates, and the script templates. The code translator generates the ActiveRMI source code files for client and server code separately. Their capsule file specifications and corresponding protocol files are different. In client code, the capsule specification defines the `lookup` capsule to search remote service from the registry and the `lookup` protocol to perform the search. In server code, the capsule specification defines the `registry` capsule to register the binding information.

After the code translation, ARMIGen generates the complete ActiveRMI source code and then compiles it to generate the executable byte code files. As for authentication ARMIGen generates MD5 authentication code for these byte code files in a specified file. Finally, ARMIGen packages the byte code files and the MD5 authentication file, and generates three executable ActiveRMI jar files for the registry, the RMI code caches, and the ActiveRMI protocol.

IV. Performance Evaluation

In this section, we report our evaluation of ARMIGen with the code generation time and the user response time. We also compare the user response time between the generated code and hand-written ActiveRMI code to show the comparable execution performance achieved by ARMIGen.

ARMIGen is developed with JDK 1.3.1. ActiveRMI is developed on Janos that includes ANTS 2.0.2, Java NodeOS 1.0.2, and Janos VM 0.0.8. The underlying OS is FreeBSD 4.6.

Generally, there are many intermediate routers between the client and the server. In our experiments, we built up a primitive environment with three computers: a server, an active router, and a client. In the first experiment, we evaluated the code generation time. The network topology of the second experiment was a isolated network environment containing the server, the client, and the active router. There was no legacy router in our experimental environments.

Four applications were used as the benchmarks in our experiments: game of life [6], matrix multiplication, knight's

	Client Code				**Server Code**			
	Parsing and file generation	Compilation	Packaging	Total Time	Parsing and file generation	Compilation	Packaging	Total Time
Game of Life	2732.8	3003.5	2468.0	8204.3	3158.7	6903.5	4449.2	14511.3
Matrix Multiplication	2695.0	2606.4	2444.9	7746.1	2931.4	5568.8	4258.8	12759.0
Knight's Tour Problem	2646.1	2989.0	2633.6	8168.6	3850.4	7457.3	4568.7	15876.3
Color-depth Changing	2686.4	2994.1	2508.0	8188.5	3122.7	6882.1	4378.9	14383.7

Fig. 5. The average code generation time in milliseconds. We compiled the code for 20 times.

	Cold Cache Miss			**Cache Hit**		
	Generated Code	**Hand-written Code**	**Overhead Ratio**	**Generated Code**	**Hand-written Code**	**Overhead Ratio**
Game of Life	8798.2	8790.4	0.09%	4697.6	4691.8	0.12%
Matrix Multiplication	8828.1	8752.7	0.86%	4695.8	4676.3	0.42%
Knight's Tour Problem	53955.9	53336.8	1.16%	49407.8	49306.2	0.21%
Color-depth Changing	8820.6	8736.4	0.96%	4696.1	4681.7	0.31%

Fig. 6. The average user response time of the generated code and hand-written code in milliseconds. Each benchmark application was executed 20 times.

tour [12], and color-depth changing. The game-of-life program is played on a grid of square cells. Each cell can be live or dead. The matrix multiplication is a 4×4 multiplication. The knight's tour program shows the steps of a knight chess that walks by an "L" rule in a 5×5 board. These three remote services consume large computing power. Finally, the color-depth changing application converts a color BMP picture to a 256-grey BMP picture.

The code generation time was measured at an AMD K6-2 400MHz PC with 392MB RAM. The generation time is composed of three parts: the parsing time, the byte code compilation time, and the back-end processing time. Figure 5 shows the detailed results for client code generation and server code generation.

The total code generation time of the client code is approximately 39% less than the server code generation, because the server code needs to be packaged to three jar files for service registration, code cache maintenance, and ActiveRMI invocation. However, the client code is packaged to only a jar file. Furthermore, ARMIGen needs to insert the registry code to the remote service code and complie it for ActiveRMI execution. Therefore, the total code generation time of server code is longer then the generation time of client code.

To verify the performance of generated code, we built up a small environment to measure the user response time. The environment contained three PCs as the client, the active router and the server. The client and the active router were two Pentium II PCs running at 300MHz with 256MB RAM. The server was an AMD K6-2 PC running at 400MHz with 392MB RAM. The network is 100M bps Ethernet. The average performance results of the generated code and the

hand-written code are depicted in Figure 6.

In this experiment, the maximum overhead ratio of the generated code is 1.16%. This is because that ARMIGen generates extra variables to record the remote method names, the arguments of the method, and the result. These variables will be additionally processed in the generated code. Overall, we can find that the performance of the generated code is comparable to the performance of the hand-written code.

V. Conclusions and Future Work

Socket programming is a general mechanism for programmers to develop distributed applications. However, it is complicated and error-prone for programmers because they need to hand-code socket functions. Java RMI provides a convenient way for programmers to design distributed applications because they do not need to handle underlying network transmission details. However, this paradigm suffers when many clients simultaneously send a great number of requests to a server. The server workload will be highly burdened and the traffic between the clients and the server is intensively congested. To these RMI shortcomings, active networks provide a new network infrastructure that can be used to improve the scalability and performance of RMI.

ActiveRMI improves the traditional Java RMI by exploiting active networks technology. With RMI code caches, the response time of ActiveRMI applications is reduced and server workload is distributed to the intermediate active routers. Furthermore, the amount of the network traffic between the active routers and the RMI server is also reduced. To facilitate ActiveRMI application development, code generation tools are crucial.

In this paper, we report the design of a code generator called ARMIGen. Programmers can use ARMIGen to conveniently develop ActiveRMI applications. They only need to design Java program code, the interface files, and the configuration. ARMIGen automatically generates the ActiveRMI source code and the executable ActiveRMI files. The preliminary experimental results show that the performance of the generated code is comparable to the performance of the hand-written code.

There are still some issues left for further discussion. For example, error handling and exception handling are not completely discussed in the present ActiveRMI development. Another issue is debugging. Since there are underlying RMI code caches, debugging ActiveRMI code becomes complicated. This will be in our future implementation plan.

References

[1] C.-Z. Yang and C.-W. Chen, "Issues on the RPC Paradigms in Active Networks," in *Proceedings of 2002 Active Networking Workshop*, (Chungli, Taiwan), pp. 206–213, Yuan Ze University, Sept. 2002.

[2] D. Tennenhouse, J. Smith, W. D. Sincoskie, D. Wetherall, and G. Minden, "A Survey of Active Network Research," *IEEE Communications Magazine*, vol. 35, pp. 80–86, Jan. 1997.

[3] D. Tennenhouse and D. Wetherall, "Towards an Active Network Architecture," *Computer Communication Review*, vol. 26, no. 2, pp. 5–18, 1996. Also in Proceedings of Multimedia Computing and Networking (MMCN 96).

[4] M.-C. Wueng and C.-Z. Yang, "Design and Implementation of a Java RMI Caching Mechanism on Active Networks," in *Proceedings of the 6th International Conference on Advanced Communication Technology (ICACT 2004)*, (Phoenix Park, Korea), pp. 561–566, Feb. 2004.

[5] M.-C. Wueng and C.-Z. Yang, "Design of Consistency Maintenance in ActiveRMI Code Caching," in *Proceedings of 2003 Active Networking Workshop*, (Chungli, Taiwan), pp. 17–22, Sept. 2003.

[6] P. Callahan, "What is the Game of Life?." http://www.math.com/students/wonders/life/life.html.

[7] F.-F. Yang, "Design of a Code Generator for Java Remote Method Invocation on Active Networks," Master's thesis, Yuan Ze University, Chungli, Taiwan, ROC, July 2004.

[8] L. Peterson, Y. Gottlieb, M. Hibler, P. Tullmann, J. Lepreau, S. Schwab, H. Dandekar, A. Purtell, and J. Hartman, "An OS Interface for Active Routers," *IEEE Journal on Selected Areas of Communication*, vol. 19, pp. 473–487, Mar. 2001.

[9] P. Tullmann, M. Hibler, and J. Lepreau, "Janos: A Java-Oriented OS for Active Network Nodes," *IEEE Journal on Selected Areas in Communications*, vol. 19, pp. 501–510, Mar. 2001.

[10] D. Wetherall, *Service Introduction in an Active Network*. PhD thesis, Massachusetts Institute of Technology, Feb. 1999.

[11] D. Wetherall, J. Guttag, and D. Tennenhouse, "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols," in *Proceedings of First IEEE Conference on Open Architectures and Network Programming (OPENARCH'98)*, pp. 117–129, Apr. 1998.

[12] D. L. Katz, B. Carnahan, E. I. Organick, and S. D. Navarro, "Education and Training: Computers in Engineering Education 1960-1964," in *Proceedings of the 1962 ACM National Conference on Digest of Technical Papers*, pp. 22–23, 1992.