Cache Consistency Control in ActiveRMI*

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

Abstract

In the traditional end-to-end network model, the intermediate routers are only responsible for storing and forwarding the packets from one endpoint to another endpoint. In such a model, the Remote Procedure Call (RPC) methodology was proposed to provide a concise and transparent programming interface. As the following development, an RPC-like mechanism RMI (Remote Method Invocation) has been incorporated in Java. However, in a large-scale environment the RPC/RMI paradigm faces many limitations in system scalability. In our previous research, we have proposed a new programming paradigm called ActiveRMI (Active Remote Method Invocation) to improve the scalability problem of the traditional RMI mechanism by using active network technology. ActiveRMI provides a hierarchical code-caching scheme in which remote ActiveRMI code is cached and built up at proximate active routers to process RMI requests. Therefore, the server load is shared and the amount of network traffic is reduced. However, consistency problem arises if multiple copies of ActiveRMI code exist at different active routers. In this paper, our approaches for cache consistency control in ActiveRMI are elaborated. More comprehensive experiments for different distributed applications are on our schedule to understand the potential bottleneck problems in the current design.

Keywords: ActiveRMI, Java RMI, Active networks, Code caching technology, Consistency protocol.

1. Introduction

In traditional network architecture, the intermediate routers are only responsible for storing and forwarding the packets from one endpoint to another endpoint. This presents an end-to-end network model in which the routers perform computations only for packet transmission. In such a scenario, the client-server computation model is widely employed in distributed application design and socket programming is the most common programming methodology. Although applications designed in socket programming are efficient, application development is tedious because programmers need to control every transmission detail. Therefore, in 1984 the Remote Procedure Call (RPC) methodology was proposed to provide a concise and transparent programming interface [1].

However, in a large-scale environment the RPC paradigm faces many limitations in system scalability. When a large number of client requests burst to an RPC server, the RPC service performance will be degraded due to two following situations. One is that the workload of the server is highly increased, and the server thus becomes the system bottleneck. Another is that the amount of network traffic to the server is also highly boosted. Therefore, not only the clients wait for the responses for a long time, but also the performance of all network services of the same area is influenced. These two situations all hurt the system scalability. In addition to the scalability problem, RPC services are fragile when server or network failures occur. Although introducing a multi-tier design can relieve these problematic situations, application development is thus complicated. The clients need to be aware of these explicitly added middle tiers. As discussed in [2], active networks provide a new network infrastructure that can be used to leverage the RMI performance improvements.

In our previous research [4], an active-network-based RMI mechanism called ActiveRMI (Active Remote Method Invocation) has been proposed. As depicted in Figure 1, RMI code can be cached at intermediate active routers in ActiveRMI so that RMI requests can be intercepted and served at the proximate active routers. Three advantages are thus achieved. First, the server bottleneck problem is alleviated because the server workload is shared with intermediate active routers. Second, the amount of the network traffic to the RMI server is reduced since the packet transmission is localized between the clients and the nearby active routers. Third, the user response time is thus shortened. Therefore, the scalability and RMI performance in ActiveRMI is further improved.

Although caching ActiveRMI code at proximate active routers improves the system scalability and RMI performance, the extra overhead for maintaining the consistency of the cached ActiveRMI code may not be ignorable and thus degrade ActiveRMI performance instead. This overhead is incurred because the code or data of the cached ActiveRMI code may be inconsistent

* This research was supported in part by National Science Council under grant NSC91-2745-P-155-003.

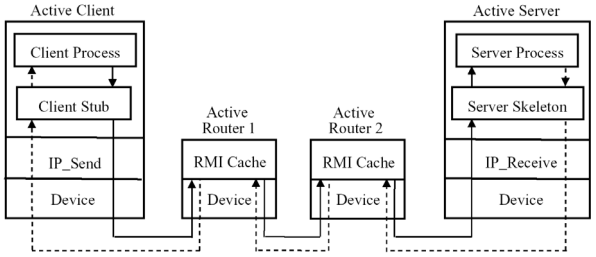

Figure 1: Architecture overview of ActiveRMI.

when the ActiveRMI code is modified at the remote RMI server without updating the cached code. Besides, clients may modify the persistent data at active routers without updating the original copy at the RMI server. Data inconsistency is thus introduced. Both situations show that consistency control is not a trivial issue in the design of ActiveRMI.

In this paper, four approaches incorporated in ActiveRMI consistency control are elaborated. They include callback promise, delayed-write, session guarantee, and write-invalidate. Furthermore, because the code can be cached at intermediate active routers, ActiveRMI employs a hierarchical distributed caching scheme to avoid unnecessary long-distance capsule transmission. If an ActiveRMI service is frequently invoked, its code is cached at multiple intermediate active routers along the network path to the RMI server. The cached ActiveRMI services and their valid data can be thus shared with other proximal clients for the future requests.

The remainder of the paper is organized as follows. Section 2 briefly reviews previous studies about the consistency maintenance of object caching. Section 3 provides a brief background description of ActiveRMI. The execution of a cached ActiveRMI application is mainly explained. Then the design of the code and data consistency maintenance in ActiveRMI is separately elaborated in Section 4 and Section 5. Finally, Section 6 concludes the paper.

2. Related Work

Caching technology has been proven to be an effective way in improving the system scalability. Many research efforts have been investigated on caching technology to enhance RMI performance of distributed object systems [5,6,7].

In 2000, Krishnaswamy et al. have adopted caching technology to provide the immediate requirements for running interactive distributed applications [5]. Everhard and Tripathi have presented a middleware so that caching technology can be insensibly added to existing RMI applications [6]. Furthermore, to reduce the overhead of maintaining the consistency of the cached copies and provide a scalable consistency protocol, Ahamad and Kordale [7] have proposed a local consistency (LC) mechanism by allowing users to control over the updates of the cached objects. However, these approaches do not consider incorporating the emerging active network technology. Therefore, the caching infrastructure can only be a simple two-level structure.

In ActiveRMI, a hierarchical caching scheme is provided so that the network traffic for maintaining the consistency can be reduced. Valid code or data can be fetched by traveling up the hierarchy rather than only remote servers. Furthermore, client numbers that remote servers need to handle are reduced because server workload is shared with intermediate active routers. Therefore, the overall ActiveRMI performance can be improved efficiently.

3. ActiveRMI Background

ActiveRMI was first proposed in [4]. The traditional Java RMI mechanism is improved by using active networks technology. This section provides a brief introduction to ActiveRMI as the background for further explanation on our design of consistency control.

3.1 Pre-processing ActiveRMI Applications

Because of the programmable characteristic of active networks, applications can be deployed over the network by transmitting active packets called *capsules.* The code is carried in capsules and injected into active routers. To deploy ActiveRMI services, capsule-related code must be automatically generated in ActiveRMI applications

As depicted in Figure 2, an ActiveRMI application contains several Java parts: an interface, a client stub, and a server skeleton. The application code is then parsed and transferred to add ANTS-specific code. MIT's ANTS is taken as the active network basis because of its versatility. After the pre-processing, ActiveRMI services can be deployed with capsules, and cached at ActiveRMI-enabled routers. When an ActiveRMI service is cached, ANTS can initiate it to provide the RMI service at the active router.

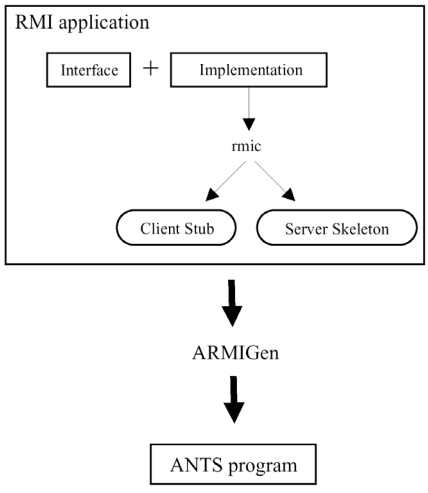

Figure 2: The cached ActiveRMI code.

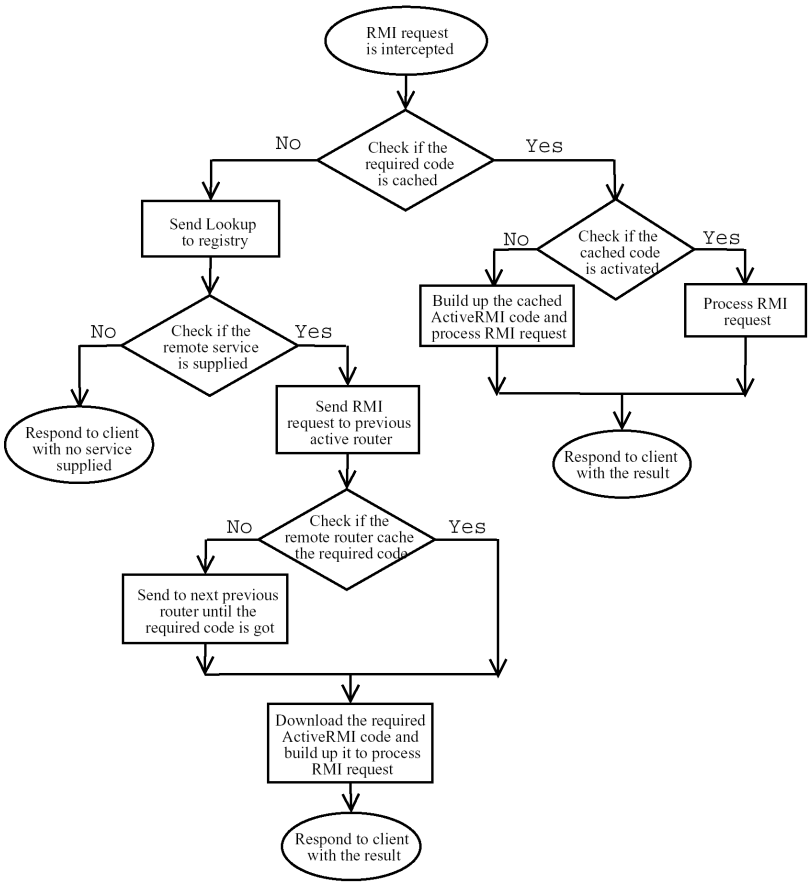

Figure 3: The flowchart of the ActiveRMI invocation protocol.

3.2 ActiveRMI Invocation

In ActiveRMI, an invocation protocol is designed to initiate ActiveRMI services at proximate active routers. Figure 3 describes the flowchart of the ActiveRMI invocation protocol. When an RMI request is intercepted by a proximate active router, the active router will check whether the required code is cached. If the required code has been cached and is still valid, it will be activated to process RMI request immediately. Otherwise, the active router sends a `Lookup` message to a remote registry to find the required RMI service. The required RMI service can be downloaded either from the original RMI server or an active router that is near the RMI server and has a valid cache copy. After the ActiveRMI code is downloaded, ANTS then creates the execution environment and initiates the service. Finally, the response will be sent back to the client and the ActiveRMI service is stopped.

4. Code Consistency Maintenance

Because an ActiveRMI service can be cached at several intermediate active routers, code inconsistency may occurs if the ActiveRMI code is modified at the server but not updated at the routers. To let the cached ActiveRMI code be up-to-date without incurring a large amount of overheads, an effective updating is needed.

In our design of code consistency maintenance, two approaches are adopted for performance consideration. The first approach is to minimize the amount of the network traffic incurred in maintaining the code consistency. Only the difference of the updated ActiveRMI code is transported. The second approach is for fault-tolerance consideration. A TTL (Time-To-Live) scheme is used to assign a lifetime to the cached ActiveRMI code to avoid invocating obsolete RMI services when network partitions occur.

When the lifetime of a cached ActiveRMI service is expired, the active router will query the RMI server whether the ActiveRMI code is updated during next invocation. If the code is modified, the fresh ActiveRMI code is downloaded to the active router. Otherwise, the TTL of the cached ActiveRMI code is reset.

5. Data Consistency Maintenance

In addition to code consistency maintenance, data consistency maintenance cannot be ignored in ActiveRMI design. Three design considerations are addressed. First, the sharing state of a cached ActiveRMI code is maintained in a session-based consistency aspect. Second, a cache manager called ARMIMan (ActiveRMI Manager) is designed to maintain the cache. Finally, ARMIMan executes a

consistency protocol to exchange consistency control information.

5.1 Sharing Considerations

Java provides four access controls *public*, *private*, *protected*, and *friendly* for classes, variables, and methods. A Java method can be further defined as *transient* or *persistent* according to whether the data of the method are *read-only* or *writable*. When clients want to access a public and persistent ActiveRMI method, data inconsistency may occur. ARMIMan maintains the data consistency of the cached ActiveRMI code depending on the sharing attributes of the methods.

Because only the object provider knows whether a method is transient or persistent, it needs to specify the access attribute of each method. When the remote server publishes an RMI service, the remoter server registers the access attribute of each method along with the remote RMI code. When an active router downloads an ActiveRMI code, the ARMIMan records the access attributes of the methods in a cache table to maintain the consistency of the cached ActiveRMI code.

5.2 The Cache Table

The ARMIMan at an active router manages the cached ActiveRMI code by recording the related information of the cached ActiveRMI code in a hash table and maintaining the state transfer of the methods. The following considerations are addressed.

1. ARMIMan records the client list and the method list to show a client invokes a method. When the data of the cached ActiveRMI code needs to be updated, ARMIMan informs the clients or active routers who invoke the same method by checking the client list.

2. Because of the finite resource or space on an active router, ARMIMan evicts the cached ActiveRMI code by the LRU replacement algorithm with double linked list data structure. The longest last access time (LAT) of the cached ActiveRMI is evicted first. Therefore, a LAT field and the previous and the next pointer of the cached ActiveRMI code are recorded.

3. Furthermore, states of the invoked methods are recorded. The state is either one of the following: *isValid*, *Exculsive*, and *isWritable*. Whether a method of the cached ActiveRMI code is transient or persistent is also recorded in the *isWritable* field. When a client wants to invoke a transient method of the cached ActiveRMI code, ARMIMan checks the *isValid* field. On the other hand, the *Exculsive* field needs to be true when a client wants to modify a persistent method.

5.3 Data Consistency Protocol

In ActiveRMI, the consistency protocol allows either a single writer or multiple readers at a given time. The consistency protocol is designed with the consideration of reducing the network traffic between the clients and the remote server. Therefore, five approaches are adopted to maintain the data consistency of the cached RMI code in ActiveRMI, including callback promise [7], delayed-write [8], session guarantee [9], write-invalidate [10] and write-shared [8]. They are elaborated in the following paragraph.

1. When an active router wants to cache ActiveRMI code, it must get a callback token, which has a valid or cancelled state from the remote server. When a client wants to modify a persistent method of the cached ActiveRMI code, this event invokes the remote server to invalidate other copies by sending a callback token with cancelled state. Only the client who caches the ActiveRMI code in *exclusive* mode, called owner, can modify the data of the cached ActiveRMI code, while other copies are in *read-only* mode.

2. The update of the invoked method is stored at the local buffer of an active router temporarily until the session is closed, rather than being sent to the remote server immediately.

3. To avoid causing unnecessary network traffic, the remote server does not update the data to other cached ActiveRMI code until the other client wants to read or modify the data. When a client experiences a read miss or write miss on the data of the cached ActiveRMI code, a request is sent to the previous active router or the remote server to get the updated data.

4. The remote server updates the cached ActiveRMI code only by transmitting the changes of the method or the cached ActiveRMI code rather than the whole method or ActiveRMI code.

In ActiveRMI, the forgoing approaches are integrated to execute the consistency protocol. Three cases are maintained in the ActiveRMI consistency protocol. A paradigm running the consistency protocol is depicted in Figure 4. In the paradigm, the same ActiveRMI code is cached at the two active routers. These active routers maintain the consistency by recording the object states of the cached ActiveRMI code, including V, *isValid* and E, *isExclusive*. A client invokes the cached ActiveRMI code in exclusive mode is called *owner*. The ARMIMan executes the consistency action depending on whether the invoked method of the cached ActiveRMI code read or write the object state.

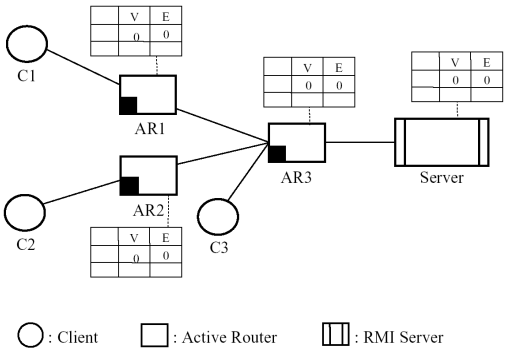

Figure 4: An example of running the ActiveRMI data consistency protocol.

In the read case, a paradigm is illustrated in Figure 5. When client C1 wants to read the object state of the cached ActiveRMI code and experiences a read-miss, the ARMIMan of AR1 needs to communicate with the server. One of the following two cases will be occurred. One is that no other client invokes the ActiveRMI code in *exclusive* mode, and then AR1 fetches the valid data on the way to the server. If an intermediate active router AR3 caches the ActiveRMI code with the valid data, the valid data will be sent back to AR1 and the method of AR1 is changed to be valid, as is showed in Figure 5. Otherwise, if no one owns the valid data except the server, the server updates the valid data along the way between the active router and the server. The other case is that while C1 wants to read the object state of the ActiveRMI code, C3 accesses it in *exclusive* mode, as depicted in Figure 6. The request of C1 then is sent to the server to ask to downgrade the other copies. The server is responsible to downgrade the copy of AR3 to *read-only* mode and ask AR3 to send the valid data to AR1.

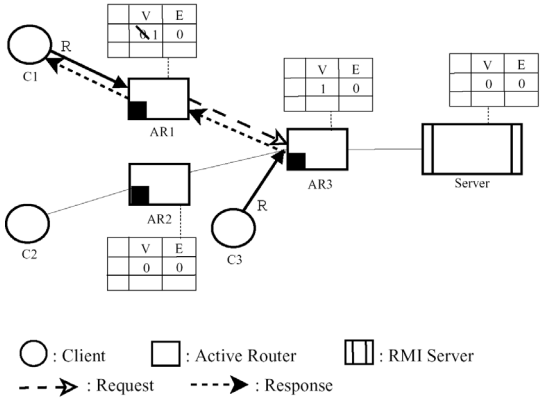

Figure 5: A read-read case of the ActiveRMI data consistency protocol.

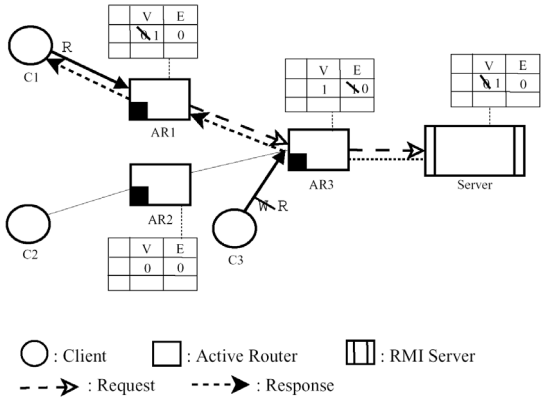

Figure 6: A read-write case of the ActiveRMI data consistency protocol.

In the write case, when C1 attempts to perform a write operation on the object state of the cached ActiveRMI code, the object state needs to be set in *exclusive* mode. Therefore, the ARMIMan of AR1 has to communicate with the server that coordinates the cached ActiveRMI code. Two possible cases need to be handled by the remote server. In one case, taking Figure 7 as an example, if the object state of the cached ActiveRMI code is in *exclusive* mode at another active router, AR3, the server downgrades the copy of AR3 and asks AR3 to send the latest data to AR1. In the other case, if the object state of the cached ActiveRMI code is in *shared* mode that many active routers, AR2 and AR3, cache the ActiveRMI code with valid data, then the server invalidates all of them and asks AR3 to send the valid data to AR1, as depicted in Figure 8.

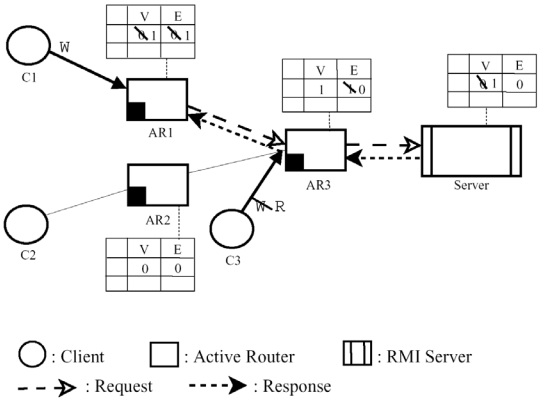

Figure 7: A write-write case of the ActiveRMI data consistency protocol.

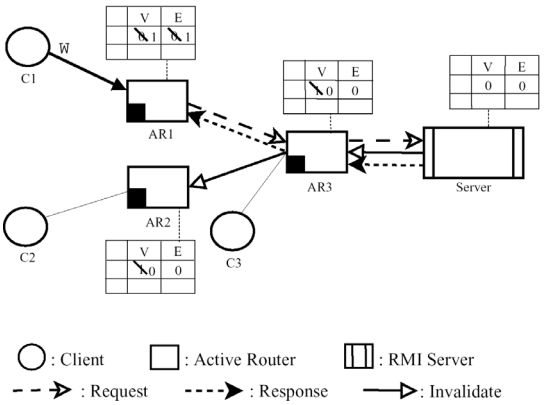

Figure 8: A write-read case of the ActiveRMI data consistency protocol.

6. Conclusions

To avoid extra network traffic for maintaining the code and data consistency of the cached ActiveRMI code degrades the ActiveRMI performance instead, in this paper the design of the code and data consistency maintenance in ActiveRMI is presented. Because ActiveRMI provides a hierarchical caching scheme and several consistency strategies are employed to maintain the code and data consistency of the cached ActiveRMI code, the amount of long-distance network communication is reduced. In the future, performance evaluations and more comprehensive experiments of ActiveRMI will be conducted for further studies. Improvements on cache consistency protocol are also in our future working schedule.

References

[1] A. D. Birrell and B. J. Nelson. “Implementing Remote Procedure Calls.” *ACM Transactions on Computer Systems*, 2(1): 39-59, February 1984.

[2] C.-Z. Yang and C.-W. Chen. “Issues on the RPC Paradigms in Active Networks.” In *Proceedings of the Active Networking Workshop*, pp. 100-105, September 2002.

[3] P. Tullmann, M. Hibler, and J. Lepreau. “Janos: A Java-Oriented OS for Active Network Nodes.” *IEEE Journal on Selected Areas in Communications*, 19(3): 117-131, March 2001.

[4] M.-C. Wueng. “Design of Code Caches in Active RMI.” Master Thesis, Yuan Ze University, July 2003.

[5] V. Krishnaswamy, I. Ganev, J. Dharap, and M. Ahamad. “Distributed Object Implementations for Interactive Applications.” In *Proceedings of IFIP/ACM International Conference on Distributed System Platforms*, pp. 45-70, 2000.

[6] M. Ahamad and R. Kordale. “Scalable Consistency Protocol for Distributed Services.” In *Transactions on Parallel and Distributed System*, 10(9): 888-903, 1999.

[7] J. Eberhard and A. R. Tripathi. “Efficient Object Caching for Distributed Java RMI Applicaitons.” In *Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms*, pp. 15-35, 2001.

[8] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. “Scale and Performance in a Distributed File System.” *ACM Transactions on Computer Systems*, 6(1): 134-154, 1988.

[9] J. Carter, J. Bennett, and W. Zwaenepoel. “Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems.” *ACM Transactions on Computer Systems*, 13(3): 205-243, 1995.

[10] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. “Session Guarantees for Weakly Consistent Replicated Data.” In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Information System*, pp. 140-150, 1994.

[11] P. Stenstrom. “A Survey of Cache Coherence Schemes for Multiprocessors.” *IEEE Computer*, 23(6): 12-24, 1990.