Performance Evaluation & Optimization about Lookup Service in Jini Architecture

Xidong Wang, Su Zhang
{wxd, zs}@cs.wisc.edu
Department of Computer Science, University of Wisconsin Madison

Abstract

Jini[tm] network technology provides a simple infrastructure for delivering services in a network and it is easy to build a efficient distributed system over it since direct transfer happens between client and server. Lookup service helps offer and find service and plays central role in Jini architecture. Design and implementation of lookup service in a prototype Jini-based NASD system are discussed in detail and the bottleneck during running is identified through detailed measurement based on log and source code analysis. We conclude that main cause of bottleneck is lookup service's implementation, particularly concurrency algorithm of RMI in which lookup service resides. Optimization techniques taken from Apache model are employed to improve performance. Also we propose that multiple replicated lookup services should cooperate to serve clients request in one local network. When Jini architecture scale up to web-wide network, we discuss the ongoing research with lesson from DNS.

1. Introduction

Jini architecture exploits features of the Java application environment to simplify the construction of a distributed system, in which the network supports a fluid configuration of objects which can move from place to place as needed and can call any part of the network to perform operations. Jini architecture is shown in Figure 1. Services are found and resolved by a lookup service. The lookup service is the central bootstrapping mechanism for Jini and provides major point of contact between service and client. In precise terms, a lookup service maps interfaces indicating the functionality provided by a service to sets of objects that implement the service. A service protocol, discovery/join protocol, is provided to allow services (both hardware and software) to discover, become part of, and advertise supplied services to others.

Communication between services can be accomplished using Java Remote Method Invocation (RMI[tm]). The infrastructure to support communication between services is not itself a service that is discovered and used but is, rather, a part of the Jini technology infrastructure. RMI provides mechanisms to find, activate, and garbage collect object groups. Fundamentally, RMI is a Java programming language enabled extension to traditional remote procedure call mechanisms. RMI allows not only data to be passed from object to object around the network but full objects, including code. Much of the simplicity of the Jini system is enabled by this ability to move code around the network in a form that is encapsulated as an object.

Clients and servers can take unicast to build connection with lookup services if they know the location of lookup services, or take advantage of multicast to find out lookup services. After that, clients can query for server objects for specific interface, and server can register its own server object, both using RMI mechanism. After a client gets server object provided by service from lookup service, it can communicate with service directly afterwards. Lookup service, which serves as a repository of services, intuitively becomes a bottleneck if there are many clients or servers requiring its service.

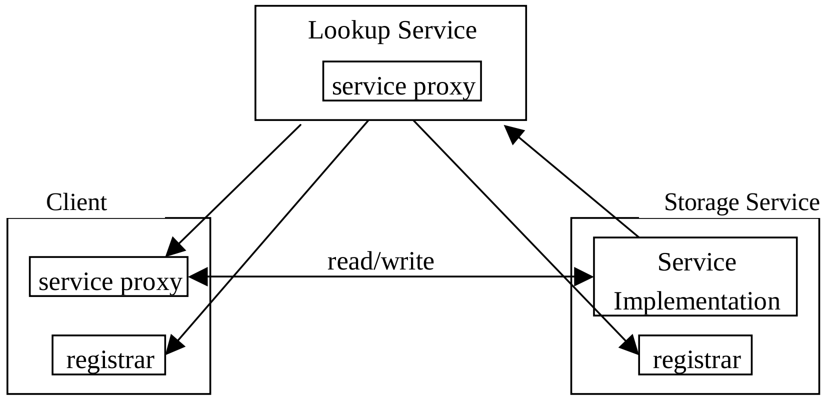

Figure 1: Jini architecture

This paper will talk about lookup service in Jini architecture, discuss component interactions when running, points out the cause of bottleneck in lookup service, and present some performance optimization. In section 2 a detailed analysis about lookup service is presented and its bottleneck is pointed out. In section 3 some optimization techniques are employed to improve performance, both in lookup service implementation and in lookup architecture. Section 4 will talk about some future developments, which can further this study. Section 5 concludes the discussion.

2. System Analysis

We can suspect that reggie server, which is the lookup service of Jini, is the bottleneck intuitively for in architecture it is in critical path of lookup service. This is proved in this section and a detailed analysis is provided to figure out the exact cause of bottleneck. Conclusion that concurrency algorithm in RMI is cause of bottleneck is deduced from experimental result and source code analysis.

Experiment Environment

System analysis is realized in JNASD, a NASD system based on Jini, since NASD share same philosophy with Jini: direct transfer between client and server. System is running on tux machine in pub lab and developed in Red Hat Linux operating system, with Apache as web server. First Jini is installed in system and then a very simple network attached storage system is designed and implemented using Jini API. Only read and write object functions are implemented for we really care about the lookup service in architecture.

Lookup functionality in server side is carefully analyzed by injecting some logging codes into source code of Jini and Reggie to make clear definitely how they operate. Also some logging codes are injected into Java source code, mainly into RMI part, to find out the algorithm of RMI and how RMI cooperate with reggie. When total NASD system is started and running, log information will be output to log file. An analyzing program is developed to filter out useful information from log according to Jini running procedure we have already known from source code. Most of log information is about temporal cost of some specific step.

In fact we encounter many difficulties in this environment setup step. Fortunately solution is attained at last for most of them. One of them is that Java source code distributed from Sun cannot be built as a whole because tux machine provide a different assemble language from what Java expects in building JVM before compiling other codes. The final system is running in AFS environment of public lab. Each test committed is influenced by share of machine and network bandwidth so that test results cannot be repeated. So for each experiment we take multiple times at distinct time and choose average value as final result.

A complete multicast lookup procedure is shown in detail. Major measures taken in client and server side are displayed in Figure 2. A detailed performance evaluation about major steps is carried on and its result is showed as follows. Then reason of bottleneck is figured out based on experimental result.

Multicast lookup procedure

When a client begins to query reggie server for some server object, commonly it choose multicast mechanism to discovery the location of reggie server. What client program really does is to create a new LookupDiscovery object and this object will communicate with reggie server to get back server object.

The first major step LookupDiscovery takes is to join multicast group so that it can make use of multicast protocol to communicate with reggie server. Then it informs reggie its listening port by including it in a multicast datagram and sending it to reggie. When reggie receive multicast datagram, it spawns a thread, start a TCP connection to LookupDiscovery's listening port. Then LookupDiscovery spawns a child thread to process later operation. The thread sends proto version to reggie's thread by TCP, reggie's thread will response by returning the object content of registrar proxy stub, which LookupDiscovery's thread will utilize it to take a RMI lookup action by cooperating with RMID in reggie server. The class representation of registrar proxy stub is retrieved from Apache server in reggie side.

Now LookupDiscovery's thread can take RMI call to reggie side. The code to call lookup action is injected by client program. The thread sends template parameter of client code to reggie side by connecting to reggie registered port in RMI. The RMI will spawn a thread to process this request. RMI thread will invoke the reggie's lookup function to process client's request and send back content of query result to client. Any class representation of result object is retrieved from web server of storage server side.

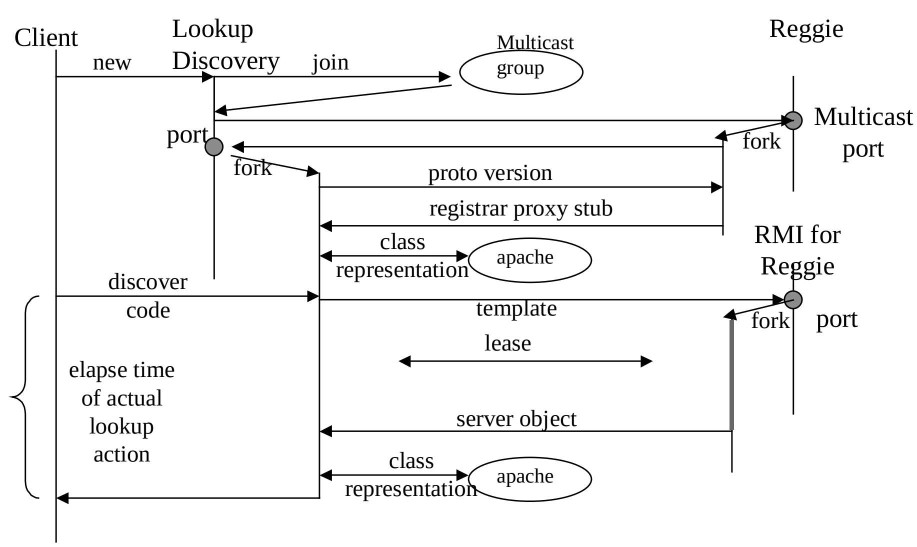

Figure 2: multicast lookup procedure

Trend of Actual Client Lookup Time

Multiple clients using multicast are started to emulate the real running situation. Elapse time of actual lookup action in client side is taken as candidate for measurement and the result shows that when number of concurrent lookup surpass some specific threshold, the average lookup time will increase substantially.

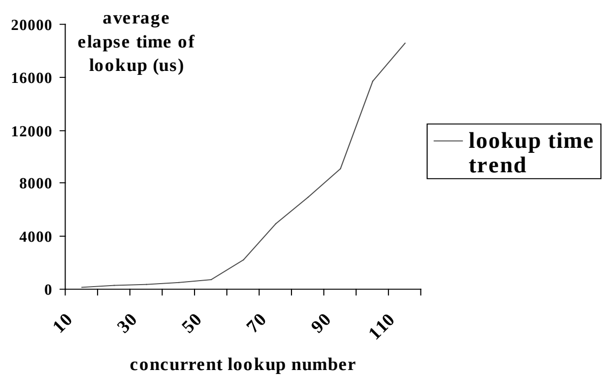

Figure 3: trend of actual client lookup time

What causes the bottleneck?

From figure 3 above we can conclude reggie become a bottleneck when there are multiple concurrent lookup requests. An intuitive remark borrowed from Database system is that query action on server object collection of reggie is the bottleneck since multiple query actions are being performed concurrently. However, that is not the truth.

Data collection in reggie side is indexed mostly by efficient Hash Map, so it takes very little time to perform a query action. reggie is also designed to serialize these concurrent query actions to resolve conflicts. Experimental data shows that query action on collection only takes less than 1 us for 200 registered objects, no matter how many concurrent lookup clients are running. And the trend does not change when number of registered server objects increase. Therefore we can conclude that query on server object collection of reggie is not the cause of bottleneck.

There is one inference about conclusion above. In some data warehouse applications data in collection are divided among multiple machines to enlarge the query throughput. Since reggie server does not share the feature of such applications that query action on data is bottleneck, it is no point to divide server objects registered in one reggie instance among others.

Now a detailed evaluation to major steps in reggie side should be taken to show the exact cause of bottleneck. Fine-grained steps are selected out as test candidate as show in below.

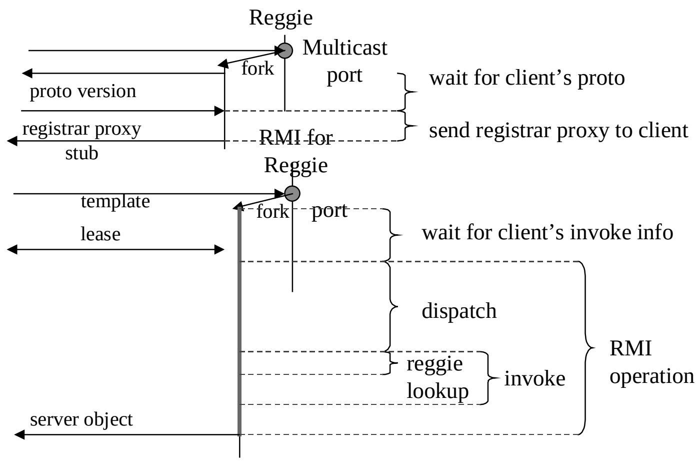

Figure 4: description of reggie side actions

Result about test above is shown in Figure 5. The time of dispatch and total RMI operation increase most substantially. Other parts except reggie lookup action also increase but with lower speed. Detailed source code analysis is taken and we found that for one procedure, performance optimization techniques, such as cache, have been employed. The problem is how RMI schedules among multiple concurrent requests. Each time RMI server receive a client call request, it create a thread to process client call. Many concurrent threads will degrade performance by competing limited CPU and memory resource and wasting time on scheduling.

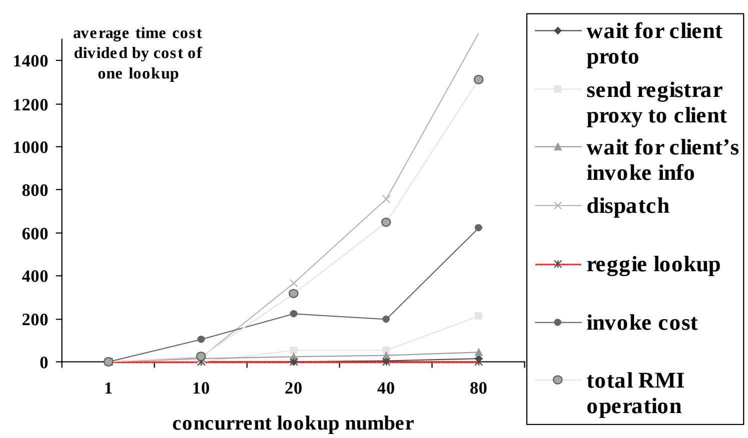

Figure 5: time cost of major steps in reggie side

Therefore we can draw a conclusion about cause of bottleneck that it is the concurrency mechanism in RMI implementation that cause the performance degrade dramatically when processing multiple requests.

3. Performance Optimization

In this section firstly we will try to improve performance of one reggie instance. After that lookup service architecture is modified and several same reggie instance take over the work of one reggie and client are suggested to replace multicast with unicast.

Change concurrency algorithm of RMI

Lesson can be taken from some network server application, whose intuitive design is also to spawn a thread to process one request. Now the design has been changed. Aapche of Unix version, taken as a example, keep one main thread or process listen for client requests, and add one job for each client request to job pool. Several (5 default) threads or processes remove one job each time from job pool and process client request represented by the job. Therefore scheduling cannot waste too much resource. We take apache model, using one worker to show effectiveness.

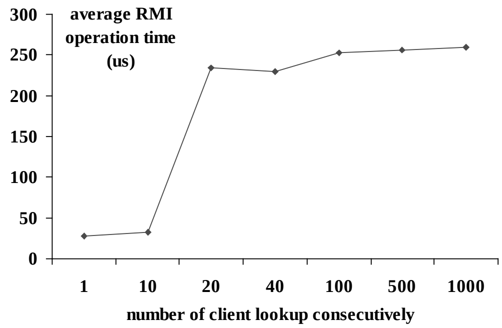

Figure 6: one worker for RMI

There is still need for one multi-thread worker pool. Because RMI operation is not completely CPU-bound, it includes disk and network operation when it acquire class representation during service, so several concurrent workers can exploit resource better. However, we can not decide the number of worker threads in pool now for it is decided by the weight of disk and network overload over CPU work.

There is still heavy burden on RMI daemon and HTTP server reggie parasite in when there are multiple client requests. Multiple replicated reggie instance should be started on different machine to exploit their RMI and HTTP resource.

Multiple replicated reggie instance

Our proposal is to keep multiple reggie instance of replicated content in local network, each on different machine, depending on different Apache and RMI daemon. It is noticed that only stub of server object is stored in reggie, therefore memory space put no obstacle to this design. It is easy to scale up. New reggie instance can be spawned with up-to-date data from existing reggie instance. One reggie instance can revoke its service without notifying its peer and system can run normally. It also improves robustness by removing single point of failure under one reggie instance situation

This alternation in design recommend client to replace multicast with unicast. Unicast can avoid time (70 us) and network bandwidth cost by multicast. It also save reggie multicast process work. It is feasible since reggie instance announce its existence periodically. Client can cache such announcement information and select randomly one service instance for unicast.

Server can parasite some load information in announcement so that client can take load-aware selection before unicast to further performance optimization.

4. Future Work

Our study suggests that more research on Java performance should be taken to make clear the weakness of its implementation. Another trend is to scale our lookup service design in local network up to web. It reminds us of DNS architecture. There is similarity between these two services since they both provide lookup functionality: query on local database by efficient index, and this query takes a relatively small fraction of process overhead.

We can learn from DNS and adopt Hierarchical structure to organize reggie servers in web-wide network. When local reggie server can not satisfy client interface request with data in local collection, it acts as agent of local client, and send interface query to reggie server in other network. When result return, it can transfer result to client and also store in local collection for reuse. Therefore some upper-level servers are needed to store location information of reggie servers indexed by interface. These upper-levels can also be organized in hierarchical structure. The architecture is shown as below.

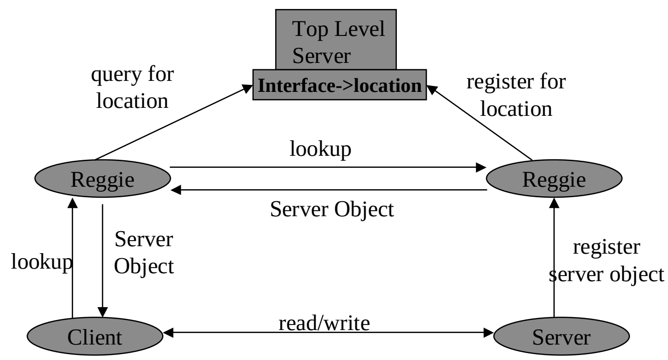

Figure 7: architecture of JNS

5. Conclusion

Jini architecture allows fluidity of all components in a distributed system, extending the easy movement of objects to the entire networked system, thus making it a proper candidate to build distributed system. The lookup service in Jini architecture serves as a central marketplace for offering and finding services. It accepts service provider to register service proxy, helps client to communicate with service provider by delivering service proxy to clients when clients request.

We make study about design and implementation of lookup service in a prototype Jini-based NASD. Source code of RMI, Jini, Reggie are carefully analyzed and some logging codes are injected to produce log information our log analyzer program can make use. Measurements show that the coordination algorithm of RMI is the main cause of performance degradation when multiple clients request concurrently because one thread per query will waste resource on scheduling work.

Performance optimization is employed both by taking coordination model from Apache and by keeping multiple same lookup service instances in local network and changing client request mode from multicast to unicast. Also we propose that some hierarchical structure of DNS should be immigrated to Jini lookup architecture to scale Jini to web-wide network.

Referrence

1. Jini Home page http://www.sun.com/jini/index.html
2. RMI Home page http://java.sun.com/products/jdk/rmi/index.html
3. Jini source code and reggie souce code
4. Jini tutorial http://pandonia.canberra.edu.au/java/jini/tutorial/Jini.html
5. RFC 1034 DOMAIN NAMES - CONCEPTS AND FACILITIES
6. RFC 1035 DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION
7. The JavaTM Virtual Machine Specification,

http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html
8. Java Language Specification, http://java.sun.com/docs/books/jls/second_edition/html/jTOC.doc.html
9. National storage industry consortium http://www.nsic.org/nasd/
10. High-bandwidth, Low-latency, and Scalable Storage Systems, http://www.pdl.cs.cmu.edu/NASD/
11. Java Remote Method Invocation, http://java.sun.com/products//jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html