Load balancing using Remote Method Invocation (JAVA RMI)

Ms. N. D. Rahatgaonkar 1, Prof. Mr. P. A. Tijare 2
1 Department of Computer Science & Engg and Information Technology
Sipna’s College of Engg & Technology, Amaravati,(M.S.) INDIA
2 Department of Computer Science & Engg and Information Technology
Sant GadgeBaba Amravati University, , Amaravati,(M.S.) INDIA

Abstract
Load balancing is the process of distributing client requests over a set of servers, and is a key element of obtaining good performance in a distributed application. Java RMI extends Java with distributed objects whose methods can be called from remote clients. RMI programs, there may be multiple replicas of a given overhead object that can be the receiver of a remote method in-vocation. Effectively distributing these requests across these replicas requires either an extra balancer processor additional code on the client for this distribution. this paper, we demonstrate the use of dynamic aspects in The client proxy is modified JAC to solve this problem, with an aspect to forward requests to a specific server, but the server is also able to shed load by altering or The overhead of this approach is removing this aspect evaluated using a set of micro benchmarks.

Keywords: Load balancing, Java RMI, Dynamic aspect-oriented programming.

1. Introduction
Java RMI (Remote Method Invocation) adds remote objects to Java programs. These remote objects reside on object servers, separate machines connected by a common network. Clients can invoke methods on these remote objects using remote method invocation, which bundles the information needed to invoke the method into a message and sends it to the appropriate object server for execution.

In compute-intensive remote object programs, clients may be invoking many expensive methods on servers. In particular, they may be invoking expensive methods on the same object, running the risk of decreasing performance by overloading the server. To improve performance this object can be replicated on several servers and requests can be distributed to a suitable replica, normally the server with the lightest load. This distribution of requests is referred to as load balancing, and is key to good performance in many distributed applications.

For this paper, we are specifically interested in dynamic load balancing; we do not assume that workload is constant and predictable enough that statically allocating replicas to clients will produce an optimal solution. We also assume that while the client must participate in a load balancing scheme, it does not have sufficient information to determine the best strategy for invoking remote objects. That is, we assume the implementation of the remote objects is in the best position to select an appropriate load balancing strategy.

This paper describes an approach to load balancing in Java RMI based on dynamic distributed aspect-oriented programming using Java Aspect Components (JAC) [13]. Aspect oriented programming (AOP) allows code (in the form of advice) to be inserted at specific points in the execution of a program. Dynamic AOP systems allow advice to be added and removed at run-time. JAC also allows advice to be transmitted over a network and applied to objects running on other object servers. We use these capabilities to create a balancer process that modifies a proxy object on the client to direct its remote method invocations to a specific replica. This advice can be altered or removed by a server to redistribute the invocations to other replicas and shed load.

The research contributions of this paper are as follows. First, it shows the use of a dynamic, distributed AOP system to modify proxy code on the client. In contrast, most work in this area focuses on allowing the client to supply aspects to modify the server object. Second, this load balancing strategy is controlled by the balancer and server processes based on their knowledge of application requirements. The strategy can be altered at runtime by these processes as necessary.

2. Literature Review
Distributed Web-server architectures that use request routing mechanisms on the cluster side are free of the problems of client-based approaches. Architecture transparency is typically obtained through a single virtual interface to the outside world, at least at the URL1 level. The cluster DNS(Domain name system)— the authoritative DNS server for the distributed Web system’s nodes—translates the symbolic site name (URL) to the IP address of one server.

This process allows the cluster DNS to implement many policies to select the appropriate server and spread client requests. DLB(Dynamic load balancing) is used to provide application level load balancing for individual parallel jobs. It ensures that all loads submitted through the DLB environment are distributed in such a way that the overall load in the system is balanced and application programs get maximum benefit from available resources. In this of the DLB has two major parts. One is called System Agent that collects system related information such as load of the system and the communication latency between computers. The other is called DLB Agent which is responsible for performing the load balancing. System Agent has to run all configured machine on the environment whereas DLB Agent is started by the user. The structure and components of the DLB environment is shown in Figure 1.
AWED provides a more exible model that supports the distribution of pointcuts and advice separately [10]. A pointcut can specify the host on which a join point applies and the host on which advice should be run. AWED permits advice to run on several hosts, which makes it useful for replication consistency mechanisms. This research could have used AWED rather than JAC. One potential problem with the used of AWED is that it uses new language constructs to specify pointcuts, and it does not appear that these constructs can be parameterized with runtime arguments. The pointcut() method in JAC does permit such parameters, making it more exible. FORMI(Fragmented object RMI) is built on a fragmented object model, where an object implementation is split across fragments that can be individually distributed across a network [4]. Fragments can be used to replicate or partition objects across the network. In FORMI, a proxy is a fragment on the client. A fragment can be replaced with another during execution, allowing the distribution of responsibilities to change at runtime. FORMI provides the same exibility as our aspect-oriented approach, and also does not impact client code. However, FORMI introduces its own stub compiler, changing the development process. Same way Dynamic Load Balancing without Packet Reordering, this paper shows that one can obtain the accuracy and responsiveness of packet-based splitting and still avoid packet reordering. This introduces FLARE, a new traffic splitting algorithm. FLARE exploits a simple observation. Consider load balancing traffic over a set of parallel paths Figure2. If the time between two successive packets is larger than the maximum delay difference between the parallel paths, one can route the second packet [and subsequent packets from this flow] on any available path with no threat of reordering. Thus, instead of switching packets or flows, FLARE switches packet bursts, called flowlets. By definition, flowlets are spaced by a minimum interval, chosen to be larger than the delay difference between the parallel paths under consideration. FLARE measures the delay on these paths and sets the flowlet timeout, to their maximum delay difference. The small size of flowlets lets FLARE split traffic dynamically and accurately, while the constraint imposed on their spacing ensures that no packets are reordered.

Figure 2: As long as the inter-packet spacing is larger than the delay difference between the two paths, one can assign the two packets to different paths without risking packet reordering.

2.1 JAVA RMI (Java remote method invocation):

The Java Remote Method Invocation Application Programming Interface (API), or Java RMI, is a Java application programming interface that performs the object-oriented equivalent of remote procedure calls (RPC).

1. The original implementation depends on Java Virtual Machine (JVM) class representation mechanisms and it thus only supports making calls from one JVM to another. The protocol underlying this Java-only implementation is known as Java Remote Method Protocol (JRMP).

2. In order to support code running in a non-JVM context, a CORBA version was later developed. Usage of the term RMI may denote solely the programming interface or may signify both the API and JRMP, whereas the term RMI-IOP (read: RMI over IIOP) denotes the RMI interface delegating most of the functionality to the supporting CORBA implementation.

The programmers of the original RMI API generalized the code somewhat to support different implementations, such as a HTTP transport. Additionally, the ability to pass arguments “by value” was added to CORBA in order to support the RMI interface. Still, the RMI-IOP and JRMP implementations do not have fully identical interfaces.

RMI functionality comes in the package java.rmi, while most of Sun’s implementation is located in the sun.rmi package. Note that with Java versions before Java 5.0 developers had to compile RMI stubs in a separate compilation step using rmic. Version 5.0 of Java and beyond no longer requires this step. Jini offers a more advanced version of RMI in Java. It functions similarly but provides more advanced searching capabilities and mechanisms for distributed object applications.

3. Analysis Of Problem

A second option is to augment the object registry to allow multiple remote objects to register remote object references using the same name. When a lookup is performed, the registry can return one of the registered references to the client, and the client invokes methods directly on that object. This approach cannot be implemented using the RMI registry supplied with Java RMI. That registry does not permit multiple entries for the same name, instead throwing an exception on successive attempts to bind with the same name. Instead, a customized registry must be used. Examples of this approach include the SmartRegistry [9] and Jgroup/ARM [8], though both systems allow multiple registrations to support replication rather than load balancing. In these systems, the registry returns a proxy with references to all available replicas for group communication. In a load balancing system, the registry would return a single reference to one of the registered objects. Another example of this approach is DNS load balancing. DNS databases can have multiple IP addresses associated with a name, and DNS servers can be configured to return different addresses according to a policy. DNS load
balancing has several limitations. First it can only return different IP addresses, so all servers on these hosts must use the same port number. Second DNS does not check the availability of a machine before returning its address, so it may return the address of a crashed server.

A problem with this strategy is that the client generally caches the object reference and uses it for all remote method invocations, which limits the ability to balance load. That is, rather than redirecting each request from each client to a different server, this strategy redirects all requests from a client to the same server. It is possible that different clients will issue a different number of remote calls and will cause different loads at their servers. To redirect requests to another server, a client would need to obtain a new remote reference by issuing another lookup to the registry, and ensure that all client code uses the new remote object. In DNS load balancing, this problem is exacerbated by the ability of intermediate name servers to also cache results, although this can be mitigated by setting the time-to-live field on the address entry to expire relatively quickly. Another serious problem with this strategy is that it falls to the client to detect and redistribute the load if the server becomes overwhelmed, by reissuing a lookup request.

There is no simple mechanism for allowing the server to automatically reduce its load by not accepting new requests. A load balancer process is placed between clients and servers. All client requests are forwarded to the balancer process, which forwards the request to a suitable server. The reply message takes the reverse path. This is shown in Figure 3. In Java RMI, the balancer would maintain a collection of references to different remote objects. For each incoming request, one of these remote objects would be selected and the balancer would invoke the same method on it, forwarding the request. The return value of the balancer method would simply forward the return value from the remote object.

A similar strategy can be used in Apache, forwarding all requests to an entry server that rewrites the URL to redirect the request to one of a set of servers [1].

This strategy has the benefit of being able to redirect each request from each client to a suitable server. In addition, incorporating new servers is relatively simple. When a new object starts on a new server, it could register itself with the balancer. From that point, the balancer could distribute requests to the new object. The balancer can also control the load on the servers by deciding how many requests to forward to any given server. Once this number has been reached, the balancer could queue up requests and forward them to servers as they complete their outstanding requests. However, this strategy adds communication overhead in the extra pair of messages between balancer and server. This overhead can be reduced by having the server reply directly to the client, which is not possible in Java RMI without altering the underlying RMI protocol. In addition, the balancer can potentially form a bottleneck since all requests must pass through it, though the amount of processing for each request is small. However, our approach has a basic security mechanism in that the client must explicitly run the proxy in a JAC-aware container to allow the server to advise it. Finer-grained control is not possible, though there are several proposals on how to address this problem [6, 12].

Java RMI Architecture:

Java RMI is based on the distinction between object interface and implementation. It relies on the fact that a client cannot distinguish between objects implementing a remote interface if their behavior is identical.

The architecture of Java RMI consists of the the layers in Figure 4 [3]. The first layer provides a proxy object on the client and a skeleton object at the server. In current versions of Java, there is one skeleton object for the server. The proxy object is a local object on the client JVM that implements the same remote interface as the object implementation on the server. The proxy translates method invocations to remote method invocations to the server. Part of this translation uses the remote object reference for the remote object held in the Remote Reference Layer. The Transport Layer handles client/server communication.

The proxy object may be statically-generated by the rmic stub compiler or may be a dynamic proxy generated at runtime by the JVM. The rmic compiler starts with a class that implements a remote interface (one derived from java.rmi.Remote). From this, rmic generates a proxy class that implements the same remote interface. The name of this proxy class is the name of the implementation with “Stub” appended. For each method in the remote interface, rmic generates code that uses the remote object reference to invoke the same method on the object implementation at the server. At runtime, when the client imports the remote object using the RMI registry, it loads this proxy class using its name. If the proxy class is successfully loaded, a proxy object is created. If not, then the second method of proxy generation is used.

The second method of generating a proxy object is using the dynamic proxy mechanism introduced in Java 1.3 [16]. Given a list of interfaces, the JVM can create a proxy implementing them at runtime. Method calls on the proxy are delegated to an invocation handler object provided by the developer. In Java RMI, if the JVM cannot load the rmic-generated proxy class, the client creates a dynamic proxy using the remote interface. A RemoteObjectInvocationHandler object is created as the invocation handler, which provides identical
functionality as rmic-generated stubs. However, we consider these dynamic proxies to be statically generated. Their functionality is fixed; they are dynamic only in that they are created at runtime.

4. Proposed work

In this section, we’ll examine some software-based load balancing approaches that have been applied to Java RMI and similar distributed systems.

The first approach is perhaps the most obvious. A load balancer process is placed between clients and servers. All client requests are forwarded to the balancer process, which forwards the request to a suitable server. The reply message takes the reverse path. This is shown in Figure 3. In Java RMI, the balancer would maintain a collection of references to different remote objects. For each incoming request, one of these remote objects would be selected and the balancer would invoke the same method on it, forwarding the request. The return value of the balancer method would simply forward the return value from the remote object. A similar strategy can be used in Apache, forwarding all requests to an entry server that rewrites the URL to redirect the request to one of a set of servers [1].

This strategy has the benefit of being able to redirect each request from each client to a suitable server. In addition, incorporating new servers is relatively simple. When a new object starts on a new server, it could register itself with the balancer. From that point, the balancer could distribute requests to the new object. The balancer can also control the load on the servers by deciding how many requests to forward to any given server. Once this number has been reached, the balancer could queue up requests and forward them to servers as they complete their outstanding requests.

In addition, this strategy allows the servers (in conjunction with the balancer) to shed load when necessary. The balancer can factor in server load when distributing requests, by having each server periodically indicate its current status. The client is not involved in this process, instead simply forwarding all requests to the central balancer process.

However, this strategy adds communication overhead in the extra pair of messages between balancer and server. This overhead can be reduced by having the server reply directly to the client, which is not possible in Java RMI without altering the underlying RMI protocol. In addition, the balancer can potentially form a bottleneck since all requests must pass through it, though the amount of processing for each request is small.

A second option is to augment the object registry to allow multiple remote objects to register remote object references using the same name. When a lookup is performed, the registry can return one of the registered references to the client, and the client invokes methods directly on that object. This approach is shown in Figure 5.

This approach cannot be implemented using the RMI registry supplied with Java RMI. That registry does not permit multiple entries for the same name, instead throwing an exception on successive attempts to bind with the same name. Instead, a customized registry must be used. Examples of this approach include the SmartRegistry [9] and Jgroup/ARM [8], though both systems allow multiple registrations to support replication rather than load balancing. In these systems, the registry returns a proxy with references to all available replicas for group communication. In a load balancing system, the registry would return a single reference to one of the registered objects.

Another example of this approach is DNS load balancing. DNS databases can have multiple IP addresses associated with a name, and DNS servers can be configured to return different addresses according to a policy. DNS load balancing has several limitations. First, it can only return different IP addresses, so all servers on these hosts must use the same port number. Second, DNS does not check the availability of a machine before returning its address, so it may return the address of a crashed server.

A problem with this strategy is that the client generally caches the object reference and uses it for all remote method invocations, which limits the ability to balance load. That is, rather than redirecting each request from each client to a different server, this strategy redirects all requests from a client to the same server. It is possible that different clients will issue a different number of remote calls and will cause different loads at their servers. To redirect requests to another server, a client would need to obtain a new remote reference by issuing another lookup to the registry, and ensure that all client code uses the new remote object. In DNS load balancing, this problem is exacerbated by the ability of intermediate name servers to also cache results, although this can be mitigated by setting the time-to-live field on the address entry to expire relatively quickly.

Another serious problem with this strategy is that it falls to the client to detect and redistribute the load if the server becomes overwhelmed, by reissuing a lookup request. There is no simple mechanism for allowing the server to automatically reduce its load by not accepting new requests.

```
Client
  obj = lookup("ObjectName")
  obj.method()

Registry
  register("ObjectName", RemoteObj1)
  register("ObjectName", RemoteObj2)

  obj1 = lookup("ObjectName")
  obj1.method()

  obj2 = lookup("ObjectName")
  obj2.method()
```

Figure 5: Balancing using multiple registry entries

5. Implication

This style of load balancing addresses the limitations of the two schemes presented in above Section. It avoids the need for the load balancer to be involved with each request, reducing its load and reducing the network latency of forwarding each request. It avoids the problem of clients caching information, since that information is now held in as aspect to which the client is oblivious. It also allows the server to shed load by unweaving aspects from clients, so they can be directed to new servers. One desirable property of our strategy is that it lends itself to balancing client sessions, where a session is a series of individual requests that form one larger, logical re-
quest. It can be advantageous to forward all requests for a single session to the same server, but balance different sessions to different servers. In our strategy, the server can achieve this by simply unweaving advice from a client at the end of a session.

Our approach to load balancing uses dynamic, distributed aspects forwarded from server to client to advise client-side proxy objects. These aspects allow us to use a balancer process to distribute clients to servers, but also allow the servers to shed load when necessary.

This paper describes an approach to load balancing in Java RMI based on dynamic distributed aspect-oriented programming using Java Aspect Components (JAC) [13]. Aspect oriented programming (AOP) allows code (in the form of advice) to be inserted at specific points in the execution of a program. Dynamic AOP systems allow advice to be added and removed at runtime. JAC also allows advice to be transmitted over a network and applied to objects running on other object servers. We use these capabilities to create a balancer process that modifies a proxy object on the client to direct its remote method invocations to a specific replica. The overall process is shown in Figure 6 We start with what appears to be the standard solution using a balancer process, where clients initially send requests to the balancer. Indeed, the first request from each client is treated using this standard solution; the request is forwarded to a suitable server, and the results follow the reverse path. This is shown in messages 1, 3, 4, and 5. However, the balancer also dynamically weaves a JAC aspect on the client proxy while the request is being processed (message 2). This aspect alters the proxy to forward requests to a specific server. Thus, all subsequent requests are forwarded directly from client to the selected server (messages 6 and 7). Importantly, the balancer applies the aspect to the client when it is blocked awaiting the reply to its remote method invocation. Removing many of the potential concurrency problems that can arise in this type of system. A client is assigned to a given server and all requests are forwarded to the single location. Here, we exploit the ability to dynamically unweave an aspect in JAC. If the server is overloaded, it can unweave the aspect from the client proxy. After this unweaving, the next request from that client is forwarded to the balancer process, which then applies a new aspect to forward client requests to another server. That is, once the aspect is unwoven, the scenario in Figure 6 repeats itself for the client, except that the client may be associated with a new server. Again, this unweaving is performed when the client is blocked awaiting a reply to reduce potential concurrency problems. This style of load balancing addresses the limitations of the two schemes presented in above Section. It avoids the need for the load balancer to be involved with each request, reducing its load and reducing the network latency of forwarding each request. It avoids the problem of clients caching information, since that information is now held in as aspect to which the client is oblivious. It also allows the server to shed load by unweaving aspects from clients, so they can be directed to new servers.

6. Conclusion

In this paper, we presented a dynamic, aspect oriented implementation of load balancing in Java RMI. Initial requests from a client are directed to a balancer process, which forwards the request to a server while simultaneously weaving an aspect on the client proxy. The woven aspect instructs the client to forward all subsequent requests to a specific server. However, if that server needs to shed some of its load, it can unweave the aspect to force the client to find another server. This approach reduces the overhead of having all requests forwarded by a balancer process but provides a more dynamic ability to redistribute load when necessary. In addition, all decisions are made by the server based on application needs.

To evaluate this approach, we presented results from a series of micro benchmarks. These results show that the client does not suffer appreciable overhead from the advice woven into its proxy, and that the cost per request does fall by 30% by sending requests directly to a server without an intervening balancer process. However, the cost of weaving and unweaving advice is high and must be amortized over a number of requests from client to server.

In the future, we could improve this research by augmenting the underlying Java RMI protocol to piggyback aspects and advice with reply messages. Our current implementation tries to overlap the transmission and weaving of advice with the execution of remote methods using threads. This augmented protocol would require clients to be more aware of the new functionality, but would reduce the overhead in applying advice over the network and obviate the need for extra threads.

7. References

References as per IEEE format:


