Load balancing using java Aspect Component(Java RMI)

Ms. N. D. Rahatgaonkar [#1], Prof. Mr. P. A. Tijare [*2]
Department of Computer Science & Engg and Information Technology
Sipna's College of Engg & Technology, Amaravati,(M.S.) INDIA
Sant GadgeBaba Amravati University, , Amaravati,(M.S.) INDIA
[1]nanditarahatgaonkar@gmail.com [*2] pritishtijare@rediffmail.com

Abstract- **Various types of scheduling algorithms are used by load balancers to determine which backend server to send a request to. Simple algorithms include random choice or round robin. High-performance systems may use multiple layers of load balancing. Load balancing is the process of distributing client request over the set of servers and is a key element of obtaining good performance in distributed application.. Java RMI extends java with distributed objects whose methods can be called from remote clients.**

Keywords--**Load balancing, Scheduling, Remote Method Invocation**

I. INTRODUCTION

In computer networking, load balancing is a technique to distribute workload evenly across two or more computers, network links, CPUs, hard drives, or other resources, in order to get optimal resource utilization, maximize throughput, minimize response time, and avoid overload. Using multiple components with load balancing, instead of a single component, may increase reliability through redundancy. Simple algorithms include random choice or round robin. More sophisticated load balancers may take into account additional factors, such as a server's reported load, recent response times, up/down status, number of active connections, geographic location, capabilities, or how much traffic it has recently been assigned. High-performance systems may use multiple layers of load balancing. Java RMI (Remote Method Invocation) adds remote objects to Java programs. These remote objects reside on object servers, separate machines connected by a 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. 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 shows the use of a dynamic, distributed AOP(Aspect-oriented programming) 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. This presents background material covering several topics important to this research, including Java RMI, aspect-oriented programming with JAC(Java Aspect Component), and load balancing.

II. LITERATURE REVIEW

This section focuses on other research in distributed AOP. 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 URL 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 to perform 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

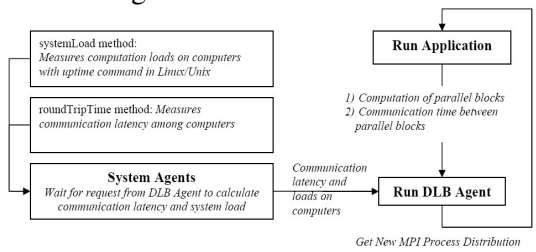

Figure 1: Components of DLB

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.

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.

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 introduce 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 land subsequent packets from this flowl 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.

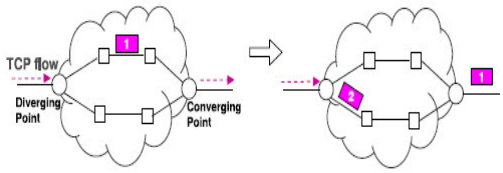

Figure 2: As long as the inter-packet spacing is larger than the delay difference between the two paths, one can assign the two packets to different paths without risking packet reordering.

III. 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 per formed, 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 Figure3. 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].

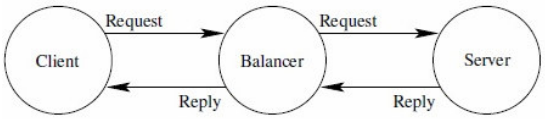

Figure 3: A Balancer forwarding request and replies

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].

IV PROPOSED WORK

This is to develop systems that dynamically turn on to be able to handle the load imposed on the system efficiently.

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 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.

The overall process is shown in Figure 4 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. This removes 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 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.

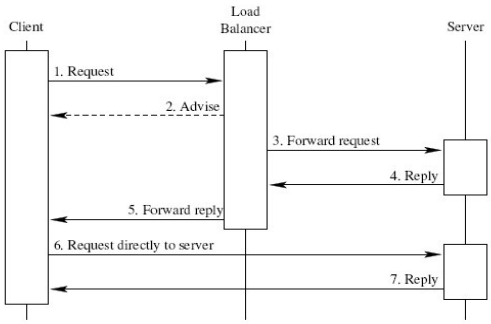

Figure 4: Load balancing with dynamic remote aspect

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.

V. 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 request. 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

In assessing our load balancing strategy, we first assessed the overhead of the first request by the client, which requires the balancer to not only forward the request to the server but also

weave advice into the client. From the perspective of the client, this first requests takes its an average. Note that we made sure to unweave the advice between successive calls to avoid any additional overhead that may accumulate by repeatedly weaving advice on the same join point. To reduce the overhead of this weaving, it is done by a separate thread that runs while the balancer forwards the method invocation to a server.

As a baseline, we measured the performance of the first load balancing strategy, where all requests must be forwarded through the balancer before being directed to an appropriate server. From the perspective of the client, requests take an average of 0.51ms per round trip. The request is simply forwarded to the server. In assessing our load balancing strategy, we first assessed the overhead of the first request by the client, which requires the balancer to not only forward the request to the server but also weave advice into the client. From the perspective of the client, this first requests takes an average of 5.64 ms. Note that we made sure to unweave the advice between successive calls to avoid any additional overhead that may accumulate by repeatedly weaving advice on the same join point. To reduce the overhead of this weaving, it is done by a separate thread that runs while the balancer forwards the method invocation to a server. After the first request, the advice woven at the client proxy forwards all subsequent requests directly to the server without involving the balancer. This avoids the overhead of an extra exchange of messages, but incurs the overhead of invoking advice woven on the client proxy. Some aspect-oriented constructs can incur a significant performance penalty in ways that are not obvious [2]. In particular, around advice may include closure objects, which are expensive to create and use. JAC uses Java Reflection which can be expensive. However, our balancing advice replaces the client proxy code rather than invoking it. To assess the overhead of the advice at the client proxy, we measured the cost of a remote method invocation with an advised proxy and the cost of normal remote method invocation directly from client to server. These two measurements give us the overhead of applying aspects to the client proxy. From the perspective of the client, the cost of a remote method with a balancing aspect was measured to average 0.357 ms per requests, which was identical to a round trip with an unadvised proxy. The advice adds no appreciable overhead to the execution of the proxy on the client. Our aspect-oriented version, which allows requests to be sent directly to a server after the first request, cuts 0.153 ms from each request, a savings of 30%. However, this does come at the cost of weaving, which takes 5.64 ms in JAC. This requires clients to forward 44 requests to a given server to make up for this extra cost. Again, though, this represents a lower bound since our experimental setup does not include any scheduling decisions, computation, or significant data transfer for method arguments and return values. In a more realistic environment, we expect this number to be lower. In particular, given that remote methods are normally large-grained to make up for communication overhead, this weaving process can be overlapped with the execution of the method and may impose little overhead in overall execution. For requests that benefit greatly from server affinity, this approach may be best even with its overhead.

When a server wishes to shed some of its load, it can unweave advice from a client proxy to force the proxy to forward its next request through the balancer. This requires an extra message from server to client while the remote method is executing. We can overlap this unweaving process with the execution of the remote method, just as we did when weaving the advice in the balancer. From the perspective of the client, a request that involves this unweaving takes an average of 8.14ms. It must be noted that our tests have an empty method body, so there is no overlap between method execution and unweaving. As a result, this number is the worst case. Again, unweaving can be overlapped with method execution to hide some of the cost.

VI. APPLICATION

A. A novel dynamic load balancing scheme for parallel systems

Adaptive mesh refinement (AMR) is a type of multiscale algorithm that achieves high resolution in localized regions of dynamic, multidimensional numerical simulations. This scheme interleaves a grid-splitting technique

with direct grid movements, for which the objective is to efficiently redistribute workload among all the processors so as to reduce the parallel execution time.

B. Dynamic Load Balancing in GlassFish Application Server

GlassFish is a fully Java EE 5-compliant application server with enterprise-ready features available under two OSI-approved licenses. Among many other enterprise-level features, GlassFish provides a very good self-management functionality extendable using the Java Management eXtension (JMX) standard. The GlassFish application server provides good facilities for cluster management and load balancing. We can use GlassFish self-management facilities, JMX, and the Application Server Management eXtension (AMX) APIs to change the load balancer configuration.

C. Dynamic load balancing of SAMR

To efficiently utilize computing resources provided by distributed systems, an underlying DLB scheme must address both heterogeneous and dynamic features of distributed systems. We also provide a heuristic method to evaluate the computational gain and redistribution cost for global redistribution. Experiments show that by using this distributed DLB scheme, the execution time can be reduced by 9%-46% as compared to using parallel DLB scheme which does not consider the heterogeneous and dynamic features of distributed systems.

D. Dynamic Load Balancing for a Grid Application

Grids functionally combine globally distributed computers and information systems for creating a universal source of computing power and information.

E. Load Balancing Web Applications

This provides a simple machine-level load-balancing mechanism, but is only appropriate for session independent or shared-session servers.

VII. 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 sub sequent 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.

REFERENCES

[1] Apache HTTP Server Project. URL Rewriting Guide, 2008. http://httpd.apache.org/docs/2.2/misc/rewriteguide.html.

[2] B. Dufour et al. Measuring the dynamic behaviour of AspectJ programs. In Proceedings of the 19th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 150–169, 2004.

[3] jGuru. Remote Method Invocation: Tutorial and Code Camp, 2000. http://java.sun.com/developer/onlineTraining/rmi/RMI.html.

[4] R. Kapitza, J. Domaschka, F. Hauck, H. Reiser, and H. Schmidt.FORMI: Integrating adaptive fragmented objects into Java RMI. IEEE Distributed Systems Online, 7(10), 2006.

[5] G. Kiczales et al.Aspect-oriented programming. In Proc. 11th European Conference on Object–Oriented Programming, volume 1241 of LNCS, pages 220–242. Springer-Verlag, 1997.

[6] D. Larochelle et al. Join point encapsulation. In Proceedings of the 2003 Workshop on Software Engineering Properties of not possible, though there are several proposals

[7] C. Lopes. D: A Language Framework for Distributed Programming. PhD thesis, College of Computer Science, Northeastern University, 1997.

[8] H. Meling et al. Jgroup/ARM: A distributed object group platform with autonomous replication management. Software: Practice and Experience, 2008. [9] N. Narasimhan et al. Interceptors for Java remote method invocation. Concurrency and Computation: Practice and Experience, 13(8-9):755–774, 2001. [10] L. Navarro et al. Explicitly distributed AOP using AWED. In Proceedings of the 5th International Conference on Aspect-Oriented Software Development, pages 51–62, 2006.

[11] M. Nishizawa et al. Remote pointcut: A language construct for distributed AOP. In Proc. 3rd International Conference on Aspect-Oriented Software Development, pages 7–15, 2004.

[12] H. Ossher. Confirmed join points. In Proceedings of the 2006 Workshop on Software Engineering Properties of Languages for Aspect Technologies, 2006.

[13] R. Pawlak et al. JAC: An aspect-based distributed dynamic framework. Software: Practice and Experience, 34(12):1119–1148, 2004.

[14] N. Santos et al. A framework for smart proxies and interceptors in RMI. In Proceedings of the 15th ISCA International Conference on Parallel and Distributed Computing Systems, 2002.

[15] A. Stevenson and S. MacDonald. Smart proxies in java rmi with dynamic aspect-oriented programming. In Proceedings of the 2008 International Workshop on Java and Components for Parallelism, Distribution and Concurrency, 2008.

[16] Sun Microsystems, Inc. Dynamic Proxy Classes, 2004. http://java.sun.com/j2se/1.5.0/docs/guide/re ection/-proxy.html

[17] Andrew Stevenson and Steve MacDonald: Dynamic Aspect-Oriented Load Balancing in Java RMI

[18] R.U. Payli, E. Yilmaz, A. Ecer, H.U. Akay, and S. Chien: DLB – A Dynamic Load Balancing Tool for Grid Computing

[19] Srikanth Kandula Dina Katabi Shantanu Sinha Arthur Berger: Dynamic Load Balancing Without Packet Reordering