DISTRIBUTED WEB SERVICE FRAMEWORK

Pranav Ganore
Department of CSEE
University of Maryland Baltimore County
Maryland, United States of America
Email: pganore1@umbc.edu

Tejhan Bharadwaj Ramkumar
Department of CSEE
University of Maryland Baltimore County
Maryland, United States of America
Email: kh50252@umbc.edu

Abstract:

The aim of this project was to create a distributed web services framework, which should gracefully handle scalability with respect to the deployed web services and balance the consumption of these services using load balancing mechanisms.

1. Introduction:

One of the main goal for this project was to build a distributed web services framework and getting to know the technicalities and traverse the depths of the various tools of trades available to implement web services in a distributed environment. Another of the requirements for this project was to provide the service descriptions using WSDL (Web Services Description Language). To satisfy these we chose Apache Axis-2 as our framework for building and deploying web services.

Apache Axis2 is a Web Services / SOAP / WSDL engine, the successor to the widely used Apache Axis SOAP stack. There are two implementations of the Apache Axis2 Web services engine - Apache Axis2/Java and Apache Axis2/C. Apache Axis2 not only supports SOAP 1.1 and SOAP 1.2, but it also has integrated support for the widely popular REST style of Web services. Axis-2 supports the Web Service Description Language (version 1.1 & 2.0), which allows us to easily build stubs to access remote services, and also to automatically export machine-readable description of your deployed services from Axis-2. It also supports asynchronous web services and asynchronous web services invocation using non-blocking clients and transports and many other features (1).

Axis 2 uses JAX-WS, in which a web service operation invocation is represented by an XML based protocol, such as SOAP. The SOAP specification defines the envelope structure, encoding rules, and conventions for representing web service invocations and responses. These calls and responses are transmitted as SOAP messages (XML files) over HTTP. Apache Axis-2 uses JAX-WS 2.0 which is a new programming model which simplifies application development through support of a standard annotation based model to develop web service applications and clients.

1.1 Apache Axis-2:

Apache Axis-2 is the core engine for a web service and it was re-designed from Apache Axis SOAP. It also has the capability to function as a standalone application server. Apache Axis2 is a Web Services / SOAP / WSDL engine, the successor to the widely used Apache Axis SOAP stack. There are two implementations of the Apache Axis2 Web services engine - Apache Axis2/Java and Apache Axis2/C. Apache Axis2 not only supports SOAP 1.1 and SOAP 1.2, but it also has integrated support for the widely popular REST style of Web services. Axis-2 supports the Web Service Description Language (version 1.1 & 2.0), which allows us to easily build stubs to access remote services, and also to automatically export machine- readable description of your deployed services from Axis-2. It also supports asynchronous web services and asynchronous web services invocation using non-blocking clients and transports and many other features (1). Some of the key features of Axis-2 are as follows: Reliable message, coordination, Security policy, Atomic transaction and Security.

1.2 JAX-WS

Axis 2 uses JAX-WS, in which a web service operation invocation is represented by an XML based protocol, such as SOAP. The SOAP specification defines the envelope structure, encoding rules, and conventions for representing web service invocations and responses. These calls and responses are transmitted as SOAP messages (XML files) over HTTP. Apache Axis-2 uses JAX-WS 2.0 which is a new programming model which simplifies application development through support of a standard annotation based model to develop web service applications and clients.

1.2.1 Web Service Description Language (WSDL):

WSDL is an XML based Interface Description Language that is used to describe the functionality offered by the web service. An interface description language is a specification language used to describe a Software component API. The purpose of WSDL is something similar to type signature in programming language. WSDL is in the form of XML language which specifies all the information about the server mapping, servlet, SOAP, XML schema and the information about the web service. The XML file gives all the information to the client when a client connects using SOAP. Also WSDL files are easy to be attacked since they are XML files. By limiting access to generating WSDL can avoid these attacks.

1.3 Apache Tomcat Server:

We have used apache tomcat server to host our web services. Apache tomcat is an open source java servlet container and it implements several Java EE specifications like Java Servlet, Java Server pages, Web socket etc. It consists of four main components: Catalina (Servlet container), Coyote (Connector component), Jasper (JSP engine), Cluster (load balancing). High availability feature has also been added to facilitate scheduling of updates without disturbing the processes. It also has user as well as system based web applications to add support for deployment.

2. DESIGN:

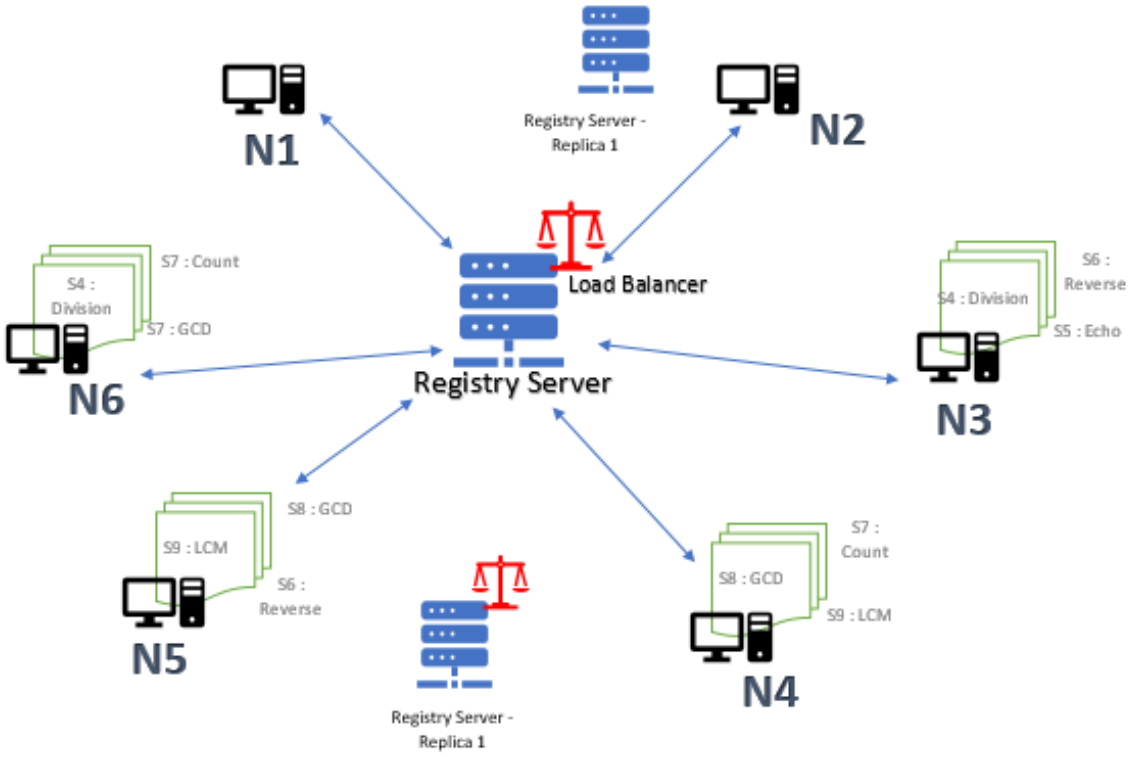

Figure:1 – Preliminary System Specification and design

The diagram above gives a high level view of how we chose to design our distributed system. Our "service registry" and Load Balancer would be deployed together as a web service together on a registry server. We have created two replicas of our service registry and Load Balancer on two different servers so as to remove the central point of failure. So we have three registry servers named: "Main Registry Server", "Registry server replica-1" and "Registry server replica-2".

We have our web services deployed on six different node servers such as Node-1 (N1), Node-6 (N6), etc. Each node server hosts three different web services where these services are deployed in a way that two different servers will have same service (which can be seen clearly in the diagram). This was to depict the efficiency of service availability and load balancing capabilities of our system on a minimal scale. These web services are built and deployed using Apache Axis-2 framework on Apache Tomcat Servers. These web services created in Axis-2 are JAX-WS web services which uses SOAP.

In our design we chose an approach which makes sure that if a node server is available then all its web services are available as well, one cannot start or stop an individual web service. Hence our node server only requires to announce a single message to get its hosted services registered on the registry servers. Whenever any node server is started it makes a POST request to all the three registry servers which results in sending a message (in JSON format) having the node server name, services hosted and the WSDL end point URL's on which these web services are exposed. If any of the registry server is down and not reachable a node server will try to announce itself periodically for a particular amount of time (For E.g., 10 times for every 20 seconds).

All the registry servers have several JAX-RS (javax.ws.rs) web services deployed using Glassfish Jersey web services dispatcher. A web service called 'RS_Listener' is exposed at '/hearing/post/register' on every registry server. Here we have used ConcurrentHashMaps Data structure as a mechanism for service registry. Once any registry server receives an announcement message i.e. on being notified about the node server start and the web services hosted by that node server, registry server (i.e. RS_Listener) will update its concurrent hash map, The screenshot below in figure.xx shows how our service registry hash map looks (Once all the node servers are registered).

```
{Addition=
{Node1=http://192.168.0.10:8081/Axis2_Node1/services/Addition_Service?wsdl,
Node2=http://192.168.0.10:8082/Axis2_Node2/services/Addition_Service?wsdl},

Multiplication=
{Node1=http://192.168.0.10:8081/Axis2_Node1/services/Multiplication_Service?wsdl,
Node2=http://192.168.0.10:8082/Axis2_Node2/services/Multiplication_Service?wsdl},

Subtraction=
{Node6=http://192.168.0.10:8086/Axis2_Node6/services/Subtraction_Service?wsdl,
Node2=http://192.168.0.10:8082/Axis2_Node2/services/Subtraction?wsdl},

Reverse=
{Node    5=http://192.168.0.10:8085/Axis2_Node5/services/Reverse_Service?wsdl,
Node3=http://192.168.0.10:8083/Axis2_Node3/services/Reverse_Service?wsdl},

Echo=
{Node1=http://192.168.0.10:8081/Axis2_Node1/services/Echo_Service?wsdl,
Node3=http://192.168.0.10:8083/Axis2_Node3/services/Echo_Service?wsdl},

LCM=
{Node5=http://192.168.0.10:8085/Axis2_Node5/services/LCM_Service?wsdl,
Node4=http://192.168.0.10:8084/Axis2_Node4/services/LCM_Service?wsdl},

GCD=
{Node5=http://192.168.0.10:8085/Axis2_Node5/services/GCD_Service?wsdl,
Node4=http://192.168.0.10:8084/Axis2_Node4/services/GCD_Service?wsdl},

Division=
{Node6=http://192.168.0.10:8086/Axis2_Node6/services/Division_Service?wsdl,
Node3=http://192.168.0.10:8083/Axis2_Node3/services/Division_Service?wsdl},

Count=
{Node6=http://192.168.0.10:8086/Axis2_Node6/services/Count_Service?wsdl,
Node4=http://192.168.0.10:8084/Axis2_Node4/services/Count_Service?wsdl}}
```

On our registry server an important web service called as "Services_Registery" exposed at '/services_registery/servicesenquiry_wsdl/{service_name}, a REST-GET method which takes the service name as a path argument /{service_name} from the client who requests for a WSDL of that service so as to consume the service exposed on that WSDL endpoint. Now whenever a client makes this request for the WSDL by providing the service name our Services_Registry first checks with the Load_Balancer (who checks the loads of all the node servers which host the

requested service (currently max two node servers) , which in turn retrieves the Node_Server which has lesser load as compared(If all the node servers are equally loaded then Load balancer will choose first node server in the list). Then the Registery_Server returns the WSDL of that service hosted on that particular node server (as returned by Load_Balancer) to client.

The load Balancer with each Regstery Server calculates a notional load and assigns to a Node_Server name in its Concurrent Hash map. (_Load_Balancer maintains another concurrentHashmap to store <NodeServerName : currentBalance>). For e.g. Say if "Addition" Service is availed on Node1, the LoadBalancer will increase the load value of that Node to (current_value+60). And will reduce the same load automatically after 20seconds (by spawning a new asynchronously run scheduler).

Whenever a Client is started it will first check if at least one of the three registery servers is up and running(if not client exits). Client will always prioritize the Main_Registery_Server over other replicas. Then a client program offers its user a list of all services (static list) to choose a service a user wishes to avail. On selecting a service, user requests for WSDL of that service from the Services_Registery web service by passing service name as path argument. Now if a NULL value is returned by the Registery_Server, client will request other replicas of the registery server for the wsdl (if they are available). The approach here is a failsafe measure whenever any of the registery servers get down and are started later but won't have the updated list of registered services by the Node_Servers(as these node servers will have sent them when they have started).

Whenever a Node_Server wants to shutdown gracefully, it will notify the 'destroy_listener' webservices of all the Registery Servers so that they can clean up their registries of the pertaining records.

But in case of abrupt failures of any nodes(which are already registered with the Registery Servers), there is a ping mechanism to check all the nodes, which is invoked each time a client requests for wsdl. So before querying the wsdl from hash map, Registery server will actually check if all the nodes in its registery are still up by pinging them and accordingly updating its hash map records, thereby providing only the most up to date results as a service discovery mechanism.

On getting a WSDL for a service, client program uses a Dynamic SOAP invocation mechanism to consume the webservice exposed on the WSDL endpoint. It parses the WSDL information present on wsdl endpoint and makes a SOAP call (similar to SOAP UI) and gives the result of the webservice to user.

3. Failures:

Cases in which this system will fail are:

1. When None of the Registery_Servers of the three are running but a client or Node_Servers are started.
2. When any of the Registery server is gracefully or abruptly shut down and restarts it won't have the updated hash map records as no Node_Server will be continuously notifying here, we could have made the Node_Servers keep notifying periodically (in asynchronous way)

but if would be expensive w.r.t. network traffic and probability of node failures, thus this is what we chose as a design for our system.

3. If any of the registery server comes to be alive after a considerable amount of time after all or any nodes are done notifying the registery servers, then this Registery Server will suffer from same problem as sated in above point. We could have solved these two issues by just making the newly started registery server to simply get the copy of the Services Registery hash maps, but the issue here was how will one know who has the latest hash map records? Again a lot of synchronization processes would have been involved which was left out for future scope to limit the scope of the project.
4. Same conditions are replicated with the load balancer Hash Maps stated in points 2 and 3.

4 . Assumptions:

Following are the assumptions considered while designing the system:

1 For best case scenario, all the Service_Registery Servers are up and running before any of the Node_Servers are up.

2 The URL's of the Registery servers are static and known to all Node_Servers which are part of the distributed system and Clients trying to avail services.

3 The ports on which the Servers are running are fixed (so as to demo them on the constrained amount of machines): Specified the ports so as to make sure if in case any of the servers are deployed on same machine the ports wont clash.
The ports are :

Registery Server (Main) : Port : 8080 // IP Address of the machine on which it is deployed
Registery Server (Replica 1) : Port : 8180 // IP Address of the machine on which it is deployed Registery Server (Replica 2) : Port : 8280 // IP Address of the machine on which it is deployed
Node1 : Port : 8081 //IP Address can be dynamic (No need to hardcode in program)
Node2 : Port : 8082 //IP Address can be dynamic (No need to hardcode in program)
Node3 : Port : 8083 //IP Address can be dynamic (No need to hardcode in program)
Node4 : Port : 8084 //IP Address can be dynamic (No need to hardcode in program)
Node5 : Port : 8085 //IP Address can be dynamic (No need to hardcode in program)
Node6 : Port : 8086 //IP Address can be dynamic (No need to hardcode in program)

4 The Node_server will be loaded by 60 notional units(by the Load_Balancer) whenever a service on it is availed by any client.

5 Any typical service will require 20sec time to finish its task(I have commented the Thread.sleep(1000*20); code in the program for the purpose of demonstration), thus the load balancer will reduce the notional load set on that Node by 60 units after 20 seconds.

6 Deployment is done in Local Area Network.

5. Testing Strategy:

Some of the major test cases were considered while testing the system. They are as follows:

5.1: Testing the state of the servers:

When the registry server performs the operation received from the client, it first checks which server contains the services. Then it request's the servers for a response. If both the servers are up and running it sends a connected message else it sends a message that the servers are not active.

Test Case	Testing the state of servers
Test input	Two servers which are down
Expected Result	Message that servers are currently unavailable
Result obtained	Servers are not active
Success/ Failure	Success

5.2: When client request for the service:

a) When the client sends a message to the registry server to perform addition which is present in Server-1 and Server-2, where server-1 is currently not running and only server-2 is active. In this case it should send the request to server-2 and the server-2 performs that operation and sends the result back to the registry server which later sends to the client.

Test Case	When only one server is active to perform the task
Test input	Check the server which has addition service and perform that operation.
Expected Result	Addition of two numbers should be sent back to the client.
Result obtained	Server-2 performs addition.
Success/ Failure	Success

b) In this case, when a client request's a particular operation, for example, the same addition service is present in both Server-1 and Server-2 and both the servers are up and running. In this case the registry server checks its Load Balancer to see which the least weight among both the nodes is and sends the request to that particular node which then processes the request and sends the result.

Test Case	When both the servers are active to perform the task
Test input	Check the server which has addition service and which has the least load among both the servers to perform that operation.(Consider Server-1)
Expected Result	Addition of two numbers should be sent back to the client.
Result obtained	Server-1 performs addition.
Success/ Failure	Success

c) When both the servers are running and the load balancer of both the servers are above the threshold value the registry server selects the first node which is present in the hash map and process the operation.

Test Case	When both the servers are active to perform the task and load balancer has met the threshold
Test input	Select the server which has addition service and which is present first in the hash map
Expected Result	Addition of two numbers should be sent back to the client.
Result obtained	Service is performed
Success/ Failure	Success

5.3: When any of the server is active, it tries to announce to all the registry server present:

As mentioned earlier, we have three registry server present in our system. When any of the server becomes active it tries to announce its presence to the registry server. Each server tries to send the announcement every 20 seconds and a maximum of 10 times.

a) In the first case if all the registry servers are active and when a server sends an announcement, all the registry servers listen to it and sends an acknowledgement.

Test Case	When all three registry servers are active
Test input	The server sends an announcement to all three registry servers
Expected Result	All the registry servers should get the announcement and send an acknowledgement back.
Result obtained	Acknowledgement received by the server
Success/ Failure	Success

b) In the second case if only one of the registry servers are active, and when the server tries to send the announcement to all the three registry servers, only one registry server

should send back an acknowledgement and the other two registry server should send no response.

Test Case	When only one registry server is active
Test input	Server sends an announcement to all three registry servers for the specified number of times
Expected Result	The server should get an acknowledgement from only one registry server.
Result obtained	Acknowledgement received from one registry server
Success/ Failure	Success

c) In the third case when none of the registry servers are active the server tries sending the announcement for the specified number of times. i.e. After every 20 seconds it sends a new announcement for 10 times.

Test Case	When none of the registry server is active
Test input	Server sends an announcement to all the three registry servers for the specified number of times
Expected Result	Server does not receive any acknowledgement after sending the announcement specified number of times
Result obtained	No acknowledgement from registry server
Success/ Failure	Success

5.4: When client tries to connect the registry server:

a) When the client program is running and if all the three registry servers are active it connects only to the first registry server, since the registry server-1 has the highest priority.

Test Case	When all three registry servers are active(or only the main registry server is active)
Test input	Client tries connect to the main registry server.
Expected Result	Client and the main registry server must be connected with one another
Result obtained	Connection is established between client and main registry servers.
Success/ Failure	Success

b) When any one of the registry server is active, depending on the priority the client checks one registry server after another. For example, if only registry server-2 and registry server-3 is active, the client first checks the main registry server followed by registry server-2 and gets

connected to registry server-2 leaving registry server-3. If only registry server-3 is active it connects to that.

Test Case	When both registry server-2 and registry server-3 are active but the main registry server is down
Test input	Client connect to the registry server-2
Expected Result	Checks for registry server-1 and connects with registry server-2 and leaves registry server-3.
Result obtained	Connection is established between client and registry server-2.
Success/ Failure	Success

Test Case	When only registry server-3 is active
Test input	Client sends an announcement to connect to the registry servers
Expected Result	Checks for registry server-1 followed by registry server-2 and finally connects with registry server-3
Result obtained	Connection is established between client and registry server-3.
Success/ Failure	Success

c) When none of the registry servers are active, the client checks if any one of the registry server is active, if not it does not connect to any of the registry servers.

Test Case	When none of the registry servers is active
Test input	Client tries to connect to the registry servers
Expected Result	Checks for registry servers and does not connect to any of the registry servers and the client program closes
Result obtained	Does not connect to any of the registry servers since none are alive and the client closes.
Success/ Failure	Success

6. Future Work/ Conclusion:

Mechanism to get the updated service registry records from other registry servers based on time stamp or other constructs to check who has the latest copy. Thus we have created a scalable distributed web service by considering all the important points and features to be present and designed it in the best scalable and efficient way possible.