ADVANCED OPERATING SYSTEM FINAL PROJECT

SongJun Park, Amruta Deshmukh, *CSEE UMBC*

Abstract: The paper describes our distributed web-services framework. Our system is capable for multiple clients and multiple servers on distributed physical machines. For service discovery, a centralized, but replicated registry was implemented in Python. The replica is designed to handle failures in the central registry. Load balancer was combined with the registry, which maintains the load assignment to multiple services consisting of Hello, Adder, and Subtractor. Load balancing assignment results show that the assignments of workloads are evenly distributed. The scalability analysis illustrates that of our system's performance was linear as the number of clients increased to 100 with the constant number of services.

***Index Terms* — Distributed system, web-services, client, server, load-balancer**

Introduction

For this project, we created a distributed web service framework consisting of Hello services, Adder services, and Subtract services. For service discovery, a centralized registry was implemented, which was replicated for fault tolerance. The load balancing was incorporated into the registry that distributes the workload based on essentially a round robin algorithm. Our design was tested on 3 different laptops where registry, services, and clients were separated in the laptops.

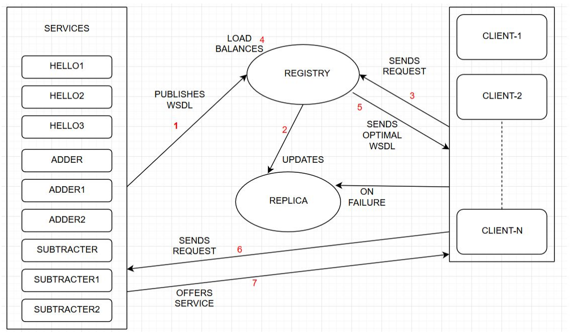
Assumptions

We assume that the IP addresses of the registry and the replica is known by all the clients and services beforehand. This needs to be hard coded in the source codes.

Our design of distributed web services assumes a stable networked servers as oppose to mobile devices operating in a mobile ad-hoc network. In other words, we are making an assumption that the network and thus communication is stable and reliable. Moreover, we assume that web services provided are changing infrequently and it is plausible that modifications to the available web services are a relatively rare event. Within this assumption, a centralized and replicated service registry for service discovery might suffice and offers design simplicity. The replication of the centralized registry would insure fault tolerance. A drawback of a replicated and centralized WSDL registry is maintaining consistency between replicas, which requires messages across network and delay in updating all the replicas to a common state. However, with a relaxed consistency between replicated registries, we believe that scalability and performance can be achieved.

System Design and Implementation Detail

For a distributed web services design, we developed an architecture consisting of a centralized service registry for service discovery and load balancing. To address fault tolerance, this registry was replicated to mitigate a single point of failure. When the services publish their WSDLs, the replicated registry also receives the same WSDL publication. For the clients, before connecting to the registry, it pings the registry, and if no response is received, then the client reaches out to the replica registry. Centralized design was selected due to simplicity of design and to minimize communication costs. Decentralized registry would incur higher communication for load balancing and the network transmission costs can dominate depending on the speed of service computation.



In the beginning, all service providers send its WSDL location to the registry. This WSDL publication process is completed by Python codes, which parses the WSDL file in the webapps/ folder of a Tomcat server, then extracts the WSDL location XML tag and sends a WSDL location as a message to the registry after prefixing the message with a service ID such as Adder1.

To locate a particular service, a client will request for a particular service to a round robin load balancer within the registry. The load balancer keep track of assignments and chooses the service that recieved the lowest number of assignments. This optimal WSDL location is returned as a message to the requesting client. WSDL location messages and load assignments are managed in a dictionary data structure.

The design implements a SOAP-based web services and were created using Eclipse IDE for Java EE Developers. Eclipse tool generated WSDL files and exported '.war' packages for deployment to a server. Tomcat 9 server was utilized for our web services deployment consisting of Hello, Adder, and Subtractor.

As for the implementation of the discovery registry/load balancer, Python framework was selected to ease the programming development. The discovery registry involved creating a multithreaded socket communication server in Python. The registry listens on a socket for WSDL publication, which would be sent by the services in the beginning. Note that when a WSDL location is received, then it is forwarded to the replica. The registry also listens for requests from the clients. These messages will being with "Request" followed by the type of service desired by the clients such as "Hello", "Adder", or "Subtract".

Replica registry is a copy to the registry where the WSDL location message would be forwarded by the registry. Essentially it is keeping a copy of the WSDLs and load assignments.

As for the SOAP clients in Python, Zeep was leveraged, which scans the WSDL and generates a programmatic interface to a SOAP server. A client will first check if the registry is operational, if not, the replica is contacted for the request. The request message will contain a "Request" string and the service it wants. The registry will return a message to the client with an optimal WSDL location. Once the WSDL location is received, the client initiates the connection with the service provider using Zeep. Here, the method function of the service operations are assumed to be known in advance by the clients. For scalability, a looping function of the clients is created as a Python file and the default loop is set to 100.

The communication steps are described in the following:

- Send WSDL to the registry
- Client request a service
- Registry returns an optimal WSDL
- Client contacts the service
- The service is offered

Code Execution Instruction Steps

Our design was tested on two Windows machines and one Mac system. The software leveraged are Eclipse Java EE Developers, Tomcat 9, Python 2.7, Zeep (Python SOAP client).

1. Download the zip file and extract.
2. Run your Tomcat server before transferring files.
3. The prerequisite for running starting Tomcat is that the JRE home and Java home environments be set.If the variables are not set, please follow this link for the same.
4. You can start the Tomcat by navigating to the Tomcat source location -> bin -> startup.bat or startup.sh
5. Move all '*.war' files under the Tomcat 'webapps/' directory. This should generate folders corresponding to the war files.
6. Move all files starting with 'send_wsdl*' to the 'webapps/' directory
7. Move 'publish-hello-add-sub.py' script file to the 'webapps/' directory
8. Hard code replica's IP address in registry2.7.py file (available at the zip location), which is the first line with variable name 'replica_hard'.To run everything locally,replace the IP with your own IP address.
9. Hard code registry's IP address in all the send_wsdl*.py files (now located webapps directory in of Tomcat), which is the second line with variable name 'registry_hard' (there are 9 send_wsdl*.py files)
10. For the three client files hello_client.py, adder_client.py, subtracter_client.py (located at the zip folder), hard code registry and replica IP addresses, which are located in the 1st and 2nd lines with

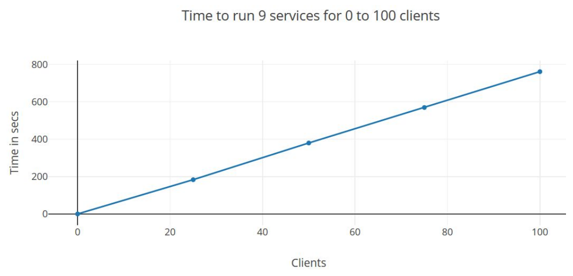
variables, 'registry_hard' and 'replica_hard' respectively.

11. Next step is to run your registry, navigate to the folder location of registry (your zip location from terminal or Windows Powershell).
12. Run the registry on machine 1:
    $ python registry2.7.py
    DO NOT CLOSE THIS TAB
13. Next step is to run your replica, navigate to the folder location of replica (your zip location from terminal).
14. Run the replica on machine 2 (replica only works if it is running on a different physical machine than the registry) by using the following command:
    $ python replica.py
15. Send WSDL files from Tomcat webapps/ directory on machine 2:
    $ python publish-hello-add-sub.py
    If on Windows machine, please make sure the port mentioned in registry is open and allows connection.
16. After executing the above command, you will see WSDLs being published to the registry
17. Run clients from machine 3:
    $ python hello_client.py
    $ python adder_client.py
    $ python subtracter_client.py
18. To test for multiple clients run:
    $ python client_loop_hello.py

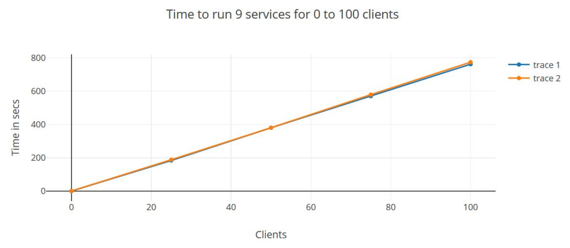
Scalability Analysis and Results

Load balancing tests indicate that the load assignment follows the evenly distributed workload assignments for each services. This implies the round robin load assignment is working correctly as designed.

The system was tested for different client loads. The performance of the system was evaluated for 1, 25, 50, 75 and 100 clients with services to be offered being 9. The performance is depicted as shown in the graph below. The graph indicates the linear scaling in execution time as a function of clients. For a constant 9 services, as the number of clients increased the response time of the system increased linearly.



The following figure depicts the runs of 100 clients for 9 services for a set of two. This shows that the system is stable and does not show much difference and is thus consistent.



We tested fault tolerance by purposefully shutting down the machine that is running the registry. It can be observed that the replica assumes the responsibility of client requests when the central registry cannot be reached by the clients.

Project Work Division

Amruta worked on the multiple services and multiple clients. Songjun worked on the registry/load balancing, and replica.

Conclusion

Our implementation is designed to work as a distributed web-service framework consisting of 9 different web services, where optimal load balancing assignment distributes the workload across multiple web services. Multithreaded registry is able to serve multiple clients. Replication of the registry was implemented to address fault tolerance. Our design showed a linear scalability when the 9 services were tested for 1 to 100 clients.

References

1. https://www.eclipse.org/webtools/community/education/web/t320.php
2. Guides to install Tomcat,eclipse, Python