CS 549—Fall 2016
Distributed Systems and Cloud Computing
Assignment Four—Websockets

In an earlier assignment, you developed a simple peer-to-peer distributed hash table for sharing bindings of keys to values. Each node in the network runs a CLI "shell" that allows bindings to be added and queried in the network. In this assignment, we will add the ability for one node's shell to remotely control another node. The communication to support this will be performed using Websockets, using a very simple protocol where command lines are sent from a controlling client to a controlled server, and the server sends responses back to the client. Sending of command lines and of responses is asynchronous: The "proxy shell" on the client just copies lines of input from the console to the server, and displays lines of response from the server on the console. The prompt that the controlling client sees in the user interface is just a string sent back to it to be displayed on the console.

This assignment introduces three new operations to the CLI:

1. `connect` *`host port`*: Make a connection request to the node identified by the host name and TCP port number. On the remote node, if a control session is not already pending or in progress, then a new pending control session is registered, and the user at that node is notified that there is a pending control request. The user at the remote node uses one of the two following commands to respond.
2. `reject`: Reject the pending connection request. The connection to the client is closed, with an explanation that the control request has been terminated, and the pending control session is removed at the server.
3. `accept`: Accept the pending connection request. An acknowledgement (the string "ACK") is sent back to the client, to let them know that their control request has been approved.
4. Normal termination of the protocol happens when the "`quit`" command is issued on the client. The client sends the "`quit`" command to the server and exits the CLI for its proxy shell. When it receives the "`quit`" command, the server closes the connection and exits the CLI for its local shell instance for the client.

There is one Web service endpoint exposed by this protocol, for obtaining a Websocket connection to remotely control a node:

`ws://`*`hostname`*`:`*`port`*`/dht/control/`*`client-name`*

Here `client-name` is a meaningful name chosen by the controlling node to identify themselves to a node when they make a remote control request. This name is displayed in the notification message displayed to the user when such a request is received. Also, since the Jersey and Tyrus frameworks that we use for this assignment are not integrated, we use separate Web server instances for each of them, so the Websocket port number above should be different from the HTTP port number used for DHT Web service calls.

To run a node in this network, you type:

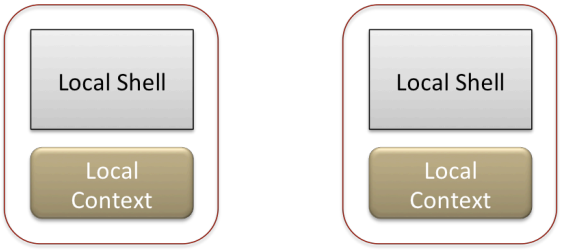
```
java –jar dht --host host --http http-port --ws ws-port --name name --id key
```

Note that you do not have to join nodes into a network for this assignment, and you are better off not trying to do so. Treat each node as a standalone network.

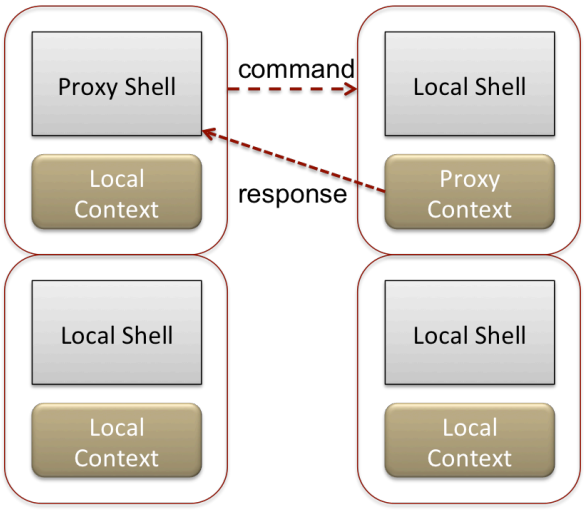
There are two important abstractions for this assignment:

1. A **shell** that accepts command lines and processes these, producing output. A *local shell* executes commands at the local node. A *proxy shell* executes commands (except those to do with remote control) on a remote node that has agreed to be remotely controlled.
2. A **context**, underlying a shell, that provides an operation for reading a line of input, and various operations for writing responses and reporting errors. A *local context* reads from standard input and writes to standard output. A *proxy context* reads from a command line buffer that it maintains internally, synchronizing with the insertion of command lines into that buffer, and sends responses over a Websocket connection to a controlling client.

Typically, a local shell runs on top of a local context:

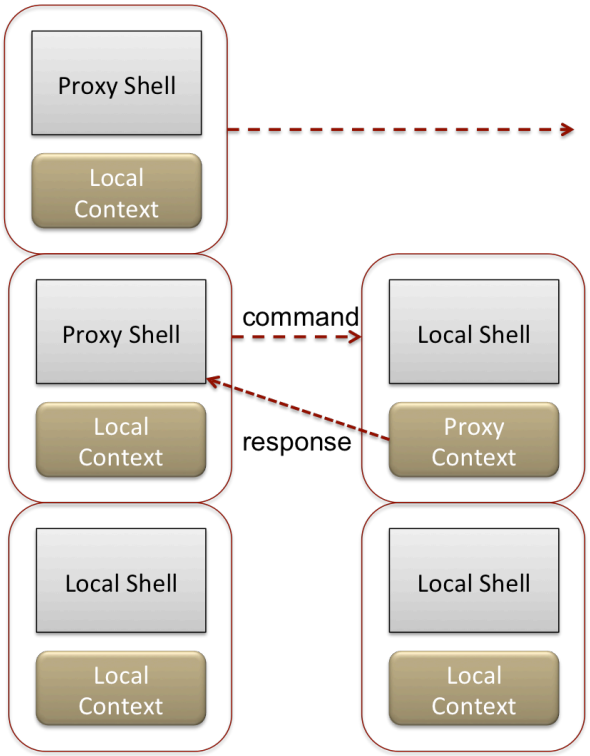


Each node maintains a stack of shells, with the currently executing shell on top of the stack. When a client takes control of a server, then a proxy shell, with the same local context as before, is pushed onto the shell stack. On the server, a local shell, with a proxy context that communicates with the controlling shell, is pushed onto its shell stack:



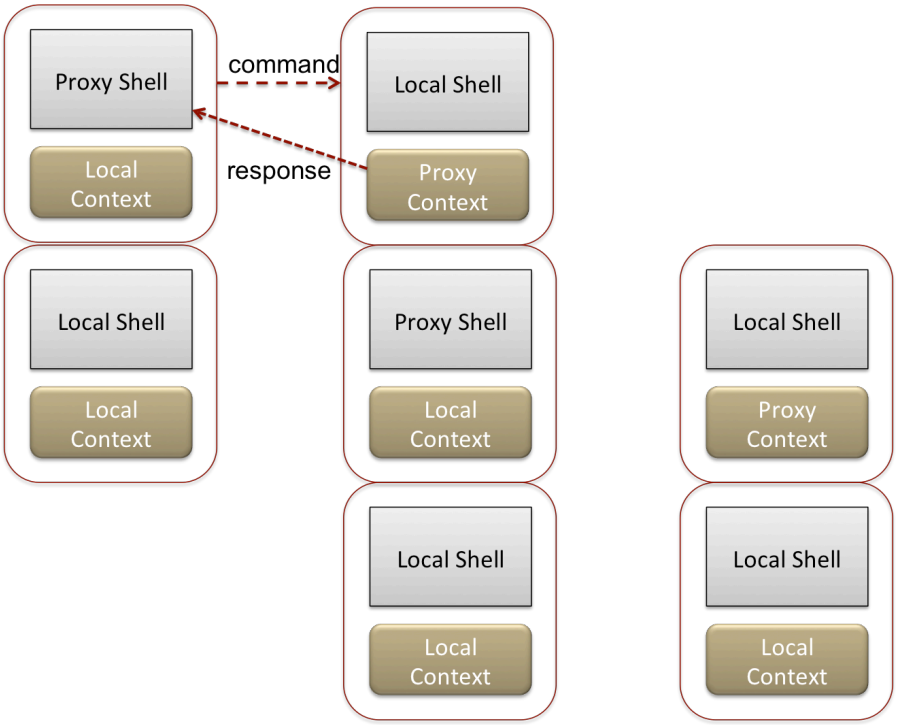
The proxy shell on the controlling client forwards commands to the remotely controlled server, which executes the commands locally. The commands are buffered at the proxy context, and read by the CLI in the local shell. Responses are sent back to the controlling shell using the proxy context. When the control session ends, both the client and the server pop their shell stacks to restore the previously executing shells. In the protocol that we use, the server closes the connection when the session is finished, although both client and server should be able to react to the connection being closed without an explicit quit command.

Although most commands are relayed by a proxy shell to the remotely controlled node, the commands for establishing and shutting down a connection (connect, accept, reject and quit) are executed locally. Consider for example if we execute a connect operation while we are currently remotely controlling another node (as above), then we obtain the following configuration:



In this scenario, at the controlling client, both its original local shell and the first proxy shell are suspended, while the client remotely controls a second node with the topmost proxy shell on the shell stack. When that remote control session terminates, the shell stack is popped and the execution of the first proxy shell resumes, and resumes sending commands to the remote node (which remains suspended throughout, until the first remote control session is terminated).

Another scenario to consider is where a node (such as the leftmost node above) receives a remote control request while it is engaged in controlling another node. If the node responds by accepting the remote control request, then a new local shell must be pushed on the shell stack, with a proxy context that communicates with the third, controlling node:



There is no scenario under which we might have a proxy shell running on a proxy context. A proxy shell always interacts with the local console for input/output.

For the code that you are provided with, there are several abstractions provided to support this:

1. `LocalShell` and `ProxyShell`, both extending `ShellBase` and implementing `IShell`.
2. `LocalContext` and `ProxyContext`, both extending `ContextBase` and implementing `IContext`.
3. `SessionManager` (which uses the Singleton pattern) handles remote control sessions. It maintains a buffer of size one for active or pending sessions, and provides logic for closing the current session, as well as rejecting and accepting a remote control request. In the latter case, the session manager pushes a new (local) shell on the shell stack, to be controlled by the remote client.
4. `ShellManager` (which also uses the Singleton pattern) maintains the stack of currently active shells. While the CLI for the topmost shell is being executed, those for the shells below it in the stack are suspended.
5. `ControllerClient` is used by the client to initiate a remote control session, and it is used by the client to respond to communication from the controlled node. The controller opens a Websocket connection to the remote node, and uses a programmatically defined endpoint to interact with it. A special case must be made for the acknowledgement sent by the remote node to accept a remote control request; the controller should install a new proxy shell on the shell stack when this happens.
6. `ControllerServer` is an annotation-defined callback class for responding to remotely controlling clients. It is responsible for notifying the user when a remote control request has been received, for responding to messages received from the client, and for handling closure or errors on the underlying connection.

Once you have your code running locally, you should test it on some EC2 instances that you define. Copy the server jar file to the EC2 instance using "`scp`," using the "`-i`" option to define

the private key file that you use to authenticate yourself to the instances. Then ssh to the instance and run the server script. It should be in a directory of the form

```
/home/ec2-user/tmp/cs549/dht-test
```

that you should have created. Now you should be able to run network nodes on several EC2 instances and have them communicate with each other.

Once you have your code working, please follow these instructions for submitting your assignment:

- Export your Eclipse project to the file system.
- Create a zip archive file, named after you, containing a directory with your name. E.g. if your name is Humphrey Bogart, then name the directory `Humphrey_Bogart`.
- In that directory you should provide the zip archive that contains your sources, and the server jar file, `dht.jar`.
- Also include in the directory a report of your submission. This report should be in PDF format. **Do not provide a Word document.**
- Your submission should include videos demonstrating testing of your assignment, as described below. Your name should appear on the videos that you provide.

The content of the report is very important. A missing or sloppy report will hurt your final grade, even if you get everything working. In this report, describe how you completed the code to obtain a working implementation. You should include critical code snippets with explanation, with a focus on the REST API and the business logic for the DHT. You should also describe how you tested the code. Again, it is very important for your grade that you do adequate testing. You should at a minimum demonstrate testing of these scenarios with the videos that you provide:

1. You should demonstrate your application working with (at least) each of the scenarios described above, including adding and querying bindings both when operating locally and remotely.
2. Record one or more short videos demonstrating your tool working. Upload these videos with your submission.
3. Display the state of your nodes at the beginning and end of the demonstration, particularly local bindings.
4. Make sure that your name appears at the beginning of each video. For example, display the contents of a file that provides your name.

Finally note any other changes you made to the materials provided, with an explanation.

Remember the format of the submission: A zip archive file, named after you, with a directory named after you. In this directory, provide these files: a zip archive of your source files and resources, a server jar file, a report for your submission, and videos demonstrating your app working. Failure to follow these submission instructions will adversely affect your grade, perhaps to a large extent.