Automated Testing Engine

CUAHSI Inc.

Prepared by Neal DeBuhr

January 26, 2018

1 Infrastructure

While continuous integration and continuous delivery workflows are the ideal situation for CUAHSI software systems, intermediate steps must be taken as the team builds up the necessary infrastructure and knowledge. The CUAHSI Automated Testing Engine seeks to support future CI/CD needs, while being reliant only on existing infrastructure and staffing resources. The engine starts with Jenkins, which is used as a “cron on steroids”. Future adaptation to a Jenkins CI/CD workflow is not trivial, but is still relatively simple. The Jenkins system is used for test initiation, results processing, and historical test results data analysis. The Jenkins system communicates with the Selenium Grid hub over the network, which in turn communicates with a network of Selenium Grid nodes. The Selenium Grid hub and Selenium Grid nodes all run on independent virtual machines. When a node is available, a single test case is sent to the node for execution. Upon test completion, the results are funnelled back through the hub to the Jenkins job which initiated the test.

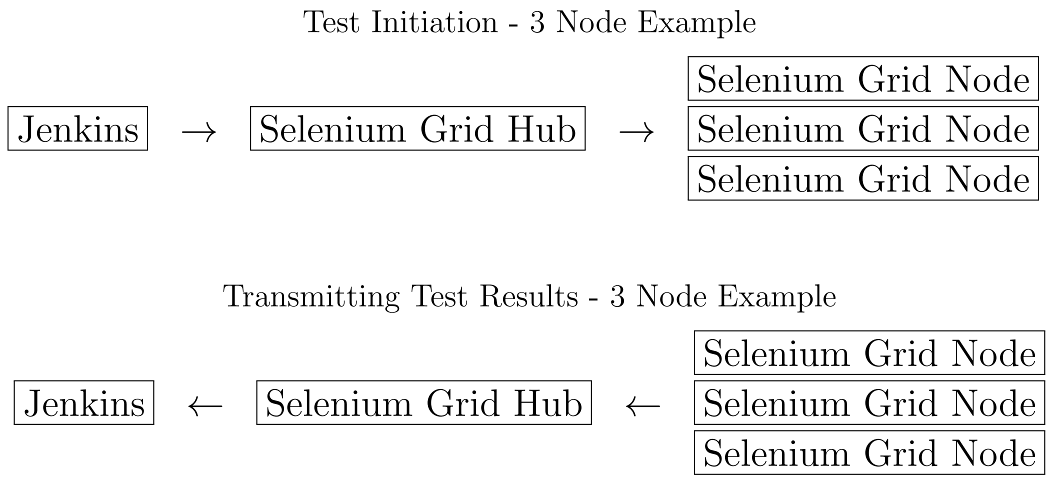

Figure 1: Infrastructure - Network Communication

1.1 Jenkins

While scheduled execution and manual execution serve as the baseline trigger mechanisms for the Jenkins system, more sophisticated trigger mechanisms can easily be added, such as GitHub-based triggers. Upon triggering the “core” Jenkins job for a specific test suite, the public cuahsi-qa-automation-engine repository is pulled from GitHub. Reading the associated test suite configuration file determines which tests should be executed in the current batch of automated tests. Using a configuration file which is separate from the test suite itself enables test scripts to be developed, fixed, or improved within the repository, without the pressure of a pending scheduled execution. After gathering information on which tests to run, the core Jenkins job then makes a series of API calls in order to:

1. Create a Jenkins job for any testcases which do not already have an established Jenkins job. A testcase job template is used to drive the configuration of the new job.
2. Initiate the necessary testcase jobs.

Jobs will be queued within Jenkins if the number of testcases exceeds the number of available executors. The executors are not running the tests themselves and have negligible resource requirements. As such, the number of executors can be set relatively high. Each testcase job communicates over TCP port 4444 to the Selenium Grid hub for initiating tests and receiving results.
The core job, the testcase template job, and the testcase jobs are all grouped within a Jenkins view. This view uses RegExp to dynamically group the tests. This approach to Jenkins view creation removes the need to manually categorize, configure, or otherwise set up new testcase jobs. In fact, creating the testcase script within the GitHub repository and adding it to the configuration file is all that is necessary to bring the testcase into operation. In the unlikely event that a testcase job needs to be removed from Jenkins, that would need to be done within the GUI or by a one-off API call. Testcases can be pulled from operation simply by removing the testcase from the configuration file, so normal use should not involve deleting Jenkins testcase jobs. Testcase numbers need to be carefully maintained. Reuse or renumbering will cause problems for historical results analysis, while duplicate

numbering will result in tests not being ran correctly. The testcase number should be thought of as a database primary key in this system.
Upon the completion of each test case, the results will be reported in standard Jenkins fashion. The Jenkins view for the specific test suite will serve as a nice dashboard for understanding the overall health of a CUAHSI software system. Improving Jenkins reporting mechanisms will be a subject of further development. Ideally, test execution reports are clean, user-friendly, and automatically delivered in an appropriate manner.

1.2 Selenium Grid

The Selenium Grid hub runs on a virtual machine within the CUAHSI development infrastructure. The hub is an Ubuntu machine which, upon startup:

1. Automatically login to cuahsi user, using systemctl graphical-target default
2. Start up a Java Runtime Environment and load the Selenium Server jar file
3. Acknowledge any registration requests from Selenium Grid nodes

The Selenium Grid nodes each run on a virtual machine within the CUAHSI development infrastructure. The Selenium Grid nodes can and should use varying operating systems and browsers. A wide variety of node configurations will facilitate the discovery of configuration-related bugs. Additionally, all the nodes should take minimal work to start up and enable for test execution. Automatic login and JRE execution are handled differently for different operating systems. For existing Ubuntu Linux nodes, the steps include:

1. Automatically login to cuahsi user, using systemctl graphical-target default
2. Start up a Java Runtime Environment and load the Selenium Server jar file
3. Register with the Selenium Grid hub - registration requests will go out on port 5555 every few seconds until acknowledged

Virtual machines were chosen, as opposed to Docker images or Jenkins environments, because are more representative of real user environments. Ideally, of course, the Selenium Grid nodes are distributed across the world on a wide variety of consumer-grade hardware. The virtual machines approach seems to strike the right balance between test environment validity and efficiency. Alternative test environment approaches can be explored in future work. Given the modularity of the system, migration to an alternative test environment approach would not be especially difficult.
Screen capture videos have traditionally been helpful to the CUAHSI team for understanding bugs and undesirable behavior identified through QA. To extend this practice from manual testing to automated testing, the Selenium Grid nodes have been outfitted with a video capture system. All running Ubuntu Linux nodes will continually check the system processes for signs of test execution (e.g. a geckodriver process). Next, the recordmydesktop tool is automatically launched by bash script to capture the screen while the test runs. When the test has finished (e.g. no geckodriver process), the bash script executes a gentle process kill. Upon receiving the kill request, recordmydesktop stops recording and starts encoding. The system process ends automatically after encoding. To enable these video captures and to more closely simulate real user activity, the CUAHSI Automated Testing Engine does not utilize headless test execution.
Video files are named automatically using epoc timestamps. Future development will involve a more direct approach to testcase video identification - something better than manually matching up timestamps. While the system currently establishes a 1080p screen for test execution, the xrandr tool can be used to change the screen resolution for Linux test environments. The test environment "screen" here is not an actual display or output, but rather an abstract object that handles pre-output graphical content. The Selenium Grid nodes do not have any physical display.

2 Test Suite

2.1 Abstraction Layers

$$\begin{tikzcd}
\boxed{\text{Test Case Scripts}} \arrow[d] & \text{CUAHSI Test Cases} & \textit{Minimum Complexity} \arrow[ddd] \\
\boxed{\text{Workflow Layer}} \arrow[d] & \text{CUAHSI High-Level Framework} & \\
\boxed{\text{Site Element Layer}} \arrow[d] & \text{CUAHSI Low-Level Framework} & \\
\boxed{\text{Selenium}} \arrow[d, shift right] & \text{Off-the-shelf Selenium} & \textit{Maximum Complexity} \\
\text{Software Product} \arrow[u, shift right] & \text{CUAHSI Software Product} &
\end{tikzcd}$$

Figure 2: Test Suite Framework

Important Notes:

- Whenever possible, complexity should be pushed down the layers (closer to the Selenium layer)
- The test cases themselves (top level) should in particular be very low complexity. Ideally, business users should be able to easily understand and perhaps even write these scripts
- Each level should only interact with one level below it (e.g. the workflow layer should not have Selenium commands)

2.2 Layers Implementation

For the HydroClient system, the CUAHSI Test Suite Framework approach is implemented using the following layers (with module names noted). While this is specific to HydroClient, other CUAHSI systems follow the same general approach.

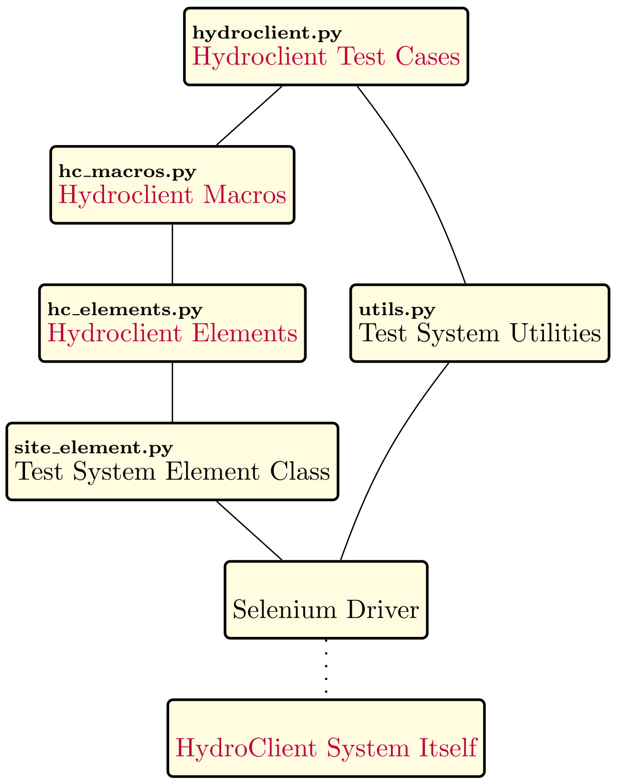

Figure 3: HydroClient Test Suite Framework