Continuous Integration System Structure

Jacob Sauer
7 March 2021

Files in the Project

A typical continuous integration system will contain three Python files for each of the three aforementioned components; *repo_observer.py* contains code for the repository observer, *dispatcher.py* contains code for the test job dispatcher, and *test_runner.py* contains code for the test runner. Next, the *helpers.py* files contains code used to transmit information via sockets. This code is utilized by all three components, which import the process of communication from the aforementioned file. Thirdly, a bash script exists for each process in the repository; such scripts enable straightforward invocation of bash and git commands. Finally, a test directory contains all tests that will be executed by the CI system.

Repository Setup

CI systems conduct tests by automatically detecting changes to repository code. Therefore, the first component to create in the continuous integration system is the master repository. In our group's previous report on the Git version control framework, we discussed the benefits of multi-branching, wherein developers can safely modify code and ensure that it is functional prior to pushing these changes to a master branch. The master repository in a CI system fulfills a similar purpose, as it is the destination of the developers' completed code. The CI system periodically pulls the contents of the master repository and examines these contents for changes. If any change is detected, then tests must be executed.

Next, a commit must be pushed to the master repository in order to verify that this functionality is present. If the CI system has been configured correctly, then the repository observer will observe this commit. The repository observer also requires a clone of the code, which means that a duplicate of the master repository must be created using the *git clone* command. Finally, *git clone* must be used a second time in order to clone the master repository code for the test runner.

Repository Observer

The repository observer monitors a particular repository for new commits. If it ascertains that the repository has been modified, it instructs the dispatcher to run tests against a particular commit ID. The observer checks for commits by storing the repository's current commit ID, updating the repository, and comparing the new commit ID with the previous commit ID.

The observer is assigned a particular repository to monitor through the *repo_observer.py* file, as this file is invoked with a specified path. In addition, the observer is provided the address of the dispatcher via the *--dispatcher-server* command.

When the repository observer's Python script is invoked, it begins a polling function. Said function parses command-line arguments and initiates an infinite while loop, which periodically checks the repository for changes. In the while loop, a bash script named *update_repo.sh* is invoked; this script identifies new commits and replays them to the observer. If a new commit is detected, then its ID is stored in a file called *.commit_id*. As mentioned previously, if the commit ID stored in *.commit_id* does not match the previous ID, then the dispatcher must be notified so that tests on the newly committed code can commence. The repository observer checks the dispatcher's server status by connecting to it and sending a status request; a response of "OK" from the dispatcher prompts transmission the latest commit.

Noe that the *.commit_id* file remains in the master repository when contact is established between the repository observer and the dispatcher. Since *update_repo.sh* script is called infinitely, this file may be erroneously sent to the dispatcher if it contains a commit ID that was already tested. consequently, the bash script removes this file in each iteration of the while loop, and only recreates it if a new commit is observed.

Dispatcher

The dispatcher essentially acts as a mediator between the repository observer and the test runners. The dispatcher enables test runners to register themselves, and when it receives a commit ID from the repository observer, it triggers a test runner to evaluate the commit. In addition, the dispatcher can redistribute the commit ID to a different test runner if any issues are encountered.

When the dispatcher's Python script is executed, a serve function is called, which parses the arguments enabling a user to specify the dispatcher's host and port. This activates the dispatcher server, as well as two threads; the first thread executed a *runner_checker* function, while the other executes a *redistribute* function. The *runner_checker* function periodically contacts each registered test runner to ensure that they are still responsive. If a test runner fails to respond, then it is removed from the dispatcher's array of valid test runners, and its commit ID is dispatched to the next available test runner. Likewise, the *redistribute* function dispatches all pending commit IDs that have not yet been dispatched a test runner.

The dispatcher server uses the SocketServer module, which affords four basic server types: transmission control protocol (TCP), user datagram protocol (UDP), UnixStreamServer, and UnixDatagramServer. By default, a TCP-based socket server is used in CI systems, as TCP enables continuous streams of data between servers, while UDP does not. SocketServer's native TCP server is not ideal for the dispatcher, as it can only process one request at a time. Consequently, if two objects attempt to establish contact with the dispatcher in rapid succession, the second object would need to wait for the first object's connection to be completed before being serviced. The dispatcher server bypasses this limitation with the ThreadingTCPServer custom class, which enables threading in the SocketServer module. Threading causes the dispatcher to initiate a new and exclusive process for each connection, and run these processes in parallel. As a result, the dispatcher can process multiple requests simultaneously.

Using the DispatcherHandler class, the dispatcher server defines a handler function for each request. This handler function examines an incoming connection request and parses the command that is being requested; it is invoked each time a connection to the dispatcher is requested. Handler functions can process four commands, which are detailed below:

1. *status*: Checks if the dispatcher server is currently active.
2. *register*: When called on a host:port pair, it stores the test runner's information in a list in order to enable future communication.
3. *dispatch*: Dispatches a test runner against a commit. The dispatcher parses the commit ID from a message and transmits it to a particular test runner.
4. *results*: Used by a test runner to report the results of a completed test. The commit ID identifies the commit against which the tests were run.

<u>Test Runner</u>

The test runner runs tests against a given commit ID and reports the results of said tests. Test runners never communicate with the repository observer; instead, they are provided instructions by the dispatcher server when the dispatcher receives a commit ID from the repository observer.

The *test_runner* Python Script invokes a serve function that activates the test runner server, and initializes a thread to run a *dispatcher_checker* function. The *dispatcher_checker* function pings the dispatcher ever five seconds in order to ensure that it is still active. If the dispatcher is deactivated, then all test runners will deactivate as well, as they cannot perform any meaningful functionality if there is no source to provide instructions to them.

Similar to the dispatcher server, test runners must be constructed as ThreadingTCPServers because the dispatcher must periodically verify that each test runner is still in operation. The flow of communication begins with the dispatcher requesting that a test runner perform tests on a particular commit ID, which it receives from the repository observer. The test runner responds affirmatively if it is ready to perform a task, which prompts the dispatcher server to close this connection. Subsequently, the test runner opens a new thread and begins the requested tests. Each ping from the dispatcher server to the test runner following this initial assignment is transmitted on a separate thread from the one on which tests are occurring.

The test runner server responds to two messages from the dispatcher: *ping* and *runtest*. *ping* verifies that the test runner is still active, while *runtest* instructs the test runner to perform tests on a provided commit. The test runner will respond to the latter message with "OK" if it is available for instruction, and with "BUSY" if it is already running tests on a previous commit. When a test runner accepts new instructions, the shell script *test_ runner_ script.sh* is invoked; *test_ runner_ script.sh* updates the test runner's accessible repository to the new commit ID. If a change in the stored commit ID is observed, then unit tests are performed, and their results are stored in a file. Upon completion of these tests, the test runner relays the results to the dispatcher.

As previously mentioned, the test runner requires a clone of the repository against which tests are run. The repository that was created with the second *git clone* command fulfills this requirement. By default, the test runner's Python script attempts to connect to the dispatcher server at port 8888 of *localhost*, but both values can be overridden. The --*host* and --*port* arguments designate a specific address for the test runner server, while the --*dispatcher-server* argument specifies the dispatcher's address.