ENSE 375 Report

Li Pan
11 March 2021

Project Limitations and Notes

Continuous integration works on Git or Mercurial version control system repositories. For example, if a developer pushes a commit to the staging repository, the continuous integration system runs a pipeline to build and test it, and the test has to be passed before the commit is merged into the main branch. Continuous integration systems monitor the master repository instead of each local repository. The master repository is usually hosted on a web server.

Continuous integration systems should not be run on a fixed schedule; instead, they should be run after every one or two commits. If a CI system is set up to run periodically, then it will only the most recent commits made after a set time period, rather than testing all the commits made within the entire time period.

The design goal of continuous integration is to check the changes in the repository periodically. In a real situation, the CI system will have a repository observer, which is noticed from the host repository. For example, Github provides post-commit hooks, which send out notifications to a URL. The repository observer will be called by the master host responding to the notification.

CI systems also include report functions. The test runner reports the outcome of each test result and either posts it to a webpage or collects all information saved into a system file. Additionally, a CI system has many architectural components. The system outlined in the text contains three components: the observer, test dispatcher, and test runner(s).

Introduction

The continuous integration system consists of an observer, a test job dispatcher, and one to many test runners. The responsibility of the observer is to watch each repository. If a commit is made, the observer informs the job dispatcher; then, the job dispatcher finds a test runner and assigns a commit test to it.

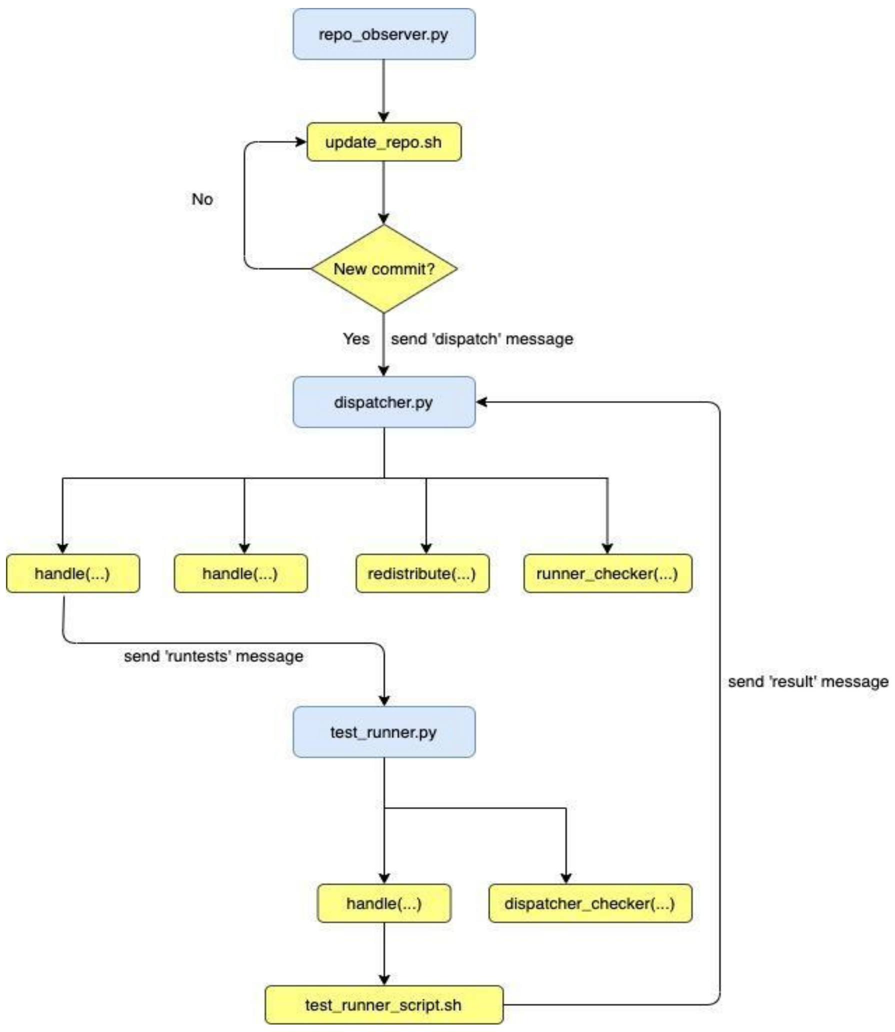
There are many architectural approaches to continuous integration, as mentioned in the previous section. One such approach is to keep an observer, a test job dispatcher, and a test runner on a single machine. This approach has a limited load bearing because it does not contain load handling. If the amount of changes made is more than what the CI system can handle, there will be a backlog. Additionally, this approach is not fault-tolerant. For example, if a computer has a power outage, there is no backup system that can keep running tests. Thus, the ideal system should be able to handle as many test jobs as necessary, and contain backups if the machine goes down.

If each component has its own process, a CI system will achieve load-bearing and fault-tolerance. Likewise, each process is independent of each other, and multiple processes can run simultaneously. This allows several test runners to run in parallel without a queued backlog test.

The architectural approach analysed in the following report is predicated on separated processes and communication sockets. Using a communication socket allows each process to work on separated and networked machines. Each unique host, or post address, is assigned to each component. Processes can then communicate by posting messages at the assigned addresses. This approach can handle hardware failures by the distributed architecture. The observer, dispatcher, and test runner can all run on different machines at the same time, and communicate with each other within a network. If one machine falters, there will be a new machine that will be a backup machine on the network. This guarantees that the overall system will not fail.

The following discussed project excludes auto-recovery code. The different distributed systems have different features. In a real situation, the CI system runs in a distributed environment so that the system always has a backup machine if one machine goes down.

Control Flow Diagram



Error Handling

Continuous integration system contains handlings for simple errors. For example, the *dispatcher.py* script can detect which processes are killed in *test_runner.py* and remove them from the process pool. Usually, a test runner is killed by a machine crash or network failure. If a test runner is killed, the dispatcher will enable another test runner that is available in the work pool, or wait for a new test runner to come into the pool. If the dispatcher is killed, the repository can catch it and throw an exception. All test runners will shut down automatically as a result.