Cider - An Event-driven Continuous Integration Server

Introduction

When implementing a custom development workflow, it happens often that multiple development tools and services need to be integrated. In this scenario it is not only necessary for the services to fulfill their respective roles, they must be able to communicate with each other as well.

CI Server

Our goal is to design a lightweight, modular CI server that is more flexible than the existing ones.

The CI server can be divided into two parts:
1) CI server is not the only part in the system. It needs to communicate with other systems. There is one input side and the output side of the communication. The system triggers a build when a code change is detected and it sends out notifications when the build is finished. This forms the communication subsystem.
2) The main purpose of every CI server is to build and test changes that are to be incorporated into the project, so naturally the other subsystem, the build subsystem, takes care of the build execution step.

The Communication Subsystem

When integrating multiple systems together to implement a custom workflow, the main point of the integration step is to get the right events into the right components. Consuming the CI server, we must often need to inform the system that a new push was uploaded into the code hosting or code review system so that the patch can be tested automatically. In other words, we need to collect various events and convert them into build requests.

On a more abstract level, we are collecting the output events of our system and converting them into the input data for some other system. This kind of interaction is by no means specific to the CI server; every two development tools that need to be integrated will be integrated in this way. From this observation, we can see that the communication subsystem is actually not on the same level as the build subsystem. It does not belong to any particular development service, it is above.

The Build Subsystem

There are two widely adopted ways how to implement the CI server:
1) CI server can emit the management functionality and only include the test execution subsystem. It then serves as a service for other development tools since there must be some other component that has API methods and compose them into higher-level actions. This lead to the conclusion that every development service must be wrapped in custom integration logic that translates the events emitted by other services to API calls. We shall call these components service agents. The communication platform should incorporate a publish-subscribe service that the agents can use to distribute events between each other.
2) The CI server can use the communication platform to send requests to the whole CI server. In this way, other CI agents can trigger builds using a single remote call API and annotate the patch with the build result. This annotation also contains the URL that can be used to access the build output.

The Build Subsystem

The communication platform can be extended with request-reply communication service to make it possible to implement the CI server on top of it. This service shall enable the agents to export services of functionality under certain method names so that other agents can use this method name to invoke the exported functionality remotely. The communication platform shall take care of routing the remote calls between the agents.

In general, the CI server, the whole process then works in the following way:
1) A new push is uploaded and detected by the relevant service agent, which then emits an event using the communication platform.
2) Another agent detects the event and calls the CI server, which is also represented by an agent in the system. This agent exports the relevant functionality over the request-reply service. Once the patch is tested, the output is saved and the result is emitted as an event.
3) This build result event is detected by yet another agent, which calls relevant service APIs and annotates the patch with the build result. This annotation also contains the URL that can be used to access the build output.

The core component of the CI server is the CI agent. The CI agent is a small blackbox that runs on the CI server and accepts requests from the communication platform. When the request arrives, the CI agent processes the request and emits an event to the CI server. This event is then handled by the CI server, which in turn emits an event to the CI agent, which then processes the event and emits an event to the CI server, and so on.

Conclusion

The proposed design and implementation lead to a platform that can be not only used instead of a standalone CI server but it can be used to integrate any number of systems or gather data and generate statistics. The agents are decoupled from each other due to the fact that they only communicate through the communication platform, either via events or method calls. The data routing is handled entirely by the platform and the agents just use the available communication services. Because the agents are standalone executables (processes), they can be installed, started and removed separately. In case one of the agents crashes, the other agents can keep running.

A prototype communication platform called Cider has been implemented as well as a prototype CI server named Cider. A set of benchmarks show that Cider is capable of distributing the data between interested agents fast enough (50 MB/s in the simplest form).