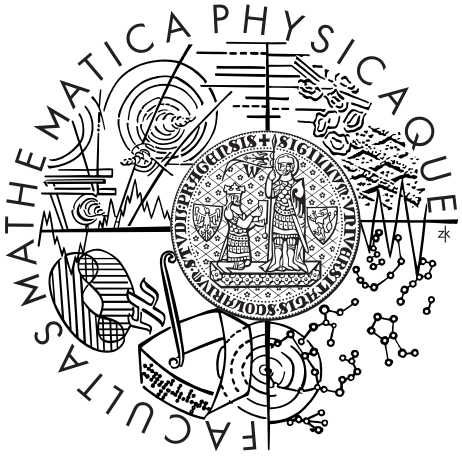
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Ondřej Kupka

Cider - An Event-driven Continuous Integration Server

Department of Software Engineering

Supervisor of the bachelor thesis: RNDr. Filip Zavoral, Ph.D.
Study programme: Computer Science
Specialization: Management of Computer Systems

Prague 2014

I would like to express my gratitude to Filip Zavoral for many pieces of advice, which helped me to carry out this thesis. I would also like to thank all the teachers and fellow students that I met during my studies and who were inspiring and motivating me to continue working.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Cider - An Event-driven Continuous Integration Server

Autor: Ondřej Kupka

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Filip Zavoral, PhD., Katedra softwarového inženýrství

Abstrakt: Profesionální vývoj software vyžaduje robustní vývojový proces. Při implementaci vybraného procesu je často zapotřebí zintegrovat několik vývojářských nástrojů či služeb. Pří návrhu integrace je potřeba rozhodnout především jak propojit dané služby tak, aby bylo celé řešení snadno spravovatelné a rozšiřitelné do budoucna. V této práci nejprve definujeme modelový vývojový proces zahrnující audit kódu a automaticke testování změn. Poté navrhujeme, jak systémy optimálně propojit pro daný proces. Řešení je založené na distribuci událostí mezi nástroji. To dovoluje nepropojovat systémy přímo a nechává prostor pro budoucí rozšíření. V další části práce se snažíme implementovat zvolené řešení za použití existujících systémů. Protože optimální systém pro testování změn nebyl nalezen, přicházíme s vlastní implementací.

Klíčová slova: vývoj software, integrace nástrojů, automatické testování změn

Title: Cider - An Event-driven Continuous Integration Server

Author: Ondřej Kupka

Department: Department of Software Engineering

Supervisor: RNDr. Filip Zavoral, PhD., Department of Software Engineering

Abstract: A solid workflow is necessary when developing software professionally. Manual integration of multiple services has to be often performed when a custom development process is chosen. It must be decided how to connect the tools together so that they are not tightly coupled so that the solution is scalable from the administrative point of view. We present a model development workflow incorporating code review and build automation. Then we propose how to integrate the services in what we believe is the optimal way. The chosen approach is based on the publish-subscribe communication pattern that makes the services decoupled to some extent and also makes it simple to add more services in the future. Having the optimal integration process outlined, we evaluate some of the existing continuous integration servers that could be potentially used for the chosen workflow. After finding no fitting system, we propose and implement a build server that integrates seamlessly with other development tools.

Keywords: development workflow, event-driven integration, build automation

Contents

1. Introduction

When developing software, the act of writing code is just one of the stages in the development pipeline, which represents the life cycle of a change that is happening in the project. Feature or architecture planning can and should precede coding and code review can be and often is one of the stages that follow. When the change reaches the final stage of the pipeline, it is ready to be released.

The trend nowadays is to release often and to release fast so that new features and bug fixes can be delivered to the client quickly. In other words, the development pipeline is optimised for low latency and high throughput. That requires superior process and tools so that the changes can be pushed through the pipeline rapidly and repetitively without harming code quality. Techniques such as automated testing, continuous integration and continuous deployment are often incorporated to make such a process possible.

Continuous integration (CI) is a practice where developers integrate their changes often, usually multiple times a day, to make the integration phase easier or at least less painful by lowering the probability of large merge conflicts or huge incompatible functional changes [7]. By exchanging code changes often, the developers limit to what extent their development source trees can diverge.

To be able to use continuous integration efficiently, a new tool called continuous integration server (CIS) is usually adopted. The task of this system is simple - every time a code change is about to be integrated, the project is built and the automated tests are run to give the developers quick feedback.

Going back to the notion of development pipeline, automated testing is again just yet another stage in the pipeline and the continuous integration server is just one of the services that are being used to govern the whole process. The list of other services participating in the pipeline often includes an issue tracker, a code hosting service and a code review system.

These services have to communicate with each other to make decisions. Since the development process is often custom-made to fit the development team's needs, it may be impossible to adopt a complete solution. Then all the tools have to be chosen one by one and integrated manually. This brings additional requirements upon the components - they must not only fulfill their respective roles perfectly, but they must be as well able to communicate with each other seemlessly and flexibly enough so that any custom workflow can be easily implemented. And this is where many current systems are not sufficient.

This lack of proper integration support is apparent in many continuous integration servers in use today as well. This is closely related to the original intended role of these systems. While many continuous integration systems are trying to embrace as much functionality as possible, supporting this idea by introducing plugins, and thus making the system more or less extendable, we believe that this is an anti-pattern and actually exactly the opposite is the right solution. The continuous integration server should not and cannot be driving the whole development process. It should support the process as much as possible, but only where it fits its original purpose. For this reason we believe a continuous integration server should merely export functionality for other development tools that are driving the process, such as the code review server. To be able to work in this

manner, the system must be naturally able to communicate with other components as it is equally important to being able to run automated tests themselves. The communication shall be event-based because that is a recognized mechanism for making multiple components decoupled from each other.

Looking back at the development process as a whole and all the development tools that govern it, it is apparent that the event-based communication pattern of collecting events and reacting to them is not restricted to the continuous integration server only. It is a pattern that appears throughout all software development services when custom integration is necessary. For this purpose a general communication platform is needed that can be used to handle action-reaction relations between various development tools in a manageable manner that, as already mentioned, keeps the components as decoupled as possible.

The goal of this thesis is thus to design such a communication platform and then use it to implement a proof-of-concept continuous integration server that would seemlessly integrate with other development tools. Unlike many continuous integration servers in use today, the proposed server shall completely hide its internal structure and export its functionality as a service for other development tools to use.

The following chapter contains the definition of the development workflow that is to be used as the chosen exemplar development process. The workflow is explained and analysed, which leads to the list of functional requirements for both the development tools and the integration platform that is to be used to interconnect the development services being used.

In the third chapter we evaluate some of the existing continuous integration servers against the specified functional requirements.

In the fourth chapter we discuss the optimal development platform in more details. The architecture for the new continuous integration server as well as the integration platform is proposed. Interesting design and implementation challenges and decisions are discussed.

The fifth chapter is focused on the prototype implementation of the proposed solution. It is explained how to implement a real workflow using the communication platform primitives and the prototype CI server implementation. The whole system is then benchmarked.

The thesis ends with a conclusion listing what was achieved and what can be further improved.

2. Problem Definition and Analysis

The services integration problem was outlined in the introduction. This chapter contains the definition of a simple and clear development workflow, which is then analyzed and the list of functional requirements is compiled for the continuous integration server that takes part in that workflow as well as the services integration platform as a whole.

2.1 Target Development Workflow

The workflow that was chosen as the model scenario is rather real and can be encountered in practice. Its main purpose is, however, to show what communication patterns can be seen in the development tools. The workflow can be summarised in the following way:

1. Developers work on the requested changes.
2. Once a developer is finished with his or her patch, he or she stages it for code review and automated testing.
3. Once the patch is accepted by the reviewer and verified by the continuous integration server, it is marked as ready to be released and it is merged into the project source tree.

Now it is clear how the overall process looks like, but what is more important is the actual data flow and exchange that is happening in the background between the development services. In other words, what events happen in the system and what actions are to be carried out to react to them? Table 2.1 lists the important workflow events, Table 2.2 then defines the actions that are to be taken upon these events, Figure 2.1 contains the flow diagram of the workflow.

There are some reactions listed in Table 2.2 that were not mentioned before, but it is not really important what these actions are. The patterns that they express are much more important and that is what we will analyse later on. Multiple events written in a single cell in Table 2.2 means that all the events must occur before the relevant action is triggered.

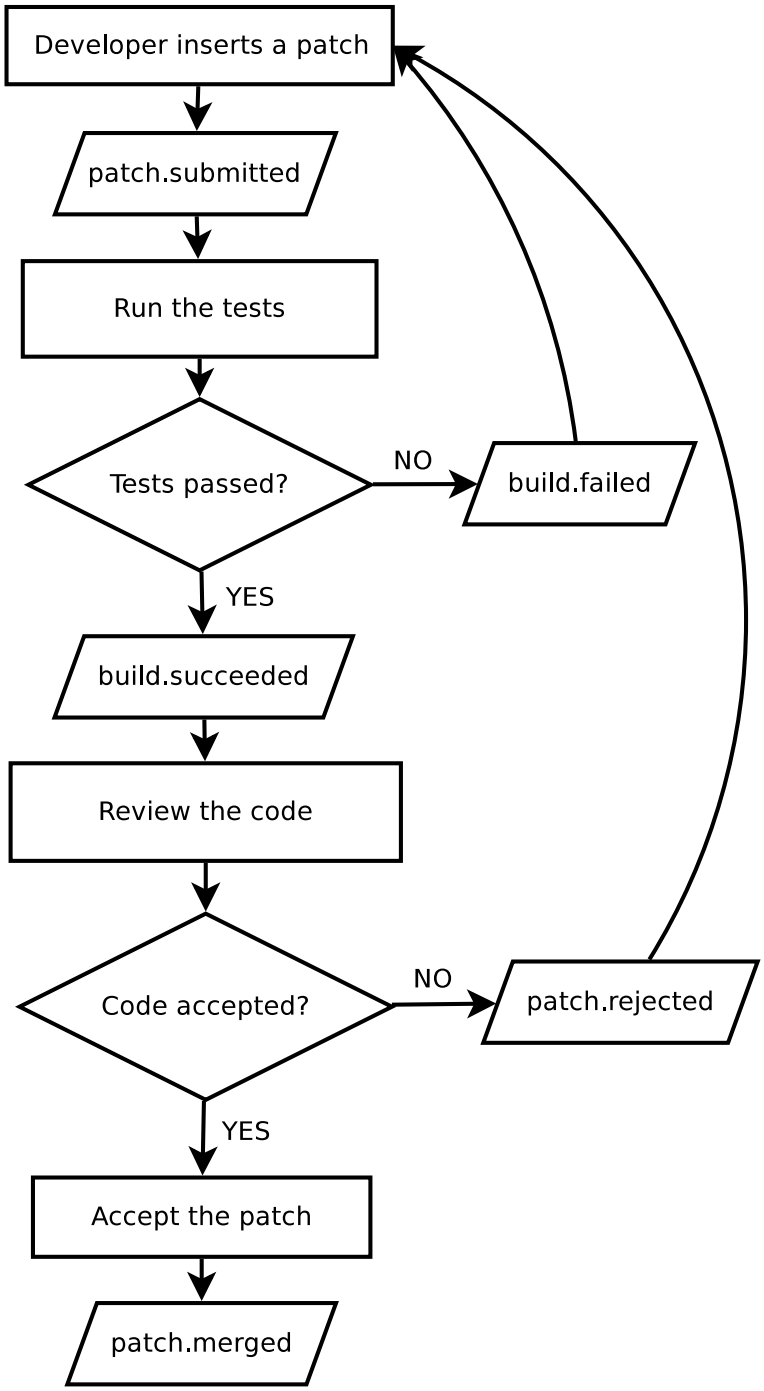
Table 2.1: Workflow events and their meanings

Event	Description
`patch.submitted`	a new patch was submitted into the code review system
`build.succeeded`	the project was built successfully and the tests succeeded
`build.failed`	the project failed to build or the automated tests failed
`patch.accepted`	the patch was accepted by the reviewer
`patch.rejected`	the patch was rejected by the reviewer
`patch.merged`	the patch was merged into the project tree

Table 2.2: Desired reactions to various workflow events

Event	Action
`patch.submitted`	Run automated tests to verify the patch.
`build.succeeded`	Annotate the patch in the code review system so that the reviewer knows the patch has been verified.
	Upload the build artifacts into the artifacts store so that other developers can access and use them.
`build.failed`	Annotate the patch in the code review system so that both the reviewer and the author know the patch is not passing and that some alignments to the code are necessary.
`patch.rejected`	Notify the author of the patch that some comments were added to his or her patch and that it needs some alignments before it can be accepted.
`build.succeeded` `patch.accepted`	Merge the patch into the relevant source tree in the source code management system.
`patch.merged`	Rebuild all the projects that are depending on the source tree that has just changed.
	Mark the relevant ticket as resolved in the issue tracking system.
	Deploy the updated branch into the staging environment.

Figure 2.1: Workflow diagram



2.2 Modeling the System of Integrated Services

Once the workflow is defined, it must be implemented somehow. In our case we would need to interlink the code review service and the continuous integration service. To be able to easily think, speak and reason about this composite system, it is useful to come up with some kind of abstract model that describes the behaviour of the system in simple terms, yet accurately.

Assuming that all the events are happening and are being processed sequentially, we can describe the whole system as a state machine. Let $S_0, S_1, \ldots, S_n$ be the sets of possible states of the development services being used. Then for the state machine representing the whole system and its set of states S it must be true that

$$S \subseteq S_0 \times S_1 \times \cdots \times S_n$$

The state transition function can be defined using the chosen event-handling actions, which naturally change the states of various components in the system. It would be defined recursively using the changes imposed on particular components in the same way the set of states is combined from the states of the components.

Even though this way of describing the system may look artificial, it allows us to express a few observations easily. Particularly the recursive way of defining the system state has some interesting consequences, most importantly that there is actually no global state directly shared between the component. The internal component states compose the global system state and there is no state that exists out of these local scopes. This leads to the observation that there is no need to control these state transitions centrally as long as all the components get the same complete information about what is happening in the system, which means that no events are lost and they arrive into the components in the same order. This guarantees that the local component state changes will always compose a correct global state since all the components will make state transitions connected to the same global change at the same time.

These observations are not particularly useful for analysing existing systems, but they give us some hints on how a new system could be designed. If there is an event-transporting middleware that provides the required semantics, the rest can be implemented in some component-specific modules that also encapsulate the state relevant to that specific component (the components cannot be stateless in general as apparent from Table 2.2 - a patch is merged only after `build.succeeded` and `patch.accepted` are detected, so certain history of events that happened in the past must be kept in some cases).

For the sake of completeness it should be mentioned that going further and pushing the event-distributing mechanism into the components is not desired. As can be seen in Table 2.2, `build.success` event is important for the continuous integration component itself as well as the code review component. Keeping the routing information in the build server would be, however, a major management and scaling issue, effectively breaking any encapsulation and isolation of the system components.

2.3 Continuous Integration Server

We have shown how to describe the composite system of integrated development tools using the notion of state machines. When an event is inserted into the system, interested components can react to it somehow.

The same action-reaction relations can be found within continuous integration servers as well. When a code change is inserted, multiple actions can be triggered. When a project build is finished, notifications can be triggered or the whole project deployed. From the global point of view, this is just another action-reaction relation, although it is local to the CIS. The important point is, however, that we cannot know in advance what events are only needed locally because there can be other services added in the future that can be interested in these CI events. This observation is used later.

We shall turn away from abstract descriptions now and discuss how the optimal continuous integration server and also the whole communication platform should work and what the functional requirements are so that it can be really used to implement the specified workflow.

2.3.1 Role of the Continuous Integration Server

First and foremost, we have to clearly describe the role of the server in the target workflow to be able to identify the communication patterns and functionality the server and the whole communication platform need to support not to keep adding features that are actually redundant and already implemented in other parts of the development stack.

Continuous integration servers, or actually any development services, can be roughly divided into two groups by how they are approached and used - they can act as standalone systems or they can be just services exporting some on-demand functionality for other development tools to use. This seems like a very raw distinction, but it is crucial. It heavily influences how much information must be kept in the system and how much the system must know about its peer development services. A standalone continuous integration server is usually internally organized around build jobs, which are basically some series of build steps representing certain projects or project branches. When a project is supposed to be tested, the relevant job is triggered and it assembles the project and runs the tests, optionally showing some statistics like how often the job fails and what was the last time it succeeded. On the other hand, when all that matters is to verify a patch posted into the code review system, all necessary input information can come from the code review system and the continuous integration server can simply return the test results, which are again stored in the code review system. In this case the only data that needs to be kept in the continuous integration server is more detailed data connected to the patch testing, e.g. the build and testing output. The organization of projects is, however, kept in the code review system, along with links into the continuous integration server. It also does not make sense to generate some of the statistics the standalone servers organized around jobs do since the patches are ariving randomly and they are not really related to each other, they each have its own development pipeline. On the other hand, if the whole system is inherently event-based, it is possible to just add

another component that collects events and generates these statistics.

In our chosen development workflow it is clear that the continuous integration server fits into the category of on-demand continuous integration as a service. That implies a few important points for the required functionality:

1. There is no need for any job management capabilities. All the static information required for a patch to be tested comes along with that patch. This will usually be the address of the repository the patch is supposed to be tested against. The tests and other static information for the continuous integration server are stored in the repository as well.

2. Even though there is no need for build jobs, there still must be a way how to pass configuration into the test runs. This can be for example a database URL to be used for the tests. A good practice is to use environment variables for that, hence there must be a way how to define environment variables on per-project and per-environment basis.

3. The server must be able to listen for events happening in the code review server and vice versa.

4. There is no need for any complex user interface, the links contained in the code review server can lead to static web pages containing the build output, or the output can be streamed to the user directly from the CIS.

2.3.2 Anatomy of the Patch Verification Phase

Even though the overall process was outlines in Table 2.2, it makes sense to describe what is happening between the code review server and the continuous integration server more explicitly. The following list describes the process in more details, also incorporating the observations from the previous section:

1. Once a new patch is submitted, `patch.submitted` is emitted.

2. The continuous integration server integration login detects the patch and fetches it from the code review system. The patch contains information that clearly identifies the source tree the patch belongs to. For Git source code management system this would be the repository URL and the branch (or in general any reference) to test against.

3. Since the repository contains all required configuration for the verification step, it can be scheduled and run. This configuration contains the definition of the environment where to run the tests as well as the relative path within the repository to the script that runs the tests. Testing environments are discussed in section 2.3.4.

4. Once the verification step is finished and its output saved, `build.succeeded` or `build.failed` is emitted. This can be as well joined into `build.finished` that contains the results. In any case, the event emitted contains the link to where the build output and results can be viewed.

5. The code review server integration logic detects that a verification run has finished and it annotates the relevant patch accordingly.

2.3.3 Anatomy of a Test Run

In the previous section the test run step is mentioned without any details. That is, however, exactly the phase the continuous integration server takes care of. The server must go through the following steps:

1. Parse the `patch.submitted` event to get all the necessary information from it, particularly where to find the relevant source code repository.
2. Fetch the environment definition from the repository. Find or create the testing environment fitting the project requirements.
3. Use the environment to run the specified script. Keep saving the script output as the script is being executed.
4. Generate a link where the output can be accessed and emit `build.finished` event containing the link together with the build and test results.

2.3.4 Testing Environment Management

Even though the previous section describes what the CI server should be doing in details, there is still one non-trivial task that has not been discussed. It is the step where the required environment for the test run is found and set up.

From the project build and test run perspective, the build and testing scripts must be run on a host that is running the right operating system, having the required development tools and dependencies pre-installed (if they cannot be installed automatically before the test run itself). This list of dependencies together with the environment variables to be set for the relevant scripts forms the testing environment definition.

From the CIS perspective, given such an environment definition, the system must be able to find or create the right environment and run the relevant scripts there with the specified environment variables set. The system must be able to manage these environments somehow. This task is usually approached from the following two sides (existing systems are discussed in Chapter 3):

Constant set of persistent environments A set of persistent long-lived environments is kept in the system and the environments are reused for the test runs. This approach has zero test run initialization overhead since the environments are simply always there and they are not shut down ever. On the other hand, the test runs may modify the environments in a non-trivial way, thus making them diverge and potentially break if the test scripts are not written carefully. Also to manage all required versions or various software packages on a single system may be problematic. Last but not least, this does not scale since the environments are added and removed manually.

Dynamic on-demand environments The testing environments can be as well created and set up on the fly as needed. When a test run is requested, a brand new environment is created and it is again destroyed after the test run is finished. This scales well and potentially saves resources, but it makes the system much more complex. The positive aspect is then that the test runs are always executing in a clean undamaged environment.

The answer to the question of which approach is better is: it depends. But it is still rather simple: the persistent set of environments may make sense only when the number of environments is small and all the scripts potentially running there are known not to cause any undesirable side effects. This approach, however, allows some bad practices, such as basing the environments on manually set up physical machines. This can work, but it brings management problems and if no configuration management is in place, the machines can simple crash or burn and the whole process of setting up the environments must be, again manually, repeated. If virtualization together with proper configuration management is used, this approach is safe and fine.

On the other hand, if virtualization and configuration management are being used, we are only a small step from being able to implement the on-demand approach, which is better in almost all aspects:

- The test runs are executed in clean environments, which is a requirement if we want a solid reproducible test results. Build failures caused by unpredictable build environment are very undesirable since they are bringing chaos into the process in the form of false negative build results.
- Untrusted scripts can be run in the environments since the possible harmful system modifications are not persisted in any way.
- Only the resources that are required at the moment are allocated.
- It is easy to do load balancing, replace faulty hardware and so on since we can decide where to place the environment every time it is requested.

The on-demand approach is, however, not for free:

- It requires more initial configuration involving virtualization and configuration management. Virtual machine or container images must be prepared in advance. This, however, pays off in the long term.
- Following the previous point, it must be noted that the virtualization software can be expensive in case there is no free alternative supporting the target operating system that is to be virtualized. Mac OS X does not have very good support in any free virtualization framework, especially because even virtualized it must run on Apple hardware because of legal reasons. This may be a reason to mix the persistent and on-demand approach together.

To summarize the points mentioned, there is in the end not much management and resources overhead incorporated in the on-demand approach. On the other hand, it brings many benefits in the way it enforces reusable environments and good practices in general. Some kind of mixed approach where some environments are on-demand and some are persistent can be implemented in case the target platform is very difficult or impossible to virtualize. The optimal system should support this.

There is one more interesting and not immediately apparent problem affecting the persistent approach. It is easier to create desired environments than to find

existing ones complying to the requirements. If we want to use a pool of persistent environments, we either have to hardcode what environments are to be used with what projects, or we need some kind of survey service or protocol that can be used to ask the existing environments if they comply or not.

The mixed approach could be implemented by having a pool of environments where some of the environments would be a meta environments implementing the ability to create new environments. These environments, when queried, would answer positively if they are able to create an environment complying with the requirements. The environments management system could then query all the environmets and transparently use a persistent one or an on-demand one since the interface would be the same.

2.3.5 Functional Requirements

To conclude this section, we present the final list of functional requirements that define the optimal continuous integration server. What is not required is also explicitly mentioned. The list is built on the ideas mentioned in the previous sections and it is strongly influenced by the fact that we are not aiming for an all-embracing continuous integration server, but rather for one that acts as a flexible service for other development tools.

1. Server actions are triggered by events inserted into the system. Manual test runs make no sense unless they follows a code change. So there must be an interface that can be used to insert relevant events into the system, or rather a way how to allow the server to pick events that are of interest.

2. The events can come from various development tools. The variety can be great, so it should be easy to teach the system to accept new kinds of events it has not encountered before.

3. The server itself holds very little configuration associated with the test runs themselves. The static configuration is kept in the project repository as a configuration file. The only dynamic configuration mechanism that must be supported is to be able to define environment variables for the test runs on per-project and per-environment basis.

4. The server must be able to find or create an environment where the tests can be run and that complies with the requirements of that particular project. Static environment assignment is fine, but the dynamic on-demand approach is preferred.

5. The test output and results must be accessible under a URL.

6. When a test run is finished, an event is emitted that contains the test results as well as the address where the output can be accessed. The system should not directly insert the event into other development tools, it should merely emit it and let other systems handle it if they are interested.

7. The system supports test runs on multiple platforms, possibly Linux, Mac OS X and Microsoft Windows.

To be able to use or construct an event-based continuous integration server and implement the desired development workflow, there must be a medium that makes it possible to transport and route events between the development tools. The continuous integration server should not be implementing this functionality since it is not bound to continuous integration in any way. It is a general action-reaction concept present throughout the whole system of development tools that makes it possible to decouple the components from each other to some extent.

2.4 Development Tools Integration Platform

Custom-made development process is often implemented using multiple development tools that interact with each other. Some of the tools help to manage the whole process by keeping important data inside of them (for example the issue tracker), some may only export functionality for other tools to use, like the continuous integration server as described in our workflow. What is then necessary is a communication mechanism that can be used to integrate these tools together. A proper communication pattern must be chosen for every scenario. The following patterns are generally supported across the development tools in one way or another:

Request-Reply One component uses the remote API of another component to get necessary information or trigger an action in the target system, usually synchronously, for example to get the list of active user accounts. There is a single source and a single destination component. Remote APIs following the REST paradigm [4] (using HTTP as the transport protocol) are very common these days.

Publish-Subscribe One component publishes events and other components can pick the events and data they are interested in. The communication is happening asynchronously, there is a single source component and potentially multiple destination components. Webhook [12] is a mechanism that is commonly used today. When an event happens in the system, an HTTP POST request is sent to the designated URL with the event details in the request body. The weakness of this approach is that some systems only support a single destination URL, so the event distribution step must be implemented manually. Webhook is actually only addressing the last mile problem (how to get the event into another system). There must be a middleware that can collect these POST requests and distribute the events to the subscribed components.

Each of these patterns has its own place. It cannot be said that one of them is wrong, the patterns can be only used in a wrong way. Request-reply makes perfect sense when a response is expected, but it should not be used to cast data to other components, because that is what publish-subscribe pattern is there for.

This observation seems trivial, but it is actually very important for integration of multiple development tools that we are discussing here. Only when a response is expected, a direct remote API call shall be issued. This simple rule also makes the components less coupled together since the event-emitting component does not have to know all the destination components. The components are

always somehow coupled together because they must understand each other's data model, but using publish-subscribe makes other components observe the source component rather than the source component being the active participant. This is making the whole communication process more transparent and easier to manage since the listening components can be added and removed without the source component knowing anything [2].

2.4.1 Functional Requirements

When we analyse the target workflow, it is apparent that all the communication that happens there is of the publish-subscribe communication pattern - the continuous integration server observing the code review tool, the code review tool then waiting for the build results. That means that the integration platform should support the publish-subscribe pattern no matter what.

Request-reply, however, is also extremely common and should not be missing, so we shall list it among the patterns that are to be supported as well.

At the end it must be noted, though, that we cannot simply list the requirements without mentioning the development tools themselves. There must be some support implemented in the tools to be able to integrate them. Particularly they must somehow emit events when something significant happens. It does not really matter in what manner this is implemented. A remote API must be also available so that the workflow integration logic can access the tools and actually perform the actions requested. This is not required if the event handler can be implemented as a plugin directly in that particular development tool.

2.5 Summary

In this chapter we defined the model software development workflow. Then we discussed how the optimal continuous integration server should work so that the chosen workflow can be easily implemented. Since the development tools themselves are not floating in the void and they need to be integrated, we also discussed how the communication platform should look like from a very high-level perspective since it is a crucial part when integrating multile components together. The platform is discussed in more details in Chapter 4.

In the next chapter we are going to describe some of the existing continuous integration servers and evaluate them against our list of requirements.

3. Existing Continuous Integration Servers

In this section we evaluate some of the existing continuous integration servers against the requirements, extending and updating previous research conducted on this topic [9]. The following systems were chosen for their popularity and compliance with our requirements (or lack thereof):

- Travis CI
- Drone
- Jenkins
- Buildbot

3.1 Travis CI

Although open sourced, Travis CI is primarily a hosted continuous integration server specifically architected to work closely with repositories hosted on GitHub [10]. All that is necessary to start using Travis is to sign up and activate relevant GitHub repositories. Travis will install hooks to detect changes and trigger builds. It requires the repository to contain a file called `.travis.yml` that holds all configuration for Travis.

Travis is written in Ruby and it uses RabbitMQ message broker to scale and make the components decoupled from each other.

3.1.1 Travis CI Build Lifecycle

Travis uses `.travis.yml` contained in the source repository to set up the build. There are, however, many services in the background that take care of build scheduling, log streaming and other tasks. The whole lifecycle of a build including the components communication looks as follows:

Listener The first component is listening to changes happening on GitHub. When a change is detected, a message is pushed onto RabbitMQ for other interested components that want to process that event.

Hub Another component collects and routes events to the interested components. For example it communicates with the notifications component, which takes care of sending notifications. The hub also enqueues build jobs and enforces Quality of Services.

Worker This component is responsible for running builds in a clean environment. It uses predefined virtual machines that are always rolled back after the build is finished so that they can be reused. Build output is streamed into the logging component, build events are pushed back to the hub.

Logs This component receives build output and streams it into the web interface while at the same time the output is being saved into the database. When the build is finished, the complete log is pushed to Amazon S3.

Web This is the web application where users can configure and watch their projects being built.

3.1.2 Evaluting Travis CI

Considering our system requirements, the positive aspects of Travis CI are as follows:

- `.travis.yml` is used to define the build configuration, including the environment that is to be used. This file is a part of the project repository.
- Every build is run in a clean virtual machine, making the builds easily repeatable without any undesirable side effects.
- The build output is accessible through the web interface.
- Linux and Mac OS X are among the supported build platforms right now, Windows support is a work in progress. Travis CI is open sourced, so theoretically it can be deployed on premises. Then the limitation of the paid tier can be to some extent avoided for the price of a private system deployment and administration.
- Travis CI supports webhooks and many other post-build notifications. These can be used to emit events to be shared with other development tools.

On the other hand, Travis CI cannot be used to build arbitrary patches. A build can be triggered only by pushing a Git branch to GitHub or by sending a pull request. Travis CI is tightly coupled with GitHub in this respect and no other input sources are available.

3.1.3 Summary

Travis CI was among the first systems that appeared on the market that brought some very interesting new ideas into continuous integration. Particularly that there is a single file in the repository that holds the build specification. The important point is that the file also specifies the environment that is necessary to be set up for the build to work, and Travis CI takes care of automatic environment creation and disposal.

Considering how Travis CI can cooperate with other tools, there is no problem with the build output. Build logs can be accessed and post-build notifications can be sent out. The main issue is that it is closely bound to GitHub on the input side. No build can be triggered without a branch being pushed to GitHub. An additional layer of indirection would be needed here.

Travis CI cannot be easily used as a service for other development tools. It was built to work with GitHub and GitHub only. It works nicely when GitHub pull request mechanism is being used for code review, but if that is not the case, there is no way to align Travis CI for custom needs right now.

3.2 Drone

Drone is a continuous integration server that appeared recently [1]. It is written in Go and it utilizes a new container-based virtualization platform for Linux called Docker. That means that only Linux is supported as the environment for building and testing projects. There is a hosted edition of Drone as well as an open source edition. We discuss the latter one here.

Drone itself is largely inspired by Travis CI, which was discussed in the previous section. It also expects a special file called `.drone.yml` to be present in the repository. Again, that file defines what environment to use and what command to run. A modern web inteface is available for Drone, which can be as well used to configure the build, for example to specify some secret build parameters.

Unlike Travis CI, which uses virtual machines for build environments, Drone uses container-based virtualization, which is much more lightweight and allows Drone to allocate less resources faster [13]. The negative side is that Drone only supports environments that can be run on Docker, which are basically only the Linux distributions running a recent version of Linux. This can change in the future since recently Docker introduced a plugin architecture that makes it possible to implement drivers for other virtualization software.

3.2.1 Evaluating Drone

Facing our list of requirements, Drone certainly incorporates some interesting practices:

- As with Travis CI, the configuration is saved in the repository. Environmental variables can be defined in `.drone.yml` or in Drone's web interface.
- The build environments correspond to Docker images, which are similar to virtual machine images. Container-based virtualization is, however, much more efficient than the traditional hypervisor-based one. The containers share the operating system kernel, so running a container is about setting up a sandbox for processes rather than booting another operating system.
- The build output can be accessed using a modern web interface.
- As with Travis CI, many kinds of post-build notifications are available, including webhooks.

Unfortunately there are facts that are making Drone fail to comply with our list of requirements. As with Travis CI, Drone as of now only supports certain code hosting services - GitHub support is implemented, BitBucket and GitLab support is a work in progress. As of now there is no general mechanism how to add support for more input sources. It is, however, very likely that it will be added in the future. Also the fact that Drone is built on top of Docker basically puts a hard limit on what target build platforms are supported. In this case it is only Linux, for now.

3.2.2 Summary

Drone takes the ideas behind Travis CI one step further by using Docker. If Linux is the only platform that is to be supported, Drone is almost certainly a better choice than Travis, also because it is very easily deployable. Since it is written in Go, the whole system is just a single statically-linked executable.

Unfortunately Drone is affected by the same set of issues as Travis CI. It can emit events when a build is finished, but on the input it is closely bound to the code hosting service. As of now there is no general mechanism that can be used to build and test arbitrary patches.

3.3 Jenkins

Jenkins is a continuous integration server written in Java [3]. It involves a single master server with multiple build slaves, which can run on Windows, or over SSH on any system that supports it. No additional setup is necessary.

The system itself is very simple to use, as it is to install. The user is provided a web interface to control all aspects of the system including build slaves management and build jobs definition. Jenkins supports plugins and there are already many of them, implementing support for various build steps and post-build actions. There is even a plugin for implementing whole development pipelines.

For defining jobs there is a per-job configuration page where it can be set up how the job is triggered, what the build steps are, and what notifications to send out once the job is finished. A job can be triggered manually by clicking, by using the RESTful API or by polling the code hosting service. A few code hosting services are supported explicitly so that the webhooks set there can be directed to Jenkins to automatically trigger builds. Since Jenkins supports job chaining, a job can be also triggered after another job is finished.

3.3.1 Evaluating Jenkins

The following list summarises and extends the list of positive features:

- Extremely simple to set up.
- Simple to define jobs using the web interface.
- Support for multiple platforms.
- It contains many plugins to perform advanced operations or talk to other services.
- It contains a full-fledged web interface to manage the whole system, including a RESTful interface. The user can also watch his or her jobs being executed, having the output streamed into your browser.
- The configuration is stored in XML files. It is possible to easily generate all the configuration and automate many management tasks using the RESTful API.

- There are plugins for almost anything, including the Libvirt plugin, which can be used to virtualize build slaves instead of using physical machines.

However, comparing Jenkins to our list of requirements, severe limitations and incompatibilities are uncovered:

- Jenkins is not really event-based in the sense that it would be possible to insert events into the system and let it react. There is some support for GitHub post-receive hooks that can trigger jobs connected to the relevant repository, but otherwise the rest of the development tools must know exactly what jobs to trigger. There is no support for any indirection, all you can do is to directly manipulate Jenkins internals by using the remote API. This makes working with Jenkins exactly the opposite of the service-oriented approach that we proposed. This can be to some extent mitigated by using custom plugins.

- Jenkins configuration is based on build jobs, which are configured using the web interface. The way Jenkins is configured supports bad management practices. It is much easier to use the web interface than to commit the scripts into the project repository, and also by using the web interface the build jobs themselves are not under version control, which is very desirable.

- The build slaves are managed very statically. Labels can be used to assign build slaves to jobs, but that is all there is to it. There is no environment creation or discovery really. There are, however, plugins that implement libvirt-backed build slaves, so virtualization can be used to create reusable environments when needed.

- Since Jenkins is not really event-based, no implicit event is triggered when a job is finished. All desired post-build behaviour must be explicitly written into Jenkins right into the build job.

Not counting the list of requirements, there are other points that are making Jenkins less suitable as the solution we are seeking:

- The system is not particularly stable and consistent. Every operation is accompanied by the fear of breaking the whole system. One can see an exception basically any time. The project issue tracker is full of bugs, but they are fixed only occasionally and many issues have been open for years.

- The system is in general not very consistent in the way jobs are configured. This is mostly because of the plugins, which can each work in a slightly different way.

- Even though the web interface is rather powerful, it is extremely ugly in its design and implementation and it is slow.

3.3.2 Summary

Jenkins is a user-friendly system to use, but all the positive aspects are completely overrun by the amount of bugs and inconsistencies.

Facing our requirements, Jenkins is exactly the system we are not looking for, even though it complies to some extent. It is the standalone continuous integration server type built around the notion of build jobs. The system includes a lot of functionality implemented through plugins, but it is not flexible at all. Once simple system, perhaps, was pushed to its limits by the need for advanced functionality that is was not built for. Job chaining looks like one of the features that was added later and even though it looks powerfull, in reality it brings more problems than it solves.

Jenkins is an unstable and inconsistent system bloated with enormous number of plugins of variable quality that should be avoided when possible.

3.4 Buildbot

Buildbot is not a complete continuous integration server. It is a framework that can be used to implement continuous integration processes for custom workflows [11]. The core idea is that the system should not impose any restrictions on what can be achived with it, the user should merely use the framework to implement his or her own processes.

A Buildbot instance comprises of a build master and some number of build slaves. The system contains the following core components [8]:

Builders Build jobs are called builders in Buildbot. They specify what build steps are to be run and what build slaves can be used for that.

Schedulers There must be a component that tells builders when to run. This is the role of schedulers. They bind events and builders together. An event can be simply a timeout that periodically triggers a builder, or it can be a new patch submitted into the code review system.

Change sources Components taking care of inserting code change events into the system are called change sources. Almost any known source control management repository can act as a change source.

Status targets To send out notifications after the build job is finished, various status targets can be specified, including IRC bots, mailers and a simple web interface.

The idea is then that the Buildbot user will specify his or her own builders, schedulers, change sources and status targets, binding them all together in a custom way to implement the desired workflow. The user can achieve this by using Python programming language. Since Buildbot is written in Python, the configuration is simply imported when Buildbot is started and becomes a part of the build master program.

3.4.1 Evaluating Buildbot

Buildbot is very close to what we consider the optimal system:

- The system is indeed very flexible and there are no obstacles for implementing a custom workflow.
- The system can accept events by using either existing or custom change sources and schedulers.
- The system can emit events by using either existing or custom status targets.
- Many build steps, schedulers, change sources and status targets are already implemented. All the user has to do is to import relevant Python module.

Even though this looks very good, there are unfortunately other requirements that are not that simple to fulfill with Buildbot:

- The build steps cannot be defined in the project repository, the configuration is done in Buildbot itself. This breaks an important principle - the build scripts should be written by the person developing the project and they should be kept as close to the sources as possible. This is a good practice. This could be implemented in Buildbot, but it would require at least custom change source and scheduler since these are the components that are binding things together. Buildbot was not really built to dynamically decide what steps to take based on the contents of the source repository. Build steps are defined in the builder factory statically and loaded when Buildbot starts.

3.4.2 Summary

Buildbot offers a lot of flexibility. It can be plugged into any existing system on both input and output by writing custom change sources and status targets. The way Buildbot is configured is, however, not compatible with our requirements. The configuration cannot be simply committed into the project source code repository. It would not make sense since the configuration contains all the relations between Buildbot components, including what build slaves to use for build steps execution. That is not something that should be committed into the repository.

3.5 Other Continuous Integration Servers

We also considered two commercial systems - Atlassian Bamboo and ThoughtWorks Go. They both include a mechanism for triggering plans (in case of Bamboo) or pipelines (in case of Go) remotely. For Bamboo either repository polling or direct API call is available, Go can be just notified about a change and it will detect what pipelines are affected.

So, it is possible to insert push events into these systems, but that is not the issue here. These systems are very powerful and complex, but that is not what

Table 3.1: Comparison of the existing continuous integration servers

	Listens for events on input	Set of input events is extensible	Emits events when finished
Travis CI	YES	NO	YES
Drone	YES	YES*	YES
Jenkins	YES*	YES*	YES*
Buildbot	YES	YES	YES
	Configuration in repository	Output accessible	Support for desired platforms
Travis CI	YES	YES	NO
Drone	YES	YES	NO
Jenkins	NO	YES	YES
Buildbot	NO	YES	YES

is requested. They fall into the standalone system type bucket and they do not fit well into the idea of a large composite distributed system of smaller pieces doing their job well and exposing pieces of functionality to each other. These systems cannot be asked to simply build a random patch since their role is to manage whole releases of software packages. So even if they incorporated the functionality that is requested, the philosophical clash is too big. For this reason we do not analyse these systems in more details here.

3.6 Summary

In this chapter we evaluated some of the existing continuous integration systems. There were basically two major requirements - the optimal system should be able to communicate with its surroundings using events, and it should not incorporate too much functionality that is not necessary. In other words, what we were looking for was a system that could be used as a service for other development tools.

What was concluded is that there is no continuous integration server that would comply with all the requirements. All of the existing solutions that we discussed always partly comply and partly fail. The results of our comparison are shows in Table 3.1. Systems having YES* in particular feature column implement the feature partially and/or extra efford is necessary to make it working. The table can be summarized in the following way:

- Travis CI comes with a single file placed in the project repository that keeps the build configuration, including the environment definition. Unfortunately, Travis CI is tightly coupled with GitHub and cannot be really extended easily to accept other input events to start build jobs.

- Drone builds on the same ideas as Travis CI, just making the build process more efficient by using containers instead of virtual machines. On the other hand, since Drone is using Docker to manage the build environments, it can use only Linux as the build environment for now. Drone has the same set

of disadvantages as Travis, although contributors actively work on making it more general-purpose than Travis CI.

- Jenkins support multiple source code management systems and Linux, Mac OS X and Windows. The configuration is, however, kept in Jenkins, and it is very static. The system is unreliable and inconsistent because of the plugins from various contributors.

- Buildbot has an interesting architecture of change sources, schedulers, builders and status targets, which is nicely event-based. On the other hand, the project build configuration must be kept within Buildbot.

Since no optimal existing solution was found, the next chapter contains our proposal of how to design a new continuous integration server and the related communication platform so that the chosen development workflow can be implemented easily.

4. Designing the Optimal Development Platform

As concluded in the previous chapter, there is no existing continuous integration server fitting the requirements entirely. In this chapter we try to address that fact by proposing a new system that would suit the chosen development workflow.

Our field of interest is not only a new continuous integration server, but also the integration platform that is to be used to connect development tools together.

The following section discusses how the communication platform should work in more details. Later on we discuss how to build the continuous integration server to both comply with our requirements and fit into the new communication framework.

4.1 Development Tools Integration Platform

In Chapter 2 we mentioned two basic communication patterns that are happening between the development tools - request-reply and publish-subscribe. These are, however, very abstract patterns and even though they may be supported in the tools that are being used, it is also important to know the implementation details, particularly if there are any communication protocols common to all the development tools, which could be potentially reused for the integration step.

Considering request-reply, most of the current tools support some kind of remote API, most commonly using HTTP as the transport protocol and following the REST API design paradigm. In this paradigm, and considering HTTP, the target system resources are represented as URIs and HTTP methods represent certain resource-specific actions. From the design point of view, such an API is perfectly fine. What is less fine is the protocol being used. HTTP was not made to support any kind of asynchronous RPC , which is the natural way how to implement any well-performing request-reply communication between any number of permanently connected components. The right way would be to establish a long-lived connection, send requests and wait for the (possibly partial) replies to come as the requests are being fulfilled. This is, however, not how HTTP works. Even when the TCP connection is reused, since there is no request identifier contained in the HTTP protocol itself, the replies must come in the same order as the requests were sent so that the sender can pair replies and requests correctly. This means that a time-consuming request can block all other requests in the queue. The solution then is not to reuse connections, which makes the whole communication much slower since a new TCP connection is needed for every request. So, a better way how to implement request-reply communication pattern is to use a transport and a protocol that truly supports asynchronous communication, and that is how the optimal platform should work.

For publish-subscribe, the situation is a bit more complicated and varied, but when a development tool supports some kind of post-action hooks, it most commonly supports webhooks as mentioned in Chapter 2. That means that a URL can be specified in the system and an HTTP POST request is sent to that URL every time an important event happens in the system. The request

body then contains the event details. This is fine, but naturally there must be an HTTP server that can process such requests. On the other hand, taking Gerrit code review system as an example, the events there can be streamed over an SSH connection. This means that there is not really any single way how all the tools are publishing events.

As also mentioned in Chapter 2, webhook is a technology to solve the last mile problem, in other words, how to deliver a payload to another system for processing. After having an event inserted into the system, it must be routed to the components that are interested in it so that these components can react to the events happening in the system.

So, there is no single technology that is supported across all the tools that can be used to transport requests or events. Moreover, the event distribution and handling part of publish-subscribe represents the custom integration step, so it must be build from scratch in any case. Appropriate technologies must be chosen to take care of this.

4.1.1 Managing Inter-Component Communication

There is still one question that remains unanswered: How to manage the communication between the components? How to form this large composite system of interconnected development tools?

Having the components access each other directly is not a viable solution. This is not only a clear design and management anti-pattern, but considering webhooks as an example, there is often just a single target URL to be notified of the change, but multiple tools can be interested in the event, so at least for publish-subscribe, we clearly need an extra level of indirection that can be only achieved using a common shared communication medium that links all the tools together.

Considering request-reply, the situation is actually simpler here since what we are talking about is a direct peer-to-peer communication that does not really have to be governed by any intermediary. On the other hand, if we are already forced to incorporate a shared communication medium, incorporating request-reply there allows for advanced request routing and automatic resource discovery so that the components being integrated do not have to know each other's address in advance.

Accepting a common communication medium opens a couple of new questions:

1. How to plug the development tools into the communication platform?
2. How to route data between the components? In other words, how to address the components and how to deliver the data?

Considering the first point, there is no access mechanism that is common to all the development tools, although some are more common than other. This means that every development tool must be wrapped in some kind of connector component that represents the relevant tool in the system and translates the protocol of the communication platform into the protocol of the tool itself.

The second point is a bit more complex. We have to distinguish the publish-subscribe and request-reply pattern. The former pattern is basically a broadcasting pattern where every event has one sender and multiple receivers, so there is

not much to discuss since the pattern is well understood. It can be implemented by having the components subscribe for an identifier that represents certain event type and by doing so they register to receive a copy of the event matching the identifier every time it is emitted.

While publish-subscribe is a fire-and-forget kind of communication where no data is returned to the emitting component, request-reply requires a bit more sophisticated routing since the request must be delivered to exactly one component, which is the one that exports the requested functionality. Once the request is fulfilled, the response must be routed back to the requester.

The following list summarizes and also slightly extends the ideas contained in this so far rather abstract section:

- The system consists of multiple development tools.
- These tools need to communicate in a way that is clear and manageable. Only a common shared communication platform provides this.
- There is a need for publish-subscribe communication pattern where a component should be able to subcribe for particular event type to start receiving all the events of the given type.
- There may be a need for request-reply communication pattern, although not really present in our target workflow. Components should be able to export functionality under specified names and other components should be able to access it remotely. The communication should be asynchronous, which means that there is no need to wait for the reply before another request can be sent.
- Since there is no communication protocol common to all the components, every component must be represented by a connector, or agent, that translates the platform protocols into the protocols being used by the relevant component. This is not tied specifically to any communication pattern, both publish-subscribe and request-reply need these connectors. The agents thus serve as facades that contain the event handling logic and translate the exported methods into the API calls of the relevant component.

Keeping this idea of interconnected agents talking to each other using one of the communication patterns, we shall now discuss how the optimal continuous integration server could be implemented to integrate well with the platform. This may give us more hints on what the exact communication semantics are supposed to be.

4.2 Continuous Integration Server

The communication platform has been described from a high level perspective. It is a set of agents representing the development tools, or in general any functionality. Every agent can emit events, subscribe for events, export methods to be called remotely as well as call remote methods exported by other agents.

In this section we describe how the target workflow, and particularly the continuos integration server, could be implemented using these primitives. We will limit ourselves to the communication between the code review system and the continuous integration server to make things simpler since that is enough to represent the communication patterns that are present in the system.

As specified in Table 2.2, this is the communication that is happening between the two given components:

- On `patch.submitted`, trigger a build in the continuous integration server.
- On `build.succeeded` or `build.failed`, put a comment into the code review system.
- On `build.succeeded` and `patch.accepted`, merge the patch into the relevant project source tree.

On `patch.submitted`, the CI server must go through the following steps:

1. Fetch the relevant build configuration file from the repository the patch is targeting.
2. Find a fitting build environment and enqueue the build request.
3. Once the build is finished, save the output and make it publicly accessible under a URL.
4. Emit `build.succeeded` or `build.failed` event.

As mentioned many times, the build configuration file should contain the test environment definition as well as the name of the script that is to be run. To make the prototype system simple and implementable in a reasonable amount of time, let us state here that the environment definition is just a label, a string that identifies target persistent build environment, which can be a physical or virtual machine. The label could be `macosx109` to run the build script on Mac OS X 10.9, or any other user-defined string.

For simplicity, let us assume that all the environments of the same label are the same in terms of their capabilities and installed software, and all the build dependencies are pre-installed. This is a rather strong assumption, but many small or even larger companies could and probably would use the system in this way, especially if they have some kind of exotic system requirements. Anyway, what this assumption means is that if the build configuration file says that the testing script is supposed to run in the environment labeled `macosx109`, in can be run in any environment matching this label.

Trying to reuse the communication platform, we propose to simply use the environment label as the method name in RPC calls. In other words, every build slave should export a method for every environment label that the build slave is tagged with. Calling one of these methods remotely effectively enqueues the build request for the given environment. Since we required the request-reply communication to be asynchronous, it does not really matter how long the request spends just waiting for being executed since idling is not blocking any other pending requests.

This architecture has one obvious benefit - the communication platform is reused for a large part of what the continuous integration server needs to implement - testing environments discovery, management and build requests queueing and routing. It, however, puts some additional requirements on the communication platform implementation, particularly the request-reply pattern. Since a standard feature of current systems is to see build output being streamed into the browser, the request-reply subsystem should allow live output streaming so that the build slaves can stream output to the agent requesting the build.

So, a more detailed sequence of steps that the continuous integration server must go through is as follows:

1. Fetch the relevant build configuration file from the repository the patch is targeting. Read the environment label from the configuration file.

2. Ask the platform to execute a remote call, the method being generated from the environment label. There are two method arguments required, one being the patch that is being tested since by definition it also contains the relevant source code repository address and revision. This is enough for the build slave to be able to clearly identify and generate the source tree to be tested. The rest can be again read from the sources. The other argument is the set of additional environment variables to be used for the test run.

3. As the output is being streamed back, stream it to the user so that he or she can see the progress. This step is optional but generally desired by the users.

4. Once the build is finished, save the output and make it publicly accessible under a URL.

5. Emit `build.succeeded` or `build.failed` event containing the URL of the relevant build output.

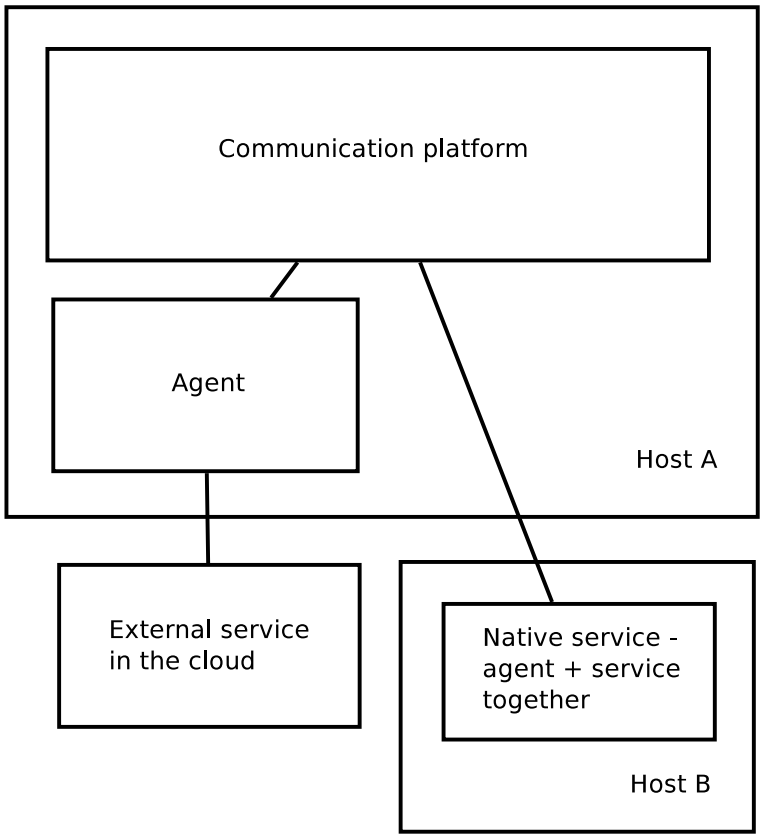
4.3 Communication Platform Architecture

While discussing how to implement a new continuous integration server in a way that would make it integrate nicely with the proposed communication platform, a few more requirements appeared, particularly associated with the request-reply pattern. The full picture, which contains the proposed architecture for the communication platform, is following:

- The system consists of multiple development tools or in general any components that need to communicate together using either publish-subcribe or request-reply pattern.

- These components are either native in the way that they are directly plugged into the communication platform, or there are some agent or connector components that translate the protocols supported by the tools into the protocols being used by the communication platform itself.

Figure 4.1: The communication platform



- For publish-subscribe, every event is associated with an event type string that other components can use to subscribe to the relevant event stream. As soon as the component is subscribed, it starts receiving the requested events, which are coming in the same order they were emitted.
- For request-reply, components export methods under specified names. Other components can call these methods remotely by the given name and the communication platform takes care of routing the requests to the relevant components. Live output or progress streaming must be supported in the direction from the receiver to the sender.
- To support multiple build environments, the system must be able to go distributed in the way that the agents, being native or not, can be scattered across multiple machines.

The ideas are illustrated in Figure 4.1.

4.4 Managing The Agent Components

The previous section gave us a very clear overview of how the platform works. What has not been discussed in more details, however, are the agent components. What are they, actually? Before answering this question, we must first understand precisely how they are supposed to work.

As we already know, every agent is a standalone component that either represents a development tool in the system or just implements some other useful functionality that is needed. As mentioned in the continuous integration section, is must be possible to scatter agents across multiple machines. This implies that they must be somehow separated from each other, and it should be possible to manage them in a way that they can be added, started, stopped, upgraded and removed from the system independently. Some other nice-to-have features could be:

- The agents can be implemented in various programming languages. This is not really a requirement, but every programming language is usually good for solving particular group of problems. And there are also personal preferences of every developer.
- Is is possible to run the agents somehow sandboxed so that they cannot harm each other. This is useful in case the whole system is offered as a service, even if the client is just another development team.

The most obvious solution is to use standalone processes for agents. This fits the requirements perfectly. Operating system processes bring the best security and robustness. The system can be set up in a way that the agents cannot reach each other directly, only via the communication platform. If one of the agents crashes, others are not influenced. Untrusted agents can be potentially run in the system.

On the other hand, this solution is not without any drawbacks:

- Even inter-process communication is still less safe than in-process one, not to mention the situation when networking is involved. Making the system distributed inevitably brings a lot of complexity because various pieces can crash, connections can be dropped and so on.
- There is a need for some kind of agent manager and supervisor. It would be a management nightmare to manually manage potentially tens of processes connected to the communication platform. A process supervisor is necessary that would take care of starting and shutting down all the agent components.
- The configuration is also a bit more problematic when the system consists of multiple processes. A good practice is to use environment variables to pass configuration to the agent processes. These environments should be managed by the agent supervisor.

To make the overall idea clear, we need to think about the communication platform and the agent supervisor as forming a platform provided as a service to the agents. This agent-hosting platform is described in the following list, which is rather detailed and also contains some implementation details that we believe must be clearly stated here:

- Every agent is represented by a source code repository, which contains the agent program sources. An agent can be installed into the system by inserting the repository address.

- The agent repository must contain scripts defining how to assemble the executable representing the agent so that the supervisor can start the agent.
- Agents are configured through environment variables. Every agent repository contains a configuration file containing the definition of the variables that must be set before the agent can be started. To make it clear, the agent will be started only after all the required variables have been set. The way how this is achieved depends on the supervisor implementation. There could be a command line utility for agent management much like the utilities almost every cloud provider provides.
- It is possible to view statuses of all the installed agents.
- Although the supervisor component and the communication platform are independent in a way, every agent is expected to use a client library to plug into the communication platform so that all the platform services are available as functions in the given programming language and all the internals are hidden from the library user, the agent developer. The supervisor should use environment variables to pass necessary data to the client library so that it can auto-configure itself and connect to the platform.
- The supervisor can be requested to stop an agent, in which case a signal is sent to the relevant process. The process is killed if the signal is not processed correctly or fast enough.
- Naturally the agents can be deleted from the system, in which case all the relevant persistent data is erased.

The whole platform as a service is built on foundations similar to many commercial platforms. We are just operating on a much smaller scale, potentially on just a single machine, the main goal being to make the agent management as simple and pleasant for the user as possible.

4.5 Routing Inbound Traffic

Having the system consisting of many small pieces, each of them potentially being a server receiving requests from external sources, it is also necessary to take care of routing these requests to the appropriate agents. It is necessary to implement a dynamic reverse proxy server in case of HTTP, but in general it may be necessary to route any inbound network connection. The listening addresses could be set as special environment variables generated dynamically when the agent is installed and passed into the agents on startup.

This has not been implemented in the prototype system, so a standalone reverse proxy must be deployed and configured manually for now. This is slightly inconvenient, but it is planned to be implemented in the future.

4.6 Communication Platform Implementation

In this section we describe how the ideas presented earlier in this chapter can be put into practice. What needs to be implemented is:

- the communication platform that the agents can use to exchange data,
- the client library that can be used by the agents to plug into the platform,
- the agent supervisor, and
- the command line utility that can be used to command the supervisor.

It is important to understand that the first two points are related purely to the data exchange functionality while the last two points are tied to the agent management. The communication platform is the core component of the system and is naturally always enabled, which, however, does not have to be necessarily the case for the management part. This is closely related to how powerful the agent supervisor is. For the initial system prototype we decided to only implement a supervisor that can handle local processes, i.e. the supervisor cannot control processes running on other hosts. So if the system only consists of agents that are running on different hosts, the agent supervisor functionality can be simply disabled, so it should be treated as an optional system component. This fact will be used later in this chapter.

This section does not mention the new continuous integration server at all since the optimal CI server as proposed in the previous chapter is just a set of agent components talking to each other using the request-reply service. The implementation of the proposed CIS is then just about implementing the agents while the communication between the build master and build slaves is entirely handled by the request-reply service. This will be described later.

4.6.1 The Communication Platform and the Client Library

To understand better what these agent components are and how they can be used, the following list describes how the target workflow functionality can be split into four agent components so that the concerns in the system are nicely separated:

1. The whole process is started by detecting a new patch being uploaded into the code review system, hence there shall be an agent collecting the code review events and emitting them as native platform events using the publish-subscribe subsystem.
2. The code review events shall be processed by another agent, which should be triggering the builds using request-reply. This agent also takes care of saving the build output and publishing it online. When a build is finished, this agent emits the event representing the result and containing the output URL. Traditionally, this agent is called build master.
3. There must be agents processing the build requests and streaming the output back. Traditionally, these agents are called build slaves.
4. The last step of the process is the code review request annotation, so there shall be an agent reacting to the build events and annotating the patches.

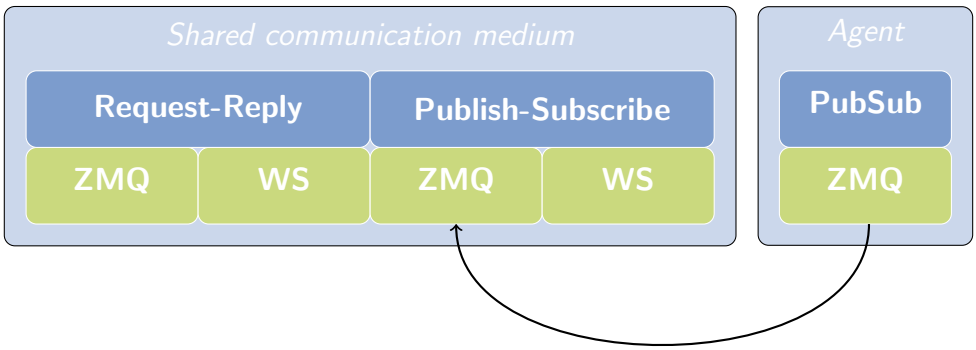
From the implementation point of view, the most important point is to decide what is common to all these agents and what differs. In other words, what functionality to include in the core system and what functionality to split away into modules that can be changed for various scenarios? There are two orthogonal axis that split the communication platform functionality:

Services First of all, every agent can decide what communication pattern it needs. It can be the publish-subscribe subsystem, the request-reply subsystem, or some other subsystem that implements some useful communication functionality useful for multiple agents. We decided to call these components *services*, so there would be the publish-subscribe service or the request-reply (RPC) service.

Service transports Every agent can choose different transport and protocol to be used to connect to the desired service. There might be different transports fitting different scenarios. In case the agents are scattered across Internet, a transport supporting encryption may become handy. On the other hand, if the agents are running on the same machine, some form of inter-process communication can be used.

These observations lead to a layered system architecture as depicted in Figure 4.2. Note that even though it is not visible on the scheme, the platform client library is also split according to the services and transports being used. The difference lies in the fact that while the shared medium must expose all the transports necessary for every service, the agent just chooses a single transport to be used for given service to plug into the platform.

Figure 4.2: The layered architecture of the communication platform



4.6.2 Shared Communication Medium

It has not been decided or said what this shared communication medium depicted in Figure 4.2 actually is, unlike the agent components, which are defined to be operating system processes using the client library to plug into the communication platform. The client library, because of the layered architecture, hides the way the agent is connected to the platform, so theoretically the communication medium can be anything that can be used to interconnect all the agents requesting the same service.

There is immense number of ways how to approach this communication medium. It can be centralized or distributed, it can bring various high availability guarantees and so on. Since we only need to implement the necessary services that can be used by the agents, and there are no other requirements such as the system being fault tolerant, we aim for the simplest solution possible. The main goal of the system is to approach continuous integration and workflow management differently, fault tolerance is really just an implementation detail which we do not discuss here.

So, the simplest way how to implement the medium is again to put all the functionality into a single operating system process and structure the application according to the layered schema presented in Figure 4.2. It must be noted, though, that this broker component is not the process itself, but rather a thread running there. The reson for this is explaned later, in Section 4.7.

Internally the broker component can be split into three parts:

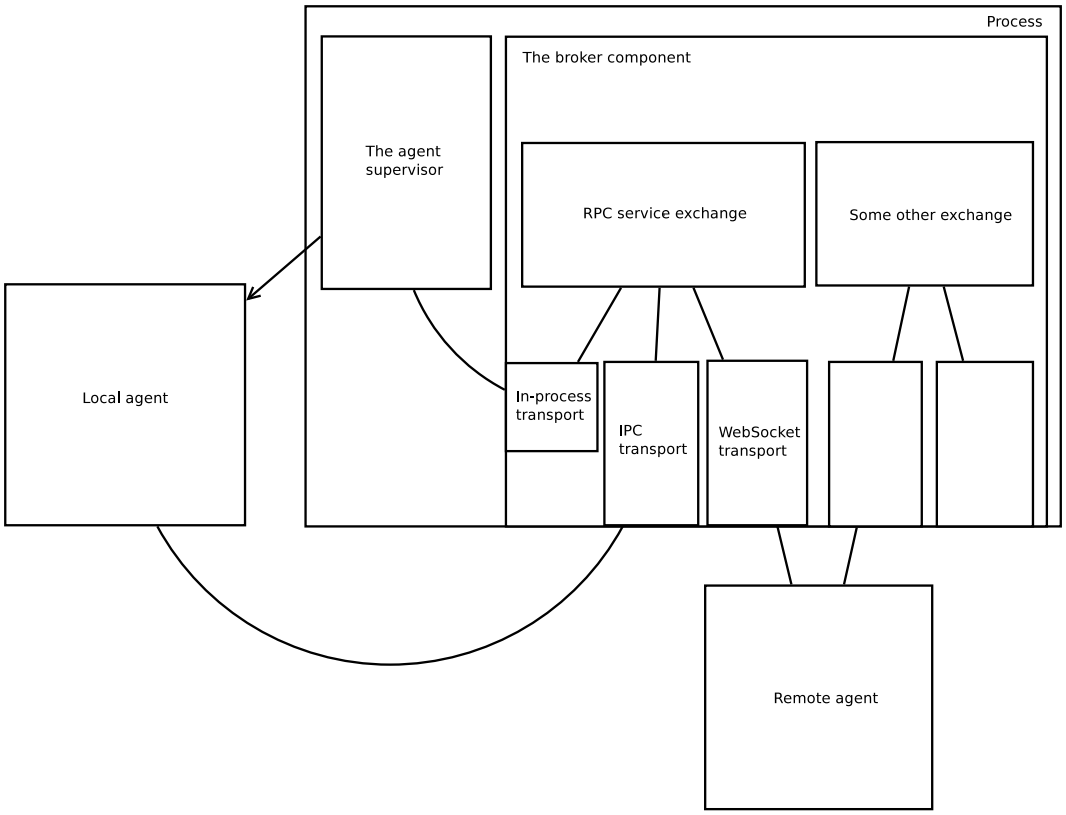
Service exchanges *Service exchange* is a component that represents particular service inside of the broker component and implements the behaviour specific for that particular service. For example, the RPC service exchange would be keeping the mapping of what agent exported what methods, which would be then used as the routing table when a request is received from one of the agents and the exchange is supposed to choose where to forward it. Request-Reply or Publish-Subscribe from Figure 4.2 are the service exchanges.

Service endpoints *Service endpoints* export certain service functionality over particular protocols. The task of the service endpoint is to abstract away particular wire protocol out of the relevant exchange. In Figure 4.2, there is an endpoint exporting publish-subscribe over WebSocket (denoted as WS under Publish-Subscribe), another one doing the same using ZeroMQ as the transport (denoted as ZMQ). The exchange just communicates with all its endpoints over the same service-specific interface, and it knows nothing about how the data is being transported to the agents. It only has to know what agent is reachable through what endpoint.

Endpoints supervisor Service endpoints are started by the *endpoints supervisor* component, which knows actually nothing about the communication happening in the broker component. It is just a supervisor, or container, for the service endpoints. The registered endpoints are started at the broker startup, and they are also restarted in case they crash if that is not happening too often. Following the same logic, stopping the broker means that all the running endpoints are stopped.

What might be surprising is that the real broker component structure does not strictly follow the schema as presented in Figure 4.2. The supervisor runs service endpoints, not services. But this is only there to pull the common functionality out of the services. Instead of having each service run its endpoints, the endpoints are registered with the supervisor object and started at once. The exchanges are then actually just common contexts (data structures) that are shared between the endpoints of the same service.

Figure 4.3: The communication platform



4.7 The Agents Manager Implementation

As mentioned at the beginning of Section 4.6, the agent management functionality should be treated as an optional component which may not be needed for every scenario.

The approach we propose is to implement the agents supervisor as an agent as well and use a special in-process transport to connect to the broker. This has multiple advantages and some interesting implications. Firstly, this makes the system management easier by keeping the number of processes to just one, but it also has some interesting consequences. Clearly it is simple to disable the management component by simply not starting the agent. More interesting is, however, that since the agent supervisor is connected to the broker, it can use all the services available, most importantly the RPC service. Exporting special management RPC methods means that there can be agents in the system that are able to start and stop other agents by calling these methods. There can be a web interface or a command line utility just using these management calls transparently to manage the whole platform.

Such a command line utility was actually implemented. It is just a short-lived agent that connects to the RPC service endpoint, performs a single management call and disconnects. More about the utility can be found in Appendix D.

The complete system architecture, including the special management agent, is depicted in Figure 4.3.

4.8 Summary

We started this chapter by describing on what basis the communication platform shall work and how it can be used to integrate various development tools. Every tool shall be represented by its agent component that facilitates the communication with other components. We came up with how the optimal continuous integration server can be implemented using the given communication primitives, by using agents only.

Analysing the agent components more closely, we described them as operating system processes that use client libraries to plug into the communication platform. A need for a supervisor and manager for these processes arose, which together with the communication platform forms a specialized platform that is provided as a service to the agents much like various cloud providers provide their application platforms as a service to their customers.

The optimal communication platform architecture was proposed. It is rather simple, but that is actually its greatest strength. The service exchanges represent the communication patterns we found necessary, but it can be any service that all agents can benefit from. The endpoints associated with the exchanges are just objects implementing particular interface that the relevant exchange requires, but what the endpoints really do is encapsulated and completely hidden from the exchanges. The actual connection between the endpoint and its associated agents can be in-process, inter-process or inter-host and using all sorts of custom protocols.

The fact that the endpoints hide what the connected agents really are is very powerful and it can be used to implement both the agent supervisor and the continuous integration build slaves using the same mechanism. For the supervisor, all that is necessary is to create an agent using an in-process transport and run the agent in the same process as the broker component, making it export various management calls via the RPC service. Various agent management interfaces can be implemented, using the same RPC mechanism to manage installed agents. There can be a web application or a command line utility and they can both connect to the broker to execute management calls since the functionality is simply exported under certain method names.

In the next chapter we describe how we implemented a real development workflow using the prototype system implementation and we evaluate the results.

5. Prototype Implementation

In the previous chapter we proposed how to design and implement the optimal communication platform and CI server so that the solution is flexible and can be used to integrate development tools in an easy and manageable way. In this chapter we describe how these two components can be used to implement a real workflow and we describe the system implementation in more details. The chapter ends with a simple performance evaluation.

5.1 Target Workflow

The workflow is very similar to the exemplar scenario that is described in Section 2.1, it just uses a real code hosting service, namely GitHub. The goal is to use the system prototype to build, test and annotate pull requests on GitHub. It is a three-step process:

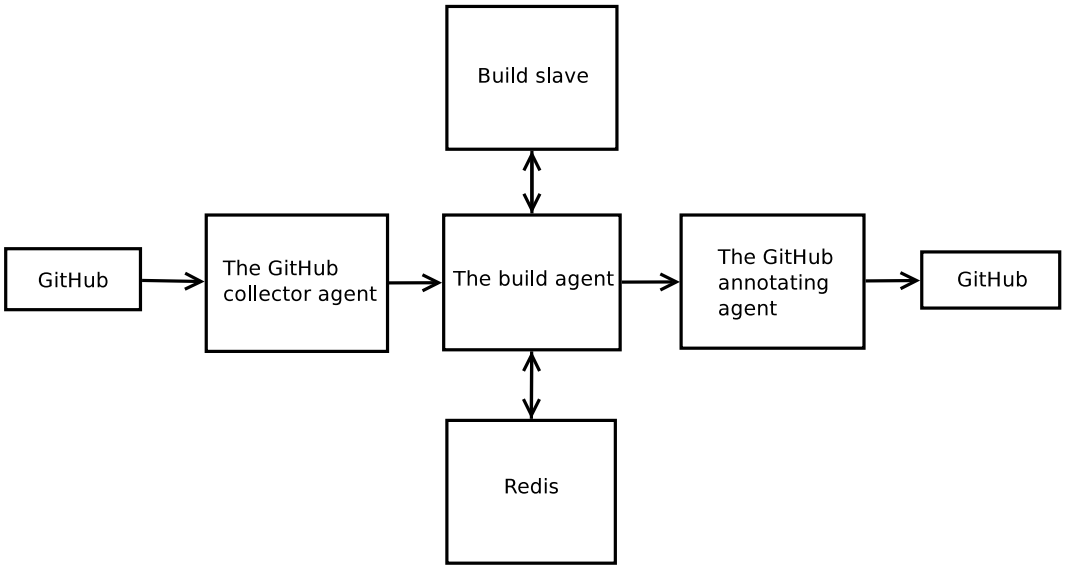
1. collect the relevant GitHub pull request events,
2. build and test the change introduced by the pull request, and
3. annotate the pull request with the build result.

5.2 Target Workflow Implementation

To separate the three concerns mentioned above, the functionality is split into three agents accordingly (illustrated in Figure 5.1):

- The first agent is a simple one. It just receives webhooks from GitHub and converts them into native publish-subscribe service events. The events are labeled as `github.pull_request`. To split this functionality into a separate agent makes sense since we cannot know in advance what other agents are interested in these GitHub events. In our case it is just a single agent, but this way makes it possible to add other subscribers later. The publishing code is also in just a single instance in the whole system.
- The second agent is the most complex one. Its purpose is to generate build requests and save the build output. As mentioned before, all necessary data is saved in the repository, in this case in a YAML file. The file specifies the testing environment (which is a string label), what script to run in this environment (relative to the repository root) and what additional environment variables to set for the script. This step naturally uses the project revision that matches the pull request. Once the configuration file is fetched from GitHub, it is used to create a build request. The request is then dispatched to the CI server subsystem. Since the CI server is implemented on top of the RPC service of the communication platform, this basically means that the right method name matching the requested environment must be generated and the method must be called with the right arguments. As the build is proceeding, the output is being streamed into this agent and

Figure 5.1: The target workflow implementation



it is buffered until the build is finished. Then the output is saved into the database (Redis) and the build result is emitted as an event. The event contains a URL that can be used to access the build output online. For this purpose a simple web server is running inside of this agent to serve the output from the database.

- The last agent is very simple again. It annotates the relevant pull request according to the build result. The build output URL is included in the annotation so that people can check the output in case the build failed. The build status functionality is split into a separate agent again because other agents in the system can be interested in the build result events, for example to send emails to the developers who broke the build.

5.3 Continuous Integration Server Prototype

To be able to use the system and develop custom agents, one must understand clearly what components build up the whole platform and how they communicate. The point of this section is to describe the prototype and explain how it can be used.

From the highest perspective there are two projects involved:

- the communication framework, code name *Meeko*, and
- the continuous integration server, code name *Cider*.

Meeko contains the communication platform, i.e. the broker and the agent supervisor components as well as the client libraries for various programming languages that are to be used to implement agent components. A command line utility to control the agents running on Meeko is also included in the project.

Cider is the continuous integration server built on top of Meeko.

Both projects were implemented using the Go programming language [5]. This language was chosen for its great support for concurrent programming and programming against interfaces.

Now to understand how these two projects are related to each other exactly, the following list contains all the executable programs that are included in the prototype (two-word names are subcommands of the executable denoted by the first word):

meekod This program is the communication platform and it contains the communication broker and the agent supervisor, which is an agent itself, just using a special in-process transport to connect to the broker. The usage is explained and documented in Appendix C.

meeko This is the command line utility for managing agents. It is a short-lived agent that connects to the broker and sends a single management request, then disconnects and exits. The usage is explained and documented in Appendix D.

cider master This subcommand of the CI server executable actually contains the same functionality as meekod, it is there just for comfort so that a single executable can be be used to start all the CI server components. See Appendix E for more information.

cider slave The build slave component can be started by invoking this subcommand. It accepts various parameters, for example the address where the RPC service WebSocket endpoint is listening, or what the access token is. See Appendix E for more information.

cider build A build request can be sent by invoking this subcommand. It is mainly for testing purposes so that the CI server users can trigger builds from their consoles and see the output being streamed back. See Appendix E for more information.

Now there is a single Git repository for every executable, but the common functionality is factored out into a few libraries:

meeko This repository contains the packages implementing the broker component and the supervisor component. The broker component is further split into multiple packages representing various services and their endpoints. It is used in `meekod` and `cider master`.

go-meeko This is the Go programming language client library. It is as well split into packages implementing various services and their transports. This library is being used in `meeko`, `cider slave` and `cider build`.

More about Cider administration can be found in Appendix E. How to write Meeko agents using go-meeko is explained in Appendix F.

5.4 Meeko Services

Meeko follows the proposed layered architecture of service exchanges and endpoints. To make the system usable, three services were implemented:

PubSub This is the publish-subscribe service already mentioned many times. The agents can subscribe for various event kinds that other agents emit.

RPC This is the service that Cider is heavily depending on. It enables agents to export functions identified by a string name that other agents can call.

Logging The last service is mainly for management purposes. It can and should be used by the agents for storing log records. The records are streamed into the broker where they can be stored and introspected centrally in a unified way.

Let us discuss each of these services separately.

5.4.1 PubSub

The PubSub service takes care of distributing events among interested parties. Every event consists of two parts - the event kind and the event object itself. The event kind is just a string identifying the event type and it must be specified for every event because that is the identifier that other agents use to subscribe for events. Every agent can publish events by calling relevant service client method, and subscribe for events emitted by other agents by specifying the event kind prefix and the callback that is to be used to process incoming events matching the prefix. That means that if the event kind of `github` is specified, the client will subscribe for `github.push`, `github.pull_request` as well as all other events matching the prefix and the same callback will be used to handle the incoming events.

From the nature of publish-subscribe, this service works in the fire and forget way. The producers just emit events and the system takes care of delivering the copies to all the interested parties. To prevent a slow consumer blocking the whole system it can happen that a slow agent misses some events in extreme cases. It can detect that it missed some events since every event has a sequence number associated with it that is incremented on every emit of that particular event kind. During the subscribe action, the client receives the current list of sequence numbers matching the event kind prefix that it is subscribing for, then it keeps checking that no gaps were detected, and if there are any, it can emit errors if it is configured to do so. This way of implementing publish-subscribe is not generally very scalable because it is necessary to keep the sequence numbers on a single place, but since there is the broker component right now that is shared, it can be used for this purpose for now.

5.4.2 RPC

The RPC service makes it possible for one agent to export some functionality as methods that are given names. Other agents connected to the same Meeko instance can then call these methods remotely.

The RPC implementation offers some advanced functionality. It is not just the regular request-response pattern:

- The implementation is fully asynchronous. No need to wait for the response before sending another request.
- Standard and error output can be streamed life from the agent handling the request to the requester. This is very useful in case a user is supposed to see some progress. This is actually what `meeko` uses when calling one of the management methods. Meeko itself is sending the output to the console, `meeko` knows nothing about the data that is actually shown to the user.
- The handling agent can send progress signals to the requester. These can be used to just signal that the handling agent is still alive, but in general they can be used for anything. The semantics must be defined in every method's documentation.
- The requester can interrupt the method handler by sending an interrupt. It depends on the method how the signal is processed, but in general the handler should be listening for this signal and terminate as soon as possible. In case the handler spawns a process, it should terminate the process and return.

So, the typical RPC call goes through the following phrases:

1. An agent registers a method under certain name with the broker. This happens by giving the method name and implementation to the client library. The library then sends a message to the broker, which registers the method in its routing table. To be precise, the RPC exchange registers the method in its routing table. From now on other agents can call this method remotely.
2. An agent decides to call a registered method. It generates the RPC method name and arguments and passes these to the client library, which takes care of all the work. The requester can also enable life streaming and progress signals. Once the request is sent, the requester can either keep doing something or just block until the response is available.
3. The request is forwarded to the broker, which chooses what agent will take care of the request. Right now the routing is handled in the round-robin way, so if there are multiple agents exporting the same method, the exchange just rotates around the list of available receiving agents. Once the handling agent is chosen, the request is forwarded, which means passing the request to the right service transport endpoint that the agent is connected to.
4. The registered method handler is invoked in the handling agent by the client library. Once finished (which can happen as a result of the interrupt signal), the request is resolved with a return code and a return value. Both is passed back to the requester which can check the return code and use the return value for its purposes. Zero return code generally means success, other return codes are method-specific and must be documented. The return value is a single MessagePack-encoded object, but the client libraries should takes care of marshalling automatically.

5.4.3 Logging

The Logging service takes care of collecting log records from the agents so that the records can be viewed or watched as they are being sent or saved for later introspection in case something goes wrong. Every log record is assigned a log level, which can be one of `trace`, `debug`, `info`, `warn`, `error` or `critical`.

This service is a must since there is not really any other way how to administer logs in a clear way. To have every agent saving the logs into its workspace is not the right solution since every agent could use a different location and format. Having the log management central also allows for some interesting use cases, like a web application allowing to browse the logs or watch the log records as they are happening. `meeko watch` subcommand uses the Logging service to stream logs into the console.

Concerning `meeko watch`, it is actually the exchange that allows for log streaming. The service itself defines only the interface for uploading log records in the direction from the agents. The service exchange then implements what is supposed to happen with the log records, e.g. they can be persisted somehow. The default exchange is not persisting log records, but allows to subscribe for particular agent's log stream much like the agents can subscribe for events using the PubSub service.

5.5 Cider Internals

Cider is a continuous integration server built on top of Meeko. The build master node is actually the Meeko broker, with the WebSocket RPC transport enabled. The transport is secured by requiring an access token to be specified during the WebSocket handshake. The build slaves are then implemented as Meeko agents, using the WebSocket transport to connect to the build master. They are expected to run on different hosts than the master itself, hence the access token.

After connecting to the build master, every build slave exports certain RPC methods that can be called by other agents connected to the same Meeko instance to trigger a build on that particular build slave. The method name consists of

- A common constant application prefix, here it is `cider`.
- The environment label. Slave labels are specified manually for every build slave and they denote the environment the slave represents. It can be `windows7`, `macosx`, `linux` and so on.
- A build *runner* that is available on the given build slave. Build runners are in general programs that can run scripts, like Shell on Unixes and PowerShell on Windows. The runners are not configured manually, they are detected by the build slave on start. There are four implemented runners: `bash`, `node`, `powershell` and `cmd`.

The final list of RPC methods is generated by doing Cartesian product on the set of labels and available runners. The set of labels always includes `any`. The RPC methods exported are then in the form of `cider.LABEL.RUNNER`, for example `cider.windows7.powershell`. So the meaning of these RPC methods

is that the build slave is requested to run certain script on the host where it is running, that is why there is the runner part of the method name. The method name contains enough information to find not only the right build slave, but also the right program to use to run the chosen script. But actually the runner part of the method name is there to prevent invalid combinations, like running a PowerShell script on a Linux build slave. It is making the routing part trivial.

Anyway, concerning the build requests, there are a few additional arguments that are necessary, namely:

- the source code repository that is hosting the script to be triggered and that is expected to host the project sources that are to be tested,
- the relative path of the script within the repository, and
- the list of environment variables that will be defined during the script execution.

Now the information that the system needs to run the build is complete. The request can be routed to the right build slave, which then knows what repository to download, what script to run and what environment to set for the script.

5.6 Testing Projects Using Cider

Even though `cider build` makes it possible to specify all necessary build configuration on the command line, there is still need for keeping the relevant data in the repository to be able to implement the chosen workflow. The builds are happening on the server asynchronously when GitHub pull requests are opened. For this purpose there must be a special file present in the top-level directory of the project repository, which is called `cider.yml`. To make the life of the developers easier, `cider build` also understands this file and always check for it in the current working directory. This makes it possible to omit large number of command line flags in case the project is already set up to include `cider.yml`.

The configuration file is described together with other configuration options in Appendix E.

5.7 Benchmarking Meeko and Cider

We have implemented a simple benchmark to see if Cider and Meeko perform well enough to be able to handle multiple concurrent builds at once, streaming all the build output to the requesters at the same time.

The benchmark runs the build master component and a single build slave locally in their own threads within the benchmark process, connecting them via WebSocket using a local TCP connection. The `testing` package from the Go standard library is being used to run the benchmark.

The benchmark can be run in multiple modes. The modes enable different subsets of what is happening during the build process to be able to see what slows down the system the most. The available modes are:

noop Do not clone the repository, do not run any script, just return success.

noop+stream Do not clone the repository, do not run any script, but stream some non-trivial output back to the requester.

discard Clone the repository, run the script, but do not stream the output back.

stream Clone the repository, run the script and stream the output back.

redis Clone the repository, run the script, buffer the output, then save it into Redis when the build is finished.

files Clone the repository, run the script, buffer the output, then save in into a file on the disk.

The number of OS threads that Go runtime uses internally can be also set for the benchmark, the default being just one thread. This does not mean that only a single Go thread (goroutine) can run and block other threads, it just means that all the Go threads share a single OS thread. Go does asynchronous I/O in the background.

Before every benchmark run, a Git repository is created in a temporary directory. This repository contains a generated Bash script that just prints 10 000 lines of text (each 21 characters long) and exits. This repository is used as the project repository that is cloned during the builds. It is deleted again when the benchmark is finished. Since the build slave uses the Git branch name to generate local directory path where to clone the repository and checkout the branch, enough branches are created in the repository so that there are no clashes during the benchmark. In this way, all the builds can run concurrently without locking each other out.

5.7.1 Benchmark Results

The benchmark was run on MacBook Pro (early 2011), CPU 2.7 GHz Intel Core i7 (4 CPUs), 4 GB 1333 MHz DDR3 RAM, disk 5400 rpm, running Mac OS X Lion 10.7.5. Every mode was run 5 times, all the collected data is listed in Table 5.1 - Table 5.12. The first output column shows how many requests were fired, the second column shows how much time it took to complete a single build on average.

Table 5.1: Benchmark: 1 thread, noop mode

10000	116141 ns/op
10000	118318 ns/op
10000	117466 ns/op
10000	117672 ns/op
10000	119870 ns/op
avg	117893 ns/op

Table 5.2: Benchmark: 1 thread, noop+stream mode

500	5568584 ns/op
500	5656427 ns/op
500	5588431 ns/op
500	5683269 ns/op
500	5536331 ns/op
avg	5606608 ns/op

Table 5.3: Benchmark: 1 thread, discard mode

50	66049025 ns/op
20	124588994 ns/op
50	72269828 ns/op
20	83775819 ns/op
10	134034625 ns/op
avg	96143658 ns/op

Table 5.4: Benchmark: 1 thread, stream mode

50	55566140 ns/op
20	50753613 ns/op
10	106343609 ns/op
20	87325214 ns/op
50	67271365 ns/op
avg	73451988 ns/op

Table 5.5: Benchmark: 1 thread, redis mode

50	91469290 ns/op
50	89834287 ns/op
50	74857127 ns/op
50	61327816 ns/op
50	56668844 ns/op
avg	74831472 ns/op

Table 5.6: Benchmark: 1 thread, files mode

10	105338329 ns/op
10	118421312 ns/op
50	59483523 ns/op
50	64875806 ns/op
10	120911117 ns/op
avg	93806017 ns/op

Table 5.7: Benchmark: 4 threads, noop mode

20000	73202 ns/op
20000	71841 ns/op
20000	72978 ns/op
20000	72959 ns/op
20000	73456 ns/op
avg	72887 ns/op

Table 5.8: Benchmark: 4 threads, noop+stream mode

500	3982016 ns/op
500	3989970 ns/op
500	4028983 ns/op
500	4000035 ns/op
500	3987918 ns/op
avg	3997784 ns/op

Table 5.9: Benchmark: 4 threads, discard mode

50	60953392 ns/op
50	74873721 ns/op
50	80092564 ns/op
20	68473920 ns/op
20	61124403 ns/op
avg	69103600 ns/op

Table 5.10: Benchmark: 4 threads, stream mode

20	89590665 ns/op
50	62025025 ns/op
50	65817256 ns/op
50	73712222 ns/op
20	90325548 ns/op
avg	76294143 ns/op

Table 5.11: Benchmark: 4 threads, redis mode

50	67857096 ns/op
50	69941030 ns/op
50	71235698 ns/op
50	58204349 ns/op
50	68735020 ns/op
avg	67194638 ns/op

Table 5.12: Benchmark: 4 threads, files mode

10	106588232 ns/op
20	91778911 ns/op
20	70745534 ns/op
20	92100082 ns/op
10	141744087 ns/op
avg	100591369 ns/op

Table 5.13: Benchmark results summary

benchmark mode	threads	builds per second
noop	1	8482.23
noop	4	13719.88
noop+stream	1	178.36
noop+stream	4	250.14
discard	1	10.40
discard	4	14.47
stream	1	13.61
stream	4	13.11
redis	1	13.36
redis	4	14.88
files	1	10.66
files	4	9.94

The results are summarized in Table 5.13. There are two major declines visible in the table, one is from noop to noop+stream, the other one is from noop+stream to discard and in general all other modes. This can be easily explained.

When we move from noop to noop+stream, it means that the repository is still not being cloned, no script is being executed, but each build generates 200 000 visible characters (in this case 210 000 bytes when we include the newline characters). Counting 250 builds per second, that is 50 MB of output streamed every second.

Changing from any noop mode to a mode involving cloning of a Git repository means that the disk must be accessed quite a lot. On top of it, the build slave and the repository resided on the same physical disk during this benchmark. All the cloning modes also start the build script, which means that a new Bash process is spawned for every build request.

The differences between the modes that are cloning are not clear. One would expect the discard mode to be definitely faster than the files mode, but that has not been measured. Saving the data into Redis also does not bring any measurable overhead compared to the stream mode. This means that the initial operation of cloning and starting a new process that happens at the beginning of every build is so expensive that in the end it does not matter what happens with the output.

Considering the differences implied by the number of threads Go runtime is allowed to use under the hood, we can see that the noop modes are gaining the most from multi-threading. This is because these modes are CPU bound while the other modes are I/O bound and the disk is the limiting factor there.

5.8 Summary

In this chapter we presented how the prototype communication platform and CI server were implemented and then used to handle a workflow incorporating real development tools. We split the functionality into multiple agents, each implementing part of the requested functionality. This leads into very desirable separation of concerts where very little code has to be duplicated between the agents. If there are agents that need to integrate with particular system, it is enough to implement a single agent collecting the data. All other agents can then access the data using the communication platform and they do not care how the data was inserted.

In the second part of this chapter we showed the results of a simple local benchmark. Running locally, the communication platform could handle around 50 MB of data per second over the WebSocket transport. This is definitely enough for Cider to work without any problems. In a production deployment, the cost of all the communication would be more expensive due to the build slaves being scattered across multiple hosts. The build scripts would also spend most of the time doing some useful work, not just generating a lot of build output. In a real deployment, the network and the disks will become bottlenecks much faster than the communication platform logic itself.

6. Conclusion

The initial goal of this thesis was to implement a continuous integration server that would be easy to integrate with other software development tools being used. We proposed and implemented a general communication platform that can be used to integrate any number of development tools in a scalable way from the administrative point of view [6]. Pieces of functionality that do not need to be coupled together are represented by standalone processes, which can be managed independently. We then used this communication platform to implement the desired continuous integration server.

6.1 Logical System Architecture

The way of integrating multiple development tools as proposed in this thesis turned out to work well. Splitting independent functionality into various agent processes not only makes the system scale from the administrative perspective, but it also supports good practices when planning how to integrate chosen systems. The agent components are just black boxes and it does not matter what programming language they are written in or what libraries they use internally. The only thing that matters is the interface they export to be accessible by other agents. The agents can be even replaced independently and the remaining agents won't even notice provided the interface remains the same.

6.2 Communication Platfrom Flexibility

The layered system architecture turned out to be very flexible, allowing for various scenarios to be addressed easily. In our case, the development tools agents together with the agent supervisor as one scenario, and the continuous integration system as another use case, these were both implemented on top of the communication platform without any issues. The immense number of combinations of various communication patterns (services) together with various transport protocols (service transports) allowed us to spread agents taking care of various aspects of the whole system across processes, hosts, but also within a single process using a special in-process transport. Abstracting the communication patterns into standalone services also makes it simple to add new services in the future without touching existing code base.

6.3 Ease of Use and Administration

Having the system consisting of multiple independent agents directly leads to good development practices. Every agent effectively works as a black box that may but does not have to be exporting some functionality for other agents to use. Designing a system on top of this platform then boils down to designing the interfaces, the way how the agents are going to communicate.

From the administrative point of view, the agents can be handled independently. It is possible to add, configure, start, stop and remove agents separately

using the command line utility implemented as a part of the management subsystem (as long as these agents run on the same host as the platform broker process).

6.4 Chosen Programming Language

The chosen system architecture was easily implementable using Go as the programming language since interfaces are first-class citizens there. All that was necessary was to define interfaces for all the services and then implement necessary exchanges and endpoints accordingly.

6.5 Robustness

Splitting the agents into processes turned out to be a robust solution, especially when the system is extended to run agents sandboxed. In this way the agents cannot harm each other, one can receive a segmentation violation signal and be killed while another component is still running fine. Using agent processes, the system can be also extended and potentially scaled at run time since the agents are really managed separately. This all makes it possible to offer the communication platform as a hosted service in the future.

6.6 Performance

The benchmark presented in Section 5.7.1 clearly showed that Meeko is capable of distributing the data between interested agents fast enough (the simple benchmark showed 50 MB/s). Also it must be noted that the benchmark used the WebSocket transport, which is not optimal for inter-process communication. When using Meeko to integrate development tools, the agents are all run locally and supervised by Meeko. A more fitting transport can be then used for that use case, e.g. Unix domain sockets. That would speed up the communication a bit.

6.7 Limitations

We have not found any real limitations of the system as of now that would prevent it from handling the scenarios it was built for. The implementation is, however, very young and it needs further care to be stabilized and ready for production. The agent supervisor is only capable or running agents on the same host and the reverse proxy component to route inbound connections automatically is missing.

6.8 Future Work

Considering the communication platform, the core system has been implemented, but it needs further care to be ready for production. Also only very basic service exchanges have been implemented and they need to be improved and extended for production use as well.

From the platform agents perspective, only the client library for Go exists as of now. To give the platform users more freedom, more client libraries should be implemented for various programming languages.

The prototype continuous integration server implemented as a part of this thesis also lacks some functionality for production use. For example there is no administration and monitoring interface really. It is not possible to see what build slaves are connected.

All these issues will be addressed in the future releases of Meeko and Cider.

Bibliography

[1] Brad Rydzewski. *Drone Documentation*. URL: http://drone.readthedocs.org/en/latest/ (visited on 04/24/2014).

[2] Patrick Th. Eugster et al. *The many faces of Publish/Subscribe*. 2003.

[3] John Ferguson. *Jenkins: The Definitive Guide - Continuous integration for the masses*. O'Reilly, 2011.

[4] Roy T. Fielding and Richard N. Taylor. "Principled Design of the Modern Web Architecture". In: *ACM Transactions on Internet Technology (TOIT)* 2.2 (May 2002), pp. 115–150.

[5] Golang.org. *The Go Programming Language*. May 2014. URL: http://golang.org (visited on 05/19/2014).

[6] Ondrej Kupka and Filip Zavoral. "Cider - An Event-driven Continuous Integration Server". In: *COMPSAC 2014 : IEEE 38th Annual International Computers, Software & Applications Conference* (2014). to appear.

[7] Martin Fowler. *Continuous Integration*. May 2006. URL: http://martinfowler.com/articles/continuousIntegration.html (visited on 02/10/2014).

[8] Buildbot Team Members. *Buildbot - Introduction*. URL: http://docs.buildbot.net/0.8.8/manual/introduction.html (visited on 02/10/2014).

[9] Ondrej Kupka. *Choosing a Build Server for a Small Business*. Jan. 2013. URL: http://tchap.wikidot.com/blog:1 (visited on 02/10/2014).

[10] Travis-CI.org. *Travis CI Documentation*. 2014. URL: http://docs.travis-ci.com (visited on 04/24/2014).

[11] Brian Warner and Dustin J. Mitchell. *Buildbot - The Continuous Integration Framework*. 2014. URL: http://buildbot.net (visited on 02/10/2014).

[12] Wikipedia. *Webhook — Wikipedia, The Free Encyclopedia*. URL: http://http://en.wikipedia.org/wiki/Webhook (visited on 04/24/2014).

[13] Miguel G. Xavier et al. "Performance Evaluation of Container-based Virtualization for High Performance Computing Environments". In: *21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing* (2013), pp. 233–240.

Appendices

A. Contents of the Attached CD

The enclosed CD contains the source code of Meeko, Cider, the workflow implementation and the thesis itself. The directory structure is following:

```
CD                                  the CD root directory
  /thesis                           the thesis sources
    /src                            the LaTeX sources
    /img                            the images
    /build                          the generated thesis PDF
  /src                              the programs sources
    /github.com
      /meeko                        Meeko PaaS
        /meekod                     Meeko executable
          /broker                   the broker component
            /services               the service definitions
            /exchanges              service exchanges
            /transports             service endpoints
          /supervisor               the supervisor component
            /implementations
              /exec                 the processes supervisor
          /daemon                   meekod and cider master logic
        /meeko                      the command line utility
        /go-meeko                   the Go client library
          /meeko
            /services               various service clients
            /transports             service transports
          /agent                    helper for writing agents
      /meeko-contrib
        /meeko-collector-github     the workflow collector agent
      /cider
        /cider                      Cider executable
          /master                   the build master subcommand
          /slave                    the build slave subcommand
          /build                    the build trigger subcommand
          /benchmark                the benchmark executable
        /cider-example              a Cider-compatible repo
      /cider-plugins
        /cider-github-trigger       the workflow build agent
        /cider-github-status        the workflow status agent
```

The source code of all Meeko and Cider packages as well as the required 3rd-party packages can be found in `/src`. Since all the programs are written in the Go programming language, Appendix B describes how to build Go packages.

The source code is hosted on GitHub and can be viewed online. The structure is the same as mentioned above, i.e. the sources are placed under the following accounts: `meeko`, `meeko-contrib`, `cider` and `cider-plugins`.

Generated source code documentation can be viewed using the `godoc` tool that comes with the Go distribution, it is not present on the CD. All that is necessary is to set `GOPATH` as described in Appendix B, then run

```
$ godoc -http=localhost:6060
```

and open the web browser and navigate to `localhost:6060`. This is the idiomatic way how to browse generated documentation in Go. The documentation can be also browsed online using `godoc.org`. All that is necessary is to insert the package import path, e.g. `github.com/meeko/go-meeko`.

B. Building Go Packages

The programs attached on the CD are all written in Go and have been tested to build successfully against Go 1.2. Having the right Go compiler installed is the only requirement, all necessary packages are present on the CD in `/src`. The compiler version can be checked by running `go version`.

B.1 Go Workspaces

Every Go project lives in a workspace, which is a directory having particular directory structure. To tell Go where the current active workspace is, `GOPATH` environment variable must be set to point to the chosen directory. In every Go workspace there are three subdirectories:

- The `src` subdirectory contains all necessary Go packages, which incorporates the project packages and 3rd-party dependencies. When a package is present in `$GOPATH/src/a/b/c`, its import path within Go is `"a/b/c"`.
- The `pkg` subdirectory contains the packages compiled into object files.
- The `bin` subdirectory contains installed executable packages. Any executable package can be installed by running `go install` in its main directory. It is practical to add `$GOPATH/bin` into `PATH`.

When `GOPATH` is set properly, `go build` can be invoked in a package directory to build the package, or `go install` can be invoked to install the package. Go creates statically-linked executables, so the executables can be just copied to another machine and it will work out of the box, provided it is the same architecture and operating system.

B.2 Building the Attached Packages

The `/src` directory on the attached CD is actually exactly the `src` directory that is a part of every Go workspace. Since every Go workspace must be placed on a writable medium, the directory must be copied out of the CD onto the hard drive. Then `GOPATH` must be set and the packages can be assembled using the `go` command.

The CD contains the following executable packages:

- `/src/github.com/meeko/meekod` is the integration platform.
- `/src/github.com/meeko/meeko` is the management utility.
- `/src/github.com/cider/cider` is the CI server.
- `/src/github.com/cider/cider/benchmark` is the CI server benchmark.

There are also some agent packages, which are executable as well, but these are not supposed to be run by the user manually. Cider takes care of compiling and running them.

C. The meekod Command Usage

`meekod` represents Meeko PaaS and can be used to start the broker components together with the agent supervisor component in a single process. Although it is written in Go, which is multiplatform, only Unix-like operating systems are supported as of now.

C.1 Configuration File

The only way how to configure meekod right now is to use a configuration file, then point meekod to it using `-config` command line flag. It is planned to also support configuration using environment variables.

The configuration file is a YAML file with the following structure:

```
broker:           // the broker component
  endpoints:
    rpc:          // the RPC service endpoints
      zeromq:     // ZeroMQ endpoint string for the ROUTER socket
      websocket:  // WebSocket endpoint
        address:  // network address to bind (HOST:PORT)
        token:    // access token to authenticate remote agents
        heartbeat_period: // must be Go time.Duration string
    pubsub:       // the PubSub service endpoints
      zeromq:     // ZeroMQ endpoint
        router:   // ZeroMQ endpoint string for the ROUTER socket
        pub:      // ZeroMQ endpoint string for the PUB socket
    logging:      // the Logging service endpoints
      zeromq:     // ZeroMQ endpoint string for the PULL socket
supervisor:       // the agent supervisor component
  workspace:      // the directory to use for cloning agent sources
  mongodb_url:    // the URL of a MongoDB instance;
                  // format [user:pwd@]host[:port][/database]
  token:          // the management token
```

What must be present in the configuration file is governed by the flags that are passed to meekod:

-disable_supervisor Do not start the agent supervisor.

-disable_ipc Do not initialise the ZeroMQ service endpoints that are being used for local communication.

Unless the supervisor is disabled, the supervisor section must be present in the configuration file. MongoDB is being used by the supervisor to keep agent data, so it must be installed in case the supervisor is enabled. Git is also required for cloning of the agent repositories.

Unless the local inter-process communication is disabled, all the `zeromq` keys must be set to the relevant endpoint strings.

An example of the meekod configuration file can be found in the meekod source directory.

C.2 Environment Variables

MEEKOD_CONFIG can be used instead of the -config flag

MEEKOD_DISABLE_SUPERVISOR can be used instead of the -disable_supervisor flag (any non-empty value evaluates to true)

MEEKOD_DISABLE_LOCAL_ENDPOINTS can be used instead of the -disable_ipc flag (any non-empty value evaluates to true)

MEEKOD_VERBOSE can be used instead of the -verbose flag (any non-empty value evaluates to true)

D. The meeko Command Usage

Meeko agents are managed using some special built-in RPC calls over the RPC service. The agent exporting these management methods actually resides in the same process as the broker component (at least concerning `meekod`) and it uses a special in-process transport to connect to the RPC service. All the management calls require a token that is always passed as one of the arguments so that not every agent connected to the RPC service can execute these management calls.

A command line utility that can be used to execute the management methods can be found in `/src/github.com/meeko/meeko` on the attached CD. It is simply a wrapped Meeko agent that connects to the broker, executes a single management call and exits. Each management call is implemented as a subcommand of `meeko`.

Before describing the available management calls, it should be noted that the utility expects a YAML configuration file in the user's home directory called `.mkrc`. In case the user needs to connect to multiple Meeko instances, the configuration file path can be specified using the `-config` command line flag. The WebSocket transport is used to connected to the RPC service so that the connection can be established to a remote host. The configuration file is structured in the following way:

```
endpoint_address: // URL of the endpoint, starting with ws(s)://.
                  // Both host and port must be specified.
access_token:     // the WebSocket endpoint access token
management_token: // the Meeko supervisor management token
```

What follows is the list of available management commands. Every command corresponds to certain `meeko` subcommand. `install` can be executed by running `meeko install`. `meeko` exits with a non-zero exit status if the RPC call fails for some reason. More detailed subcommand help including the available command line flags can be viewed by running `meeko <command> -h`.

install Calling `Meeko.Agent.Install` method, this command installs the relevant agent by its repository URL. The URL scheme specifies the source code version system and transport, for example `git+https://` tells Meeko to clone using Git over HTTPS, `git+file://` asks Meeko to clone a local repository. `git+ssh://` is also supported, but the Meeko administrator must make sure Meeko uses an SSH key that can be used to clone requested repositories without any blocking. The user must specify the agent alias that is then used across management calls to identify particular agent instance.

upgrade Calling `Meeko.Agent.Upgrade` method, this command upgrades the requested agents. The agent is specified by its alias. That means that the sources are updates and the agent is restarted.

list Calling `Meeko.Agent.List` method, this command returns the list of installed agents together with some basic info.

remove Calling `Meeko.Agent.Remove` method, this command removes the requested agent from system. The agent is again specified by its alias.

info Calling `Meeko.Agent.Info` method, this command prints the description of the requested agent, including its variable description and the values of the variables as set. Basically it pretty-prints the agent's database record.

env Calling `Meeko.Agent.Env` method, this command prints the current agent environment that will be passed to the agent when it is started.

set Calling `Meeko.Agent.Set` method, this command can be used to set a variable for the chosen agent.

unset Calling `Meeko.Agent.Unset` method, this command unsets certain agent variable.

start Calling `Meeko.Agent.Start` method, this command starts the chosen agent.

stop Calling `Meeko.Agent.Stop` method, this command stops or later kills the chosen agent.

restart Calling `Meeko.Agent.Restart` method, this call basically chains `Meeko.Agent.Stop` and `Meeko.Agent.Start` again.

status Calling `Meeko.Agent.Status` method, this command returns the list of applications with their current statuses, which can be `stopped`, `running`, `killed` or `crashed`. If an agent alias is specified, only that agent's status is returned.

watch Calling `Meeko.Agent.Watch` method, this command streams the agent's Logging service output to the console as the records are emitted by the agent.

D.1 Meeko Usage Example

Having the system set up, what follows is the sequence of commands and output that shows how to install and run the GitHub collector agent that was implemented as a part of this thesis:

```
$ meeko install \\
    git+https://github.com/meeko-contrib/meeko-collector-github \\
    as gh-collector
>>> Creating the agent workspace ... OK
>>> Cloning the agent repository ...
Cloning into '/var/lib/meeko/workspace/gh-collector/_stage'...
<<< OK
>>> Reading agent.json ... OK
>>> Validating agent.json ... OK
>>> Moving files into place ... OK
>>> Running the install hook ...
+ godep go install -v
```

```
github.com/meeko-contrib/go-meeko-webhook-receiver/receiver/server
github.com/meeko/go-meeko/meeko/services
github.com/meeko/go-meeko/meeko/services/logging
github.com/cihub/seelog
github.com/tchap/go-patricia/patricia
github.com/meeko/go-meeko/meeko/services/pubsub
github.com/meeko/go-meeko/meeko/services/rpc
github.com/dmotylev/nutrition
github.com/pebbe/zmq3
github.com/meeko/go-meeko/meeko/transports/zmq3/logging
github.com/meeko/go-meeko/meeko/transports/zmq3/loop
github.com/ugorji/go/codec
github.com/meeko/go-meeko/meeko/utils/codecs
github.com/meeko/go-meeko/meeko/transports/zmq3/pubsub
github.com/meeko/go-meeko/meeko/transports/zmq3/rpc
github.com/meeko/go-meeko/agent
github.com/meeko-contrib/go-meeko-webhook-receiver/receiver
github.com/meeko-contrib/meeko-collector-github/handler
github.com/meeko-contrib/meeko-collector-github
<<< OK
>>> Inserting the agent database record ... OK

Success
$ meeko info gh-collector
Alias:          gh-collector
Name:           meeko-collector-github
Version:        0.0.1
Description:    Meeko collector for GitHub events
Repository:     git+https://github.com/meeko-contrib/(...)
Variables:
                Name:   LISTEN_ADDRESS
                Usage:  TCP network address to listen on; (...)
                Type:   string
                Value:  unset

                Name:   ACCESS_TOKEN
                Usage:  token to be used for hooks authentication
                Type:   string
                Value:  unset

# Listen on localhost because we are behind Nginx.
$ meeko set LISTEN_ADDRESS for gh-collector to localhost:8888

Success
$ meeko set -ask ACCESS_TOKEN for gh-collector
Insert the value of ACCESS_TOKEN:

Success
```

```
$ meeko start -watch gh-collector
>>> Streaming logs for agent gh-collector
[INFO] Logging service initialised
[INFO] PubSub service initialised
[INFO] RPC service initialised
[INFO] Forwarding github.issues
(...)
```

E. The cider Command Usage

This is the executable that can be used for starting all components of the Cider continuous integration server. The functionality is divided into three subcommands:

cider master This subcommand starts the build master. It is actually the same thing as invoking `meekod`, so the way how to configure this subcommand is explained in Appendix C.

cider slave This subcommand starts the build slave. The configuration is explained below.

cider build This subcommand can be used to trigger a build of the chosen project repository. The configuration is explained below.

E.1 Build Slave Configuration

To configure `cider slave`, there are two options right now:

- environment variables
- command line flags

E.1.1 Environment Variables

The following environment variables can be used, their meaning is the same as their flag counterparts as specified in the next subsection:

CIDER_MASTER_URL

CIDER_MASTER_TOKEN

CIDER_SLAVE_IDENTITY

CIDER_SLAVE_LABELS

CIDER_SLAVE_WORKSPACE

E.1.2 Command Line Flags

The following flags are accepted by `cider slave`:

-master the build master WebSocket endpoint URL

-token the build master WebSocket access token

-identity a unique build master identifier

-labels a list of environment labels, comma-separated

-workspace the directory to use for project workspaces

-executors number of jobs to run in parallel

-verbose print more verbose output

-debug print debug output

Flags have the highest priority and overwrite any other configuration.

E.2 Build Trigger Configuration

`cider build` can be used to enqueue a build request in Cider and see the build output being streamed back into the console. It can be also configured using a configuration file, command line flags or environment variables. Since this file is expected to be committed in a repository, the access token cannot be specified in the configuration file. An environment variable or command line flag must be used for that.

E.2.1 Configuration File

It is a YAML file again:

```
master:
  url:          // the master URL, schema and port must be there!
                // example: wss://cider.example.com:443/connect
slave:
  label:        // the chosen environment label,
                // 'any' is always valid and available
repository:
  url:          // the repository URL as in mk install,
                // e.g. git+ssh://git@github.com/foo/bar#branchA
script:
  path:         // relative path to the script that is to be run
  runner:       // the script runner,
                // one of node, cmd, powershell, bash
  env:          // list of env variables that will be set
    - FOO=BAR
    - BAZ=TOOR
```

When `cider build` is run, it is expecting this file named as `cider.yml` to be in the current working directory. The same configuration file is being used by the prototype workflow, it is expecting this file to be placed in the top-level directory of the repository and to be also called `cider.yml`. In case the file is found, all the parameters must be supplied as command line flags or environment variables.

E.2.2 Environment Variables

Where not mentioned explicitly, the meaning of the following variables is the same as their flag counterparts mentioned in the next subsection.

CIDER_MASTER_URL

CIDER_MASTER_TOKEN

CIDER_SLAVE_LABEL

CIDER_REPOSITORY_URL

CIDER_SCRIPT_PATH

CIDER_SCRIPT_RUNNER

CIDER_SCRIPT_ENV_key - equivalent to -env key=...

E.2.3 Command Line Flags

The following flags can be used to overwrite the relevant values from the config file, or set the values in case the config file is not being used:

-master

-token

-slave

-runner

-repository

-script

-env The format is `KEY=VALUE`, can be specified multiple times on the command line.

F. Implementing Custom Meeko Agents

To understand how a Meeko agent can be implemented, the agent life cycle in Meeko must be explained in more details. Every Meeko agent goes through the following phases, or jumps between them:

Installation Meeko exports a management call where it accepts a source code management repository path and it is able to download the agent sources from there. Every agent has a workspace assigned and the repository is downloaded into the `src` subdirectory in the workspace. Once downloaded, Meeko looks for `.meeko/agent.json` within the repository, which is the agent metadata file that describes the agent, most importantly telling Meeko what variables must be set before the agent can be started. The structure of this file is described at the end of this appendix. Meeko then looks for an executable script located at `.meeko/hooks/install` within the repository and executes that script. The goal of the script is to create an executable in `$WORKSPACE/bin` that represents the agent. It can be any executable as long as it is named after the agent, hence the whole process is completely language agnostic. If the script succeeds, Meeko saves the agent metadata into the database, creating the agent record, which is nothing more than `agent.json`, just having the variable values filled it. These values can be set using the management calls and they are exported as environment variables for the agent executable when Meeko is requested to start the agent. That is the only way how an agent can be configured.

Starting Once the agent is installed in Meeko, it can be started, provided that all the required variables are set. These are specified in `agent.json` and Meeko will not start the agent until all the compulsory variables are assigned values. Once the values are specified, the agent can be started. Meeko looks for `$WORKSPACE/bin/AGENT_NAME` and starts that executable, exporting the arguments as environment variables.

Stopping The agent can be stopped, naturally. Currently this means that SIGTERM signal is sent to the agent process. If that is not enough, SIGKILL is sent after certain time period.

Upgrade The agent can be upgraded, which means that the sources are updated and `.meeko/hooks/upgrade` script is run. This script's goal is again to create a new executable placed in the right directory. Then the agent is restarted if running so that the new executable is loaded.

Removal Once the agent is not needed any more, it can be removed from the system. That means that its workspace is deleted and the database record is dropped.

Now that it is more clear how an agent is managed from outside, let us look at the same steps from the agent's point of view:

Installation Every agent is represented by a single source code repository. That repository must include `.meeko` directory in the top level directory. A file called `agent.json` in that directory specifies certain metadata like the agent name, version and arguments. Its structure is defined in the `meekod/supervisor/data` package. Then there are two scripts that are to be placed into `.meeko/hooks`. It is `install` and `upgrade`, which are run during the install and upgrade management commands. Their goal is to create or re-create the executable representing the agent, and place that executable into `$BINDIR/$AGENT` where `BINDIR` is a variable containing the path of the directory where Meeko expects the agent executable to be. It is exported for the hook scripts. `AGENT` is then the name of the agent being installed as inserted in `agent.json`. `cider-plugins/cider-github-status/.meeko` directory on the attached CD contains a real world example. The scripts there, however, are very simple, because the whole mechanism is compatible with how Go commands build and install executables and all the steps happen automatically in case the agent is written in Go.

Starting When Meeko is asked to start the application, it looks for its executable, which is then used to start the process representing the agent. All the compulsory variables from `agent.json` must be set using the management calls before the relevant agent can be started. When the variables are set, Meeko starts the process with all the variables exported as environment variables.

Stopping When Meeko is asked to stop an agent, it first sends SIGTERM to the agent, and if the agent process does not terminate in time, which is now by default 5 seconds, it receives SIGKILL and is terminated immediately.

Upgrade Running upgrade just means that the agent sources are updated, the executable is rebuilt and if everything is fine and the agent is running, it is restarted to start using the new executable.

Besides containing the metadata directory, the agent is of course expected to use the client library to plug into the communication platform. It can implement any kind of functionality, it is just an executable program after all.

F.1 Agent Metadata File

```
{
  "name": string
  "version": string
  "description": string
  "clone_dir": string
  "vars": {
    VARIABLE_NAME: {
      "usage": string
      "type": "string" | "integer" | "float64" |
              "boolean" | "duration"
      "secret": bool
      "optional": bool
    }
  }
}
```

In this schema, `VARIABLE_NAME` represents a variable that will be exported for the agent when `meeko start` is called.

`clone_dir` allows agents to be cloned into a custom subdirectory of their workspace. This is useful particularly for agents written in Go since they often need to be cloned according to the import paths they use, which can be `github.com/foo/bar`.

G. Supported Meeko Service Endpoints and Protocols

The following Meeko service endpoints are available:

- RPC service
  - ZeroMQ 3.x transport
  - WebSocket transport
  - In-process transport
- PubSub service
  - ZeroMQ 3.x transport
  - In-process transport
- Logging service
  - ZeroMQ 3.x transport

Local agents are expected to use the ZeroMQ 3.x IPC transport, necessary environment variables are exported into the agent's environment so that the client library can configure itself automatically.

Remote agents are expected to use the WebSocket transport, which is for now only available for the RPC service so that Cider could be implemented.

We do not document the exact low-level protocols here, because every endpoint uses internally a protocol of its own that is specific to that particular service and endpoint. These protocols are not really important unless the task is to implement a new client library. The agents themselves know nothing about the underlying protocols since they always use a finished and working client library that completely hides all the unnecessary complexity.