ORB ACADEMIC PUBLISHER

Int. J. Comput. Sci. Theor. App., 2015, vol. 3, no. 2., p. 51-59.
Available online at www.orb-academic.org

International Journal of
Computer Science:
Theory and Application
ISSN: 2336-0984

On Architecture and Design of Event-Based Continuous Integration Server

Ondrej Kupka, Filip Zavoral*

Charles University in Prague, Czech Republic
Email: zavoral@ksi.mff.cuni.cz
**Corresponding author.*

ABSTRACT
The lack of proper integration support is apparent in many continuous integration servers in use today. This paper proposes a design of a communication platform and its use to implement an event-driven continuous integration server. 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.

KEYWORDS
Integration server — Event-driven development.

1. Introduction

When developing software, continuous integration is a practice where developers integrate their changes often, usually multiple times a day [1]. To be able to use continuous integration efficiently, a 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. Within the development pipeline, automated testing is another stage and the CIS is being used to govern the whole process. 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. 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 CIS 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 the CIS 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.

This paper proposes a design of a communication platform named Meeko [2] and its use to implement a proof-of-concept CIS named Cider [3]. 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 rest of the paper is structured as follows: The following section provides more details of what the problematic scenarios not easily approachable by the current CIS are. Related works are discussed and existing systems are evaluated in more details. Then the functional requirements of the required system are defined. Next sections describe the chosen system design, architecture and implementation in more details. The last section concludes the paper.

Table 1. *Events and their meanings*

Event	Description
`patch.submitted`	a new patch was submitted into the code review system
`build.succeeded`	the automated tests succeeded
`build.failed`	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. *Desired reactions to various 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 the reviewer knows the patch 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 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

2. Problem Definition and Analysis

Before we can start evaluation of existing systems, we define a model scenario, in this case a development workflow.

2.1 Target Development Workflow

The workflow 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 summarized in the following way:

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

Table 1 lists the relevant workflow events, Table 2 then defines the actions that are to be taken upon these events.

2.2 Modelling the System

We propose an abstract model that describes the behaviour of the system. 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, particularly that there is actually no global state in the system. Everything is actually happening in the components and can be thus pushed into the components. The only assumption is that all the components get the same complete information about what is happening in the system - no events are lost and they arrive into the components in the same order.

These observations are not particularly useful for analyzing existing systems, but they give us some hints on how a new system could be implemented. 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 context relevant to that specific component. No direct synchronous communication between the components should be necessary since the events carry all the information contained in the system and the interested components can simply cherry-pick relevant events and persist them to be able to make decisions later.

Going further and pushing the event-distributing mechanism into the components is not desired. As can be seen in table 1.2, `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 Role of a Continuous Integration Server

We have described the development toolchain using the notion of state machines. The same abstraction can be applied to the CIS itself. It contains a set of predefined build steps that are tied to each other in various ways, the most common one being a successful build step triggering the following automated tests step. It would be optimal to find a server working on the same principles as our abstract model. What is more important, however, is for the system to be flexible enough to implement the chosen workflow.

The CIS can be divided into two groups - they can act as standalone systems or they can be services exporting their functionality. 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 CIS 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. 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 CIS can simply return the test results. In this case the only data that needs to be kept in the CIS are 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 CIS.

In our development workflow, 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.

2. There still must be a way how to pass configuration into the test runs. A good practice is to use environment variables for that.

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 web pages containing the build output, or the output can be streamed to the user using technologies such as WebSocket.

2.4 Functional Requirements

Based on previously mentioned assumptions, we propose the list of functional requirements for the CIS.

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 connected to 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 to the requirements of that particular project. Static environment assignment is fine, but some kind of 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.

2.5 Development Tools Integration Platform

Custom-made development process is often implemented using multiple development tools. Some communication mechanism should 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.

- 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 multiple destination components.

Each of these patterns has its own place. 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 patterns 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 issue by a development tool. 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 at will without the source component knowing anything [4].

As a side note it should be mentioned that even such a popular mechanism as Webhooks does not follow exactly this simple principle we just mentioned. The destination URL must be specified, which is making the systems coupled together. So there is a need for another component to collect Webhooks and implement proper publish-subscribe. This decision of using HTTP POST request can be justified, though, because then the emitting components are really simple. On the other hand, there is a need for a proper publishing mechanism anyway and this way it is just moved somewhere else.

As already mentioned, there are two basic communication patterns - request-reply and publish-subscribe. When we check the target workflow, it is apparent that all the communication happening there is of the publish-subscribe pattern - the CIS observing the code review tool, the code review tool then waiting for the build results. That means that publish-subscribe is the core requirement that must be implemented by any communication platform that is to be used for the implementation of this particular workflow.

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 custom event handlers can access the tools and actually perform the actions requested. This is not required if the event handler can be somehow implemented as a plugin directly in that particular development tool.

3. Existing Continuous Integration Servers

In this section we evaluate some of the existing CIS against the requirements, extending and updating [5]. The following systems were chosen for their popularity and compliance with our requirements (or lack thereof).

3.1 Travis CI

Travis CI is primarily a hosted continuous integration server specifically architected to work closely with repositories hosted on GitHub. Travis will install hooks to detect changes and trigger builds. Travis is written in Ruby and it uses RabbitMQ message broker to scale and make the components decoupled from each other. The whole lifecycle of a build including the components communication looks as follows:

1. travis-listener 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.

2. travis-hub collects and routes events to interested components. It notifies travis-tasks about important events that are supposed to trigger some kind of notification. travis-hub also enqueues build jobs and enforces Quality of Services.

3. travis-worker 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 travis-logs, build events are pushed back to travis-hub.

4. travis-logs 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.

5. travis-web is the web application where users can configure and watch their projects being built.

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.

- 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 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. Travis CI is tightly coupled with GitHub in this respect and no other input sources are available.

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. A build cannot be triggered without a branch 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 CIS that appeared recently. It is written in Go and it utilizes a new container-based virtualization platform for Linux called Docker. Drone is inspired by Travis CI. It also expects a special file .drone.yml to be present in the repository. That file defines what environment to use and what command to run. A modern web interface is available for Drone.

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 [6]. 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. Facing our list of requirements, Drone certainly incorporates some interesting practices:

- As with Travis CI, the configuration is saved in the repository.
- The build environments correspond to Docker images. Container-based virtualization is 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. There is no general mechanism how to add support for more input sources. 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.

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 a better choice than Travis. 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. There is no general mechanism that can be used to build and test arbitrary patches.

3.3 Jenkins

Jenkins is a CIS written in Java. 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. 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.

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. The following list summarizes and extends the list of positive features:

- Extremely simple to set up and to define jobs using the web interface.
- 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 and the VirtualBox 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.

- Jenkins configuration is based on build jobs, which are configured using the web interface; this 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.
- 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.

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.
- 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.

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 powerful, 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 CIS. It is a framework that can be used to implement continuous integration processes for custom workflows. 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 own processes.

A BuildBot instance comprises of a build master and some number of build slaves. The system contains the following core components [7]:

- Builders - Build jobs are called builders. They specify what build steps are to be run.
- Schedulers - This component tells builders when to run. Schedulers 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.
- 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 own builders, schedulers, change sources and status targets, binding them all together in a custom way to implement the desired workflow. Since BuildBot is written in Python, the configuration is a part of the build master program. BuildBot is very close to what we consider the optimal system:

- The system is 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 way jobs are configured is not compatible with the requirements. The build steps cannot be defined in the project repository, the configuration is done in BuildBot itself. BuildBot do not 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.

BuildBot offers a lot of flexibility. It can be plugged into any existing system 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.

3.5 Other CISs

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 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. For this reason we do not analyze these systems in more details here.

3.6 Summary

There is no contemporary CIS that would comply with all the requirements. All of the existing solutions that we discussed always partly comply and partly fail.

4. Designing the Development Platform

4.1 Development Tools Integration Platform

Considering request-reply, most of the current tools support some kind of remote API, most commonly using HTTP and following the REST API design paradigm. 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. 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. A better way how to implement request-reply communication pattern is to use a transport and a protocol that truly support asynchronous remote requests, and that is how the 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 support so-called webhooks. 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 as 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.

4.2 Managing Inter-Component Communication

Having the components access each other directly without some kind of resource discovery mechanism is not a viable solution. This would be not only a clear 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 need an extra level of indirection. For request-reply the situation is actually simpler. 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 resource discovery.

Agreeing on a common communication medium opens a couple of new questions:

1. How to plug the development tools into the medium?
2. How to route data between the components?

There is no access mechanism that is common to all the development tools. Every development tool must be wrapped in a connector component that represents the tool and translates the protocol.

Considering the second point, we have to distinguish the publish-subscribe and request-reply pattern. The former pattern is basically a broadcasting pattern. 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.

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 given 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

- The system consists of multiple development tools.
- These tools need to communicate in a clear way; a common shared communication platform provides these requirements.
- There is a need for publish-subscribe and request-reply communication patterns.
- The communication should be asynchronous.
- Since there is no communication protocol common to all the components, every component must be represented by a connector or agent.

4.3 Continuous Integration Server

Following the table 1.2, the following communication occurs:

- 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.

We can advance further to design the CIS internals. When `patch.submitted` event is received, the 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.

The environment definition is represented by a particular label. Trying to reuse the communication platform, we propose to simply use the environment identifier as the method name in an RPC call, thus enqueueing the build request in the target fitting environment. Since we required the request-reply communication to be asynchronous, it does not really matter how long the request will spend just waiting for being executed since idling is not blocking any other functionality.

This architecture has one obvious benefit - the communication platform is reused for a large part of what the CIS needs to implement - testing environments discovery, management and build requests queueing and routing. It, however, puts some additional requirements on the communication platform implementation or 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.

A more detailed sequence of steps that the CIS 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, method being named after the environment label.

3. As the output is being streamed back, save it into a database, or stream it to the user and then save it at once when the build is finished.

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.

4.4 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, or in general to route TCP connections. The listening addresses could be set as special environment variables passed into the agents on startup.

5. Implementation

In this chapter we describe how to implement the ideas presented in the previous chapter.

There shall be a broker component representing the server side of the communication platform. It is the communication hub that the agents can connect to. The broker shall be layered in the following way:

- *The broker* itself knows actually nothing about the communication happening in the lower layers, it is just a supervisor for various service endpoints, which are explaned below.

- *Service exchange* is a component that represents particular communication pattern or some other functionality that can be somehow useful for all agents. It holds the global state associated with the communication pattern it represents. For request-reply the exchange would be keeping the mapping of what agent exported what methods, which is then used as the routing table.

- *Service endpoints* export functionality as provided by certain exchange over particular protocol. The exchange just communicates with the endpoints over the same exchange-specific interface, and it knows nothing about how the data is being transported to the agents.

The simplicity of the architecture is actually its greatest strength. The exchanges represent the communication patterns we found necessary, but it can be any service that all agents could benefit from. For example, there could be a service collection logs from all the agents and saving them on a single place. The same goes for the endpoints. Nothing is said about what transport or protocol they represent. The actual connection between an endpoint and an agent can be inter-process, inter-host, but also in-process communication.

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. This means that 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 using the request-reply communication pattern.

6. Conclusion

In this paper, we proposed an architecture of the event-based continuous integration server.

6.1 General System Architecture

The chosen layered 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 (exchanges) together with various transport protocols (endpoints) 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.

Not only horizontal but also vertical splitting of functionality turned out to be a good choice. Abstracting communication patterns into so-called services makes it very easy to add additional functionality in the future without touching existing potentially well-tested and proven code.

6.2 Programming Language

The system architecture approach was easily implementable using Go as the programming language since Go has a very good support for programming against interfaces. All that was necessary was to define the exchange and endpoint interfaces for various patterns.

6.3 Robustness

Splitting the agents into processes turned out to be a robust solution, especially when the system is extended to run agents somehow sandboxed. In this way, the agents cannot harm each other. 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 potentially offer the communication platform as a hosted service in the future.

6.4 Performance

Using ZeroMQ messaging library for both request-reply and publish-subscribe prototype endpoints turned out to be sufficient in terms of performance. Even without any attempts to explicitly tune performance, the system is capable of transporting more than 10 thousand messages (= events) per second when using the ZeroMQ inter-process transport (Unix domain sockets). Occasional build requests as occuring in a small company working on a couple of projects cannot challenge this throughput either when the platform is used to implement a continuous integration server.

6.5 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.

One limitation was, however, discovered while trying to deploy the communication platform to production, which was in this case a cloud application hosting provider. The agent supervisor installs the agents by downloading their sources and creating an executable using special scripts present in the agent repository. This does not work well on these cloud platforms as a service since the files saved to disk there are usually not durable and they can be deleted at any time. The supervisor, however, counts on the fact that the repositories are not disappearing randomly after being downloaded.

In a way this limitation is not surprising, because our system is already a platform as a service for the agent components, so deploying the whole system as a single application would basically mean that we are implementing a platform as a service within another platform as a service.

Another deployment option could be to really use an existing cloud platform to deploy each and every agent as well as the broker as an application for the chosen cloud platform. This would, however, bring some more security considerations since all the connections to the broker would have to be somehow authenticated since they would be effectively the same as any other connection across the Internet. This deployment architecture has not been attempted.

The conclusion is that using proper infrastructure (as a service) to deploy the system onto a real full-fledged host is the preferred way of doing things.

Acknowledgments

The authors would like to thank the SVV project no. SVV-2015-260100 and GACR project no. P103/13/08195S, which supported this paper.

References

[1] FOWLER, Martin. Continuous Integration. May, 2006. [Online]. Available: `http://martinfowler.com/articles/continuousIntegration.html`. Last accessed Fevrary 10 2014.

[2] KUPKA, Ondrej. Meeko homepage. 2013. [Online]. Available: `http://meeko.io`

[3] KUPKA, Ondrej. Cider homepage. 2013. [Online]. Available: `https://github.com/cider`

[4] EUGSTER, Patrick Th, FELBER, Pascal A., GUERRAOUI, Rachid, et al. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 2003, vol. 35, no 2, p. 114-131.

[5] KUPKA, Ondrej. Choosing a Build Server for a Small Business. January, 2013. Available: `http://tchap.wikidot.com/blog:1,visitedon2014-02-10`

[6] XAVIER, Miguel G., NEVES, Marcelo Veiga, ROSSI, Fabio D., et al. Performance evaluation of container-based virtualization for high performance computing environments. In : *Parallel, Distributed and Network-Based Processing (PDP)*, 2013 21st Euromicro International Conference on. IEEE, 2013. p. 233-240.

[7] Buildbot Introduction. [Online]. Available: *http://docs.buildbot.net/0.8.8/manual/introduction.html*, Last accessed Fevrary 10 2014.