POLITECNICO DI TORINO

Department of Control and Computer Engineering
Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Envoy: a simplified approach for the integration of hardware devices into software applications

Supervisor:

Prof. Massimo VIOLANTE

Co-supervisor:

Teodoro PICCINNI

Candidate:

Cristian MENALDO

Academic Year 2020/2021

Abstract

The context of this thesis is the development of the Envoy middleware software, a solution minded to simplify the integration of hardware peripherals for the automation of cash handling.

This thesis has been developed within the innovation department of ARCA Technologies S.r.l., the producer of these devices. The company, based in Bollengo (TO), is the Italian branch of a wider company based in Mebane, North Carolina, USA.

This project has successfully integrated, into the Envoy platform, the most successful cash automation device sold by ARCA, the CM18.

Envoy is a middleware software, written in C++, that exposes an API (Application Programming Interface) that simplifies the integration of this kind of hardware devices into software applications. The software houses that choose Envoy to integrate cash handling devices, won't care about the kind of protocol (hexadecimal, string, ...) a machine uses, because Envoy parses the protocols and manages the low level communication. On this basis, Envoy receives JSON (JavaScript Object Notation) requests over a RESTful protocol, translates them into the device protocol and returns the replies to the application in the same simple and clean JSON format.

In order to be able to integrate this machine, it has been necessary to study how it works, with all its components, the path that is followed by the banknotes inside of it, but most importantly its protocol, so that it has been possible to build the parser of the commands.

Before starting to develop the code, an extensive study has been carried out, about all the tools needed in a software development flow. This flow has been based on the Agile and DevOps methodologies and has gone through three phases: development, testing and release. The development phase has taken advantage of tools like Jira, used for planning the work; Jenkins, which is a tool for CI/CD (Continuous Integration and Continuous Delivery) of software products; and Git, a VCS (Version Control System) that facilitates the collaboration between programmers and allows to keep track of the software product history. The tool that has been used in the code testing phase is a tool to implement unit testing, the Boost.Test library, which is part of the Boost C++ framework.

After this phase of study, the code for the integration of the machine into Envoy has been developed, using the tools mentioned previously and implementing the commands to open and close the communication with the machine, the ones to retrieve the status of the cassettes and the status of the various components, the commands to set and read the date and the time, the command to deposit the banknotes and lastly, the one to only count them.

Finally, as conclusion of the project, a collaboration with the customer support team has been done, in order to understand how the issues with the customers are addressed, once the software has been delivered.

Table of Contents

List of Figures

Acronyms

RPC
: Remote Procedure Call

API
: Application Programming Interface

JSON
: JavaScript Object Notation

HTTP
: HyperText Transfer Protocol

CGI
: Common Gateway Interface

RLM
: Reprise Lincense Manager

IDL
: Interface Definition Language

QA
: Quality Assurance

CI/CD
: Continuous Integration and Continuous Delivery

VCS
: Version Control System

GUI

Graphical User Interface

OOP

Object-Oriented Programming

IDE

Integrated Development Environment

DLL

Dynamic-Link Library

XML

Extensible Markup Language

REST

REpresentational State Transfer

HTML

HyperText Markup Language

LDN

Logical Device Name

RTC

Real Time Controller

Chapter 1

Introduction

1.1 The company: ARCA story

"ARCA is a company leader in the cash automation industry with worldwide offices: US, where there is the global headquarters in North Carolina (**Fig. 1.1**), UK, Italy, in Bollengo (TO), where I currently work, and France.
It all started in 1998, when Mort O'Sullivan, the founder and current CEO, while he was studying at University in Edinburgh, Scotland, he collaborated with Compuflex, a company that develops software to control cash automation hardware and where he had already worked before. The project was about the creation of a new cash dispensing system for the oldest bank in Ireland, belonging to the Bank of Ireland.

Figure 1.1: ARCA Headquarters in Mebane, North Carolina [**1**]

After that experience, he returned to New York City, where he founded Arca.Tech Systems in his apartment, thinking that there was a great potential in cash automation in the US. His first customers were businesses that needed equipment for use in self-service checkouts, kiosks and ATMs. Those products were cash dispensing devices produced by third-party OEM manufacturers. In 2000, the company released its first branded product, the ARCA 2000, that was sold at a much lower price with respect to the competition, since their customers were looking for a cheaper cash dispenser than the ones that were available on the market.

In 2001, ARCA moved to the RTP (Research Triangle Park) area of North Carolina, where it started its growth; the partnership with the Italian manufacturer CTS Group began and thanks to this ARCA started to launch more innovative products, such as the evolution of the cash dispenser, the cash recycler: this device allows also to accept and store notes as a deposit. One of the first cash recyclers introduced in the US was the CM24, in 2004.

Seven years after the foundation in the United States, ARCA expanded into Scotland, England and Ireland. Moreover, in 2007, it won a spot on the Inc 5000 list, a prestigious award that recognizes the fastest growing companies in the US.

In 2008, it launched the device that would become its flagship product with more than 26,000 units sold all over the world, the **CM18** cash recycler. The CM18 had so much success that ARCA replicated its platform over many other models. After this period of great success, the company purchased the CTS Group in 2014.

Nowadays, among the ARCA's customers there are financial institutions, retail businesses, OEM self-service checkout and kiosk manufacturers. They offer multiple systems that increase efficiency and revenue and reduce costs. ARCA's successful performance has been recognized and honored by many of the industry's most prestigious award organizations." [**1**]

1.2 Cash automation devices

The cash automation devices are machines that mainly execute some simple but important tasks: accepting, validating, depositing, counting and dispensing cash, thus automating the cash cycle. They can be found in banks, credit unions and back-office retail cash rooms.

The banknotes are placed into a specific slot, called feeder, then they pass through a bill validator that recognizes their denomination (i.e. their value and currency type) and their validity: if a banknote is valid, then the machine stores it in the proper cassette, based on its denomination; otherwise it is rejected and put in the reject slot. When, afterwards, a withdrawal is needed, the machine takes the banknotes from the cassettes and puts them into the output slot. Here it can be perfectly seen why these machines are also called cash recyclers: the money that is deposited it's the same that is dispensed.

"Cash recycling has a huge impact on every process and every person in financial institutions, improving the operational efficiency and the productivity of the staff, and reducing the labor costs and the other costs associated with the manual cash handling. Moreover, it increases the amount of cash that can be stored and enhances the quality of cash controls." [**2**]

As said before, the cash recycler which ARCA had a great success with is the CM18, it can be seen here on the side and it has the following specifications:

- 3.5" operator touch screen interface
- Modular note storage allows for multiple configurations based on capacity needs
- Simultaneous processing of up to four currencies
- ARCA Care remote service plan
- Lighting system on input/output bins for customer usability
- Lockable upper track for customer applications

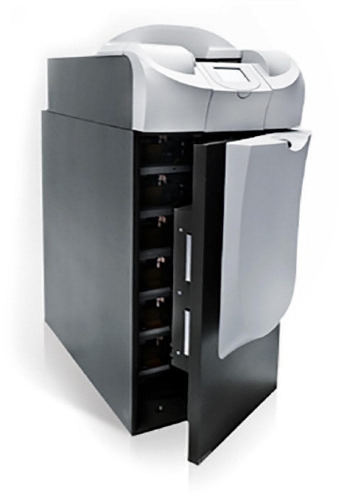


Figure 1.2: CM18 [**2**]

- Capacity of $85,000 to $200,000+ depending on configuration
- Deposit/dispense in batches of up to 200 notes without limits per transaction
- Full image sensors for visible, UV and IR light plus magnetic and ultrasound sensors
- RS232, USB 2.0 and TCP/IP connectivity
- **CM18 Tall (CM18T):**
  - 6/8/10/12 cassettes
  - Dimensions (h × w × d): 37.01 x 17.32 x 36.10 in / 940 x 440 x 917 mm
  - Weight: 945.78 lb / 429 kg
- **CM18 Short:**
  - 6/8 cassettes
  - Dimensions (h × w × d): 26.42 x 17.32 x 36.10 in / 671 x 440 x 917 mm
  - Weight: 714.30 lb / 324 kg

1.3 The project: Envoy overview

Envoy, also called EnvoyRPC, is an application written in C++, that allows customer applications to communicate with cash handling devices via Remote Procedure Calls (RPC). The project goal is to produce a lightweight, language-agnostic API (Application Programming Interface) for customer applications, as long as they can send JSON commands and receive JSON responses through a dedicated Apache HTTP server supplied by EnvoyRPC, so that they may more easily communicate with the automated cash handling machines sold by ARCA.

The general installer of Envoy contains Apache HTTP server, CGI applications, other software modules, licensing system, configuration files and driver for devices. The Apache HTTP server acts as intermediary between the customer application (the client) and EnvoyRPC itself, exchanging JSON commands (from the application to the machine) and JSON responses (from the machine to the application). Further details will be explained in **Sec. 3.1.1**.
CGI applications are mainly involved in the validation and translation of the JSON commands and responses.
The other software modules manage physical communication (USB, LAN, RS232, etc.) with devices and the creation of the commands and responses according to the specific protocol of a machine.
This general structure is represented in **Fig. 1.3**.

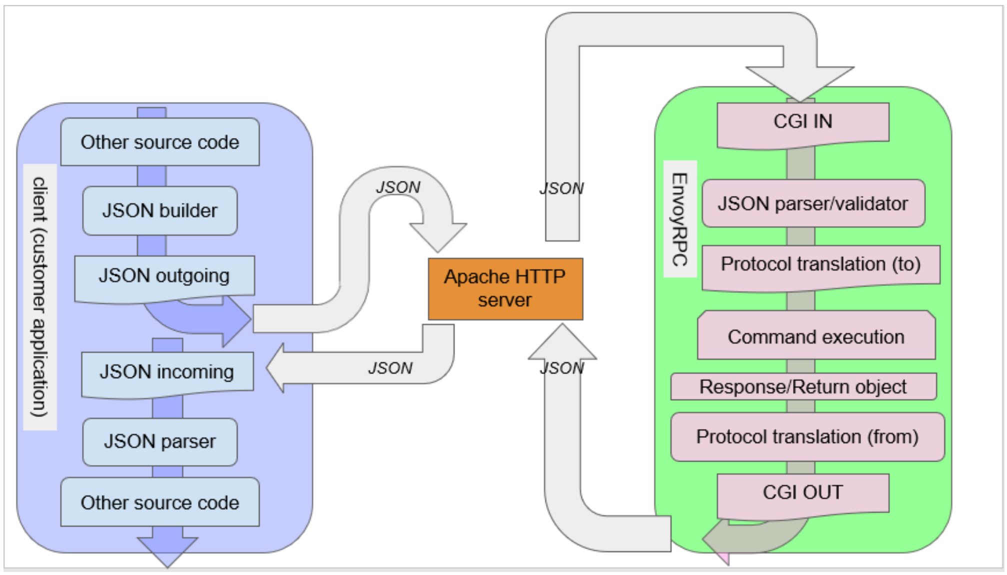


Figure 1.3: Envoy structure

In particular, the module responsible for the translation of the JSON objects translates the command to be sent to the machine from the JSON format to the specific protocol of that machine and vice versa, i.e. it creates the JSON response starting from the response received from the machine.
Moreover, among all the modules, there is also the module for the logging, used to print to a log file all the calls to the different functions, including all commands coming into the system and all the responses going out, in order to understand, in case of failure, where something went wrong; the module dedicated to the errors/exceptions handling and the one that manages the licenses that belong to the Reprise Lincense Manager (RLM).
Another one is the unit and functional test module, that run the tests to verify if all the pieces of code behave as expected, either taken individually (unit testing) or interacting together (functional testing).

Third party libraries are also needed: such as "libusb", used to manage the USB communication with the devices; "rapidjson", for the creation of the JSON objects and their handling; and "INI", for manipulating and parsing .ini configuration files.

Everything is compiled with the usage of the CMake build system either for Linux or for Windows and installers for Linux Debian and Windows are generated.

The API is a JSON API that uses Apache with a collection of EnvoyRPC templates and executables. The application conducts all configuration, licensing and device communication through specific JSON messages. JSON responses are then returned. As long as the application can be made to execute GET and POST, and send and receive JSON data, the particular language used to write the application does not matter. The customer's application is free to use utilities such as cURL, Java packages such as java.net.http, Perl, Python or any custom-designed HTTP communication modules.

The JSON templates are the basic format for all commands and responses. They define the Interface Definition Language (IDL) of the API in terms of what the customer application sends and receives. The application can either use copies of the command templates and store them for use as needed or it can fetch them from EnvoyRPC on the fly and fill out the variables. Like all JSON strings, they are human-readable.

The Apache service used for EnvoyRPC runs just as any other Apache server. In the case of EnvoyRPC, access is granted through port 8081 (localhost:8081). It can be stopped, started and restarted. On installation, the service is started immediately and is already configured. For example, it will start upon host reboot.

On uninstallation, it is entirely removed from the host. EnvoyRPC does not run as a service itself, when a command is completed by sending a response, the executable stops completely and exits.

Moreover, EnvoyRPC provides a browser interface that can be used to run all commands and display their responses. It can be used to configure the system or the devices, test commands, read the documentation and learn about the system. The browser interface acts as another application with JavaScript converting to/from JSON.

Chapter 2

Development flow

In the development of hardware and software products each company implements its own workflow. Along the evolution of the IT systems, due to the demanding needs for reliability and compliant release times, various schools of thought emerged, helping developers in the organization of their work.
In the context of the Envoy project development, I have applied two of the most popular methodologies widespread nowadays: **Agile** and **DevOps**.
In particular, the first one helped me to define features of the product that I was developing and when to develop them; while the second one has driven the management of the software development, defining steps to ensure the finding of errors and to reduce the release time of the application.
I will go into the details of each single methodology in the coming paragraphs.

2.1 Agile software development

The agile process is an iterative approach to software development, to deliver the product incrementally, instead of all at once. Every software is a composition of several functionalities, each of them having to pass through 6 steps: **requirements**, **design**, **develop**, **test**, **deploy** and **review**. Then if the review is satisfactory the functionality is added to a release, otherwise it's necessary to go back to the requirements and redo the other steps.

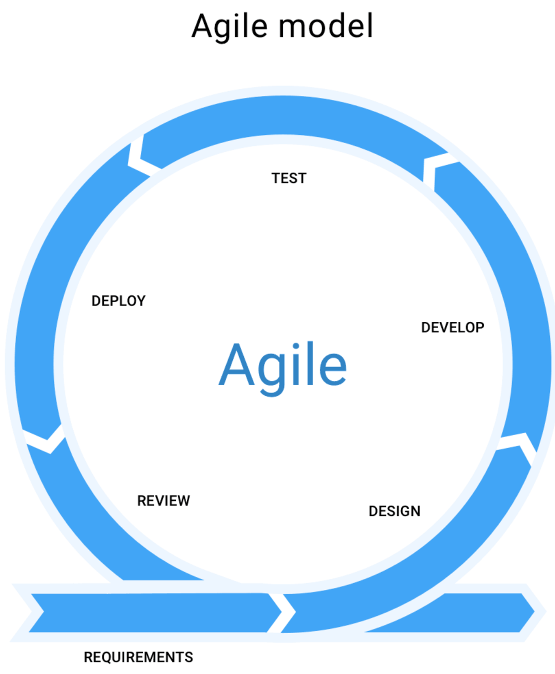


Figure 2.1: Agile process phases [4]

Before seeing in detail what is done in each step of the cycle, let's spend few words explaining what is the **Scrum Team** and which are the roles within it.
The Scrum Team is defined as a hierarchy of employees working on a software development project, interacting to achieve agile development.
There are essentially three roles: the **Product Owner**, who is the responsible for the entire project, someone that deals with the customers and understands what they want to create; the **Scrum Master**, who is the middleman that connects and manages the flow of information between the Product Owner and the **Development Team**, that is composed of the employees that write code, developing the software.

Now the explanation of the various steps:

- **Step 1: Requirements**
  In this initial phase the Product Owner writes the first documentation, listing all the initial requirements, such as the end result of the project, the features that will be implemented and the ones that will not be initially supported, because in this phase it is recommended to consider only the strictly necessary features, while the other ones will be added once the product is deployed and the core features work well.

- **Step 2: Design**
  After having written the requirements, the Product Owner gathers the Development Team, which meets and decides which are the best tools to deal with the project, in order to reach the best possible result; these tools are the programming language, the frameworks and the libraries the project will use.

- **Step 3: Develop**
  This is the longest and most important phase, since it consists in writing code and so realizing the actual software product.

- **Step 4: Test**
  In this step the code written in the previous phase is tested to verify that there are no bugs and that it is compatible with every piece of code which has been written before by the developers. Moreover, the QA (Quality Assurance) team makes sure that the code behaviour meets the requirements.

- **Step 5: Deploy**
  This is the phase in which the product, i.e. the application, is deployed on the servers and so made available to the customer either for a demo or for a real use. In the future cycles new features will be added to the already installed software and if any bugs are present they will be fixed, thus providing updates.

- **Step 6: Review**
  In this last step the Product Owner meets the Development Team again to review the progress made towards the completion of the project and the fulfillment of the requirements. In addition, the developers can suggest how to solve the problems that have arisen during the previous phases and the Product Owner takes their ideas into consideration.

In order to explain better the Agile methodology let's suppose we have to build an application: the software development team takes time to break the application down into **User Stories**, that are functional requirements of a desired feature which must be implemented, described in a narrative form, indicating who wants what and why, with some acceptance criteria that must be satisfied.
Then those user stories are broken down into tasks, all of which is located in the backlog of whatever project management software that has been chosen, for example **Jira** (see **Sec. 2.1.1**).
After that, a sprint is created, that usually lasts two weeks, in which those tasks composing the user stories are taken, based on what is needed to get done in this sprint, according to the prioritization set by the project manager or the customer. Each day during the sprint a task is picked and once it has been fully completed, that means coded, tested, successfully built and submitted for peer review, it is the turn of the next task.
All this stuff is repeated every day until the sprint is over and then all the code is examined by the QA team. Afterwards, the development team, the product owner and maybe the customer meet to determine what is needed to get done in the next sprint.
The next sprint is basically the same, but it may happen that is necessary to add more tasks to the backlog, because maybe the information coming from the QA testers tells that some other tasks must be accomplished either by the end of this sprint or the next one.
And this cycle is repeated, with minor variations, for each sprint till the completion of the application.

2.1.1 Jira

Among all the available software products for project management, for the Envoy project I have used Jira.
Jira is a software suite developed by Atlassian company in 2002, used to plan and manage software development projects based on Agile methodologies.
It is mostly used for the Scrum Agile methodology, the methodology, as explained before, employed by teams that are working on new features with a tight schedule for finishing their work as they try to align with a projected launch target or the coordination with delivery from other teams.

Like in all the Agile project management softwares, also in Jira there is a **Backlog**, that is the list of all the tasks and features in which the whole software product has been broken down near the beginning of the project, after having set the user stories.

In Jira the word **Issue** can represent whatever bit of work, for example a task, a bug or a story, within the workflow towards the completion of the project.
Moreover, there is a specific hierarchy in the pieces composing the project: the Project is split into Epics/Components, then each Epic is divided in more Stories, each of which divided in Tasks, that, in turn, can be subdivided in Sub-tasks.
An example of this hierarchy is shown in **Fig. 2.2**.

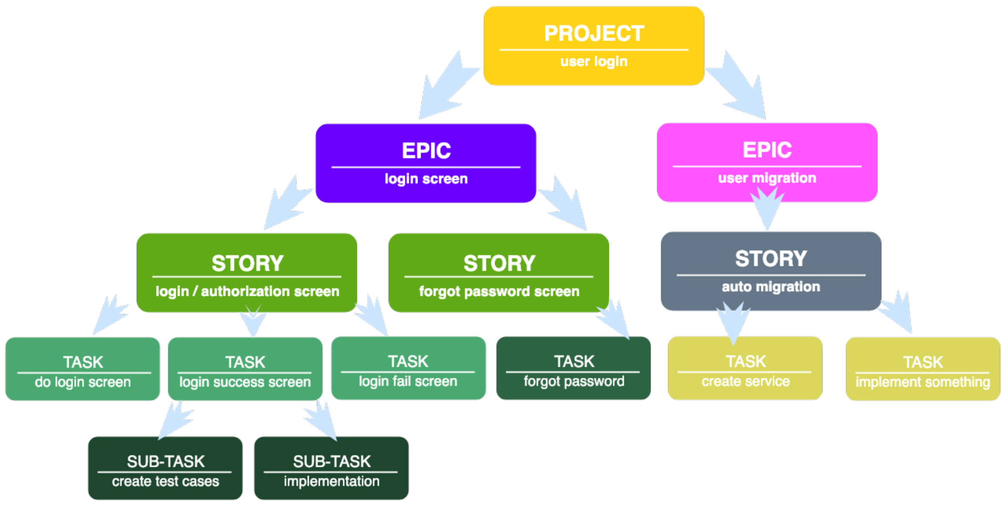


Figure 2.2: Jira hierarchy example

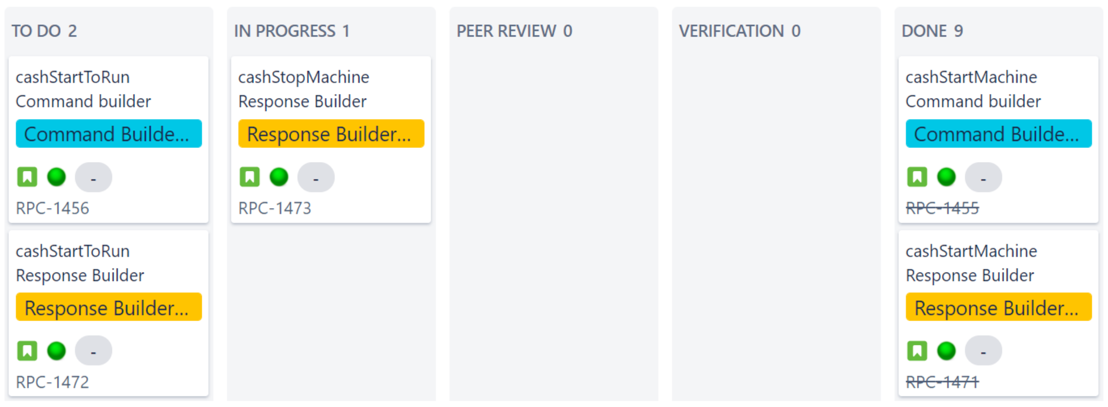
The first issue that must be created in a project is an **Epic**, which represents a large amount of work that needs to be broken down into a number of smaller stories and tasks and it may take several sprints to complete it.

In order to complete an epic, it's necessary to create several stories. **Stories** are usually used by Product Managers to describe planned work for a specific feature of a product.
Some points are assigned to each story depending on its complexity: in Agile, **Story Points** are used to estimate work, thus planning the right amount of work to be done in a sprint. Story points help to overcome the uncertainty of estimating, while still creating a useful quantifier for the items in the backlog, they are just a relative measure of complexity.
To each issue can be assigned a story point of 1, 2, 3, 5, 8, or 13, these are numbers in the Fibonacci sequence, they are typically used because, when tasks are small, the ability to imagine exactly what it will be done is fairly accurate: when the estimates are 1 or 2 these are actually usually somewhere around proportionately reliable, the 2 really is twice as hard as the 1. But as tasks get bigger the ability to estimate exactly gets worse and worse, the error in estimation becomes greater and greater and at that scale the difference between a 12 or 13 is mostly irrelevant. Fibonacci numbers map nicely to this error in estimation and that makes it a good sequence to use for story pointing.
Each story is assigned to a team member, who becomes the responsible for that story.

Tasks are typically used by any team member to describe other planned, non-story work. A single task should take at least a few hours to complete, this helps to minimize the amount of energy spent tracking things that are done very quickly and it ensures that the critical and high priority items are not lost in a sea of items that do not require as much visibility.
However, a single task should not take more than three days to complete, with this as a guideline it is ensured that the team can reprioritize at least twice per week, if needed. This also helps to make sure that every team member is always less than 72 hours from getting the next thing done.

Moreover, **Bugs** are another type of issue that can be created, they are used to explicitly call out unplanned work and this is helpful when the team is trying to improve the overall quality of the software product.

As said before, in the Scrum style of Agile, teams break their work up into iterative batches called **Sprints**, in this way they are able to set clear and iterative milestones for their work.
A sprint has a start date and an end date and they are often two weeks long; while the sprint is active no new work should be added to the active sprint, instead, it is put in one of the upcoming sprints.
Every task or sub-task present in the sprint has to pass through 3 steps during the workflow: *to do*, *in progress* and *done*. At the beginning of the sprint all the tasks are in the *to do* state, then during the sprint they are picked up one by one, put in the *in progress* sate and once completed in *done*. When all the tasks have reached the *done* state, the sprint can be considered completed and it can be closed.
These steps are represented through a, so called, Kanban board (**Fig. 2.3**).



TO DO 2

cashStartToRun Command builder
Command Builde...
-
RPC-1456

cashStartToRun Response Builder
Response Builder...
-
RPC-1472

IN PROGRESS 1

cashStopMachine Response Builder
Response Builder...
-
RPC-1473

PEER REVIEW 0

VERIFICATION 0

DONE 9

cashStartMachine Command builder
Command Builde...
-
~~RPC-1455~~

cashStartMachine Response Builder
Response Builder...
-
~~RPC-1471~~

Figure 2.3: Kanban board

2.2 DevOps

DevOps is a software development approach which involves continuous development, testing, integration, deployment and monitoring of the software throughout its development lifecycle.
Whereas the Agile process is between the customer and the development team, DevOps is between the development team (Dev) and the IT operations team (Ops).

DevOps allows software development teams to implement the so called **Continuous Integration** and **Continuous Delivery** (**CI/CD**) through the automation of all the relative phases of the entire process, since automation is the concept that underlies this approach, thus helping to decrease the time to market of their products.

"In the CI/CD acronym CI always refers to the Continuous Integration, an automated process, applied by the developers, in which the new modifications made to the code of the application are regularly compiled, tested and merged to a shared repository. In this way it is possible to solve the problem of conflicts among the several branches of an application, in the development phase.

Instead, CD can stands either for Continuous Delivery or Continuous Deployment, related concepts frequently used interchangeably. Both are involved in the automation of the next phases of the workflow, but sometimes they are used separately in order to illustrate the level of automation that has been applied.

Continuous Delivery means the process through which the changes made to the application by a developer are automatically tested, searching for bugs and then uploaded to a repository, from which the operations teams deliver them to a production environment. This is a solution to the problem of poor communication between the development and operations teams. To that end, Continuous Delivery has the purpose to guarantee minimal effort to deliver new code.

The other possible acronym for CD, Continuous Deployment, refers to the automatic deployment of the modifications made by the developer from the repository to the production, where they become available for the customers. In this way it is possible to avoid manual procedures from operations teams, that otherwise would slow down the deployment of the applications. This phase exploits the advantages of the Continuous Delivery, automating the next phase of the workflow." [**5**]

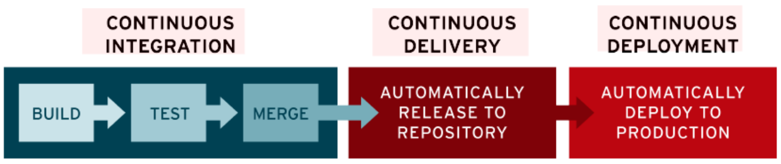


Figure 2.4: CI/CD flow [5]

In the DevOps approach we can distinguish the following phases:

- **Development**: in this phase, composed of planning, coding and building, the software is developed continuously, but the entire development process is divided into smaller cycles, in order to speed up the software development itself and the delivery process.
- **Testing**: then, in the testing phase, the QA team tests the new piece of code, that implements a new functionality, using tools to identify and fix bugs and verifying that all the specific requirements are satisfied.
- **Integration**: the next stage is the integration, where the new functionality is integrated with the other previously tested functionalities, and integration testing is performed to check if the new feature interacts properly with the other ones, not introducing any bug.
- **Deployment**: once the code has been tested and it was found that there are no bugs, it can be deployed and with DevOps approach the deployment process takes place continuously.
- **Monitoring**: the monitoring phase consists in the job of the operations team that deals with a potential unwanted behaviour of the application or eventual bugs found in production.

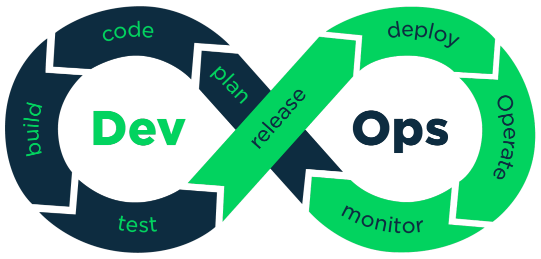


Figure 2.5: DevOps phases [6]

Among the tools used by DevOps for automation there is **Jenkins**, which is a tool that provides continuous integration and testing and helps to integrate new features into a software product by quickly finding eventual issues as soon as a build is deployed.

2.2.1 Jenkins

Jenkins is an open-source CI/CD tool and automation server written in Java, developed by Kohsuke Kawaguchi in 2011, that allows continuous development, test and deployment of newly created codes.

Before Jenkins, developers had to wait until the entire software code was built and tested before checking for errors, in this way fixing bugs was very difficult, since they had to scan the whole code. There was no iterative improvement of the code and the software delivery process was slow. Before, all the codes were pulled from the repository only after a defined deadline and build together, whereas with Jenkins the code is pulled whenever a change in the code is committed and so the source code is built continuously.

Jenkins pipeline

Jenkins workflow is based on **pipelines**, that are series of stages through which every new piece of code of an application has to pass, from the initial development environment to the final production environment.

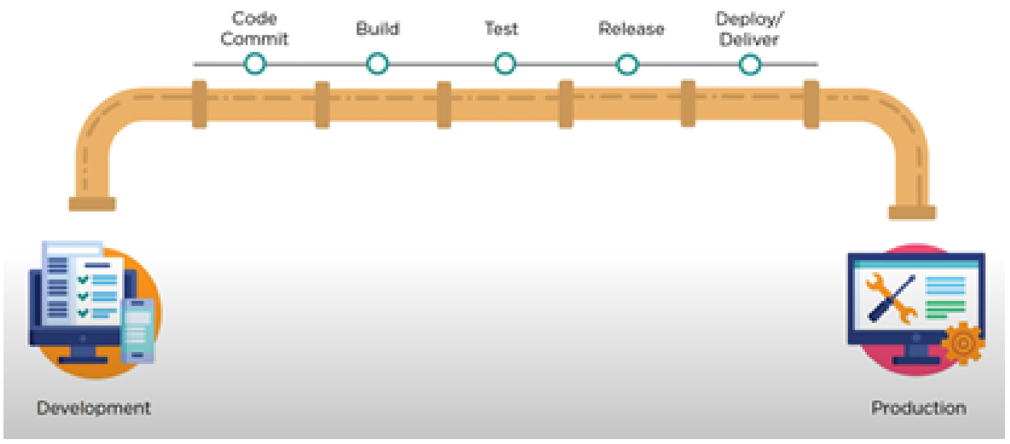


Figure 2.6: Jenkins pipeline

The first stage consists in committing the new piece of code, written in the development environment, to a repository, such as a Git (see **Sec. 2.3.1**) server, so in this instance Jenkins is used to commit the code automatically.

Jenkins will then create a build of the code and part of that build process is actually going through and running through tests. Developers are already comfortable with running and writing unit tests to validate their code but there may be additional tests that Jenkins is running. So for instance, as a team they may have a standard set of tests for how they actually write out their code, so that each team member can understand the code that has been written and those tests can also be included in the testing process within the Jenkins environment.

Assuming everything passed the tests, the development team can then get everything placed in a release ready environment within Jenkins.

And finally the code is ready to be deployed or delivered to a production environment. Jenkins, with its server environment, is the tool that helps the developer to be able to deploy its code to the production environment and the result is the quick passage from a developer to a production code.

Jenkins provides 2 ways of developing a pipeline code: **scripted** and **declarative**.
The scripted pipeline is based on Groovy script and one or more *node* blocks does all the work throughout the entire pipeline.
Instead, the declarative one provides a simple syntax to define a pipeline without the need of a Groovy script and here a *pipeline* block defines the work to be done throughout the pipeline. The script is divided in *stage* blocks, within which it's defined the *agent*, which is the host that runs that block, and the *steps*, that are the actions to be executed.
It's good practice to put the pipeline script on a Git repository.

```
pipeline {
    agent none
    stages {
        stage('Example Build') {
            agent { docker 'maven:3.8.1-adoptopenjdk-11' }
            steps {
                echo 'Hello, Maven'
                sh 'mvn --version'
            }
        }
        stage('Example Test') {
            agent { docker 'openjdk:8-jre' }
            steps {
                echo 'Hello, JDK'
                sh 'java -version'
            }
        }
    }
}
```

Listing 2.1: Declarative pipeline example [**7**]

Jenkins architecture

Jenkins has a master-slave architecture: a server, the master, pulls the code from the repository every time a commit is detected and then it distributes its workload to all the slaves, which carry out builds and tests and produce test reports. The slaves can be run on different operating systems.

Moreover, Jenkins is composed of different kind of servers: a **continuous integration server**, that checks the repository at regular intervals and pulls any newly available code; a **build server**, such as Maven, which builds the code generating an executable file and in case the build fails it sends a feedback to the developers; a **test server**, that executes specific test scripts, which are written, for example, in Selenium, and also in this case, if something goes wrong, a feedback is sent to the developers; and finally, if there are no errors, the tested application is then deployed to the **production server**.
Maven and Selenium are just two of the hundreds of plugins available for Jenkins.

2.3 Version Control Systems

Before version control systems, programmers would share code through emails or USB pen drives and keeping backups was essential to the development process, in order to not lose all the work done. A VCS (Version Control System) helps programmers to keep track of revisions to a code base and there are three types of VCS:

1. the first one is local, each time you make changes to the files they are kept locally as patches and you can revert to previous change by adding together all the patches, but this does not help when you are trying to collaborate with someone else
2. so, the second generation of VCSs are centralized, it means that all the revisions are kept on a server and multiple clients can access these files, but if something were to happen to the central server, then all your revision history would be lost
3. now the third generation is the distributed VCS, where there is a central server with all the history, but differently from the previous generation, every local collaborator has its own history kept as well

Among all the distributed VCSs, the most famous is **Git**.

2.3.1 Git

Git belongs to the third generation of VCSs, in fact it is an open-source distributed version control system, originally developed by Linus Torvalds (the developer of Linux) in 2005, to handle efficiently either small or large projects.
Every project has its own Git repository, which is essentially a public shared folder that holds the project itself, with all the folders, source codes, etc. and each repository is located on a server and can be cloned to many different clients, so that it is possible to have as many collaborators as needed working on a project. Each cloned copy also has all the necessary information to revert any past changes with the full history at the programmer's disposal.

So, Git allows files in the local repository to be in three different states: **unmodified**, meaning that the file is the same locally as it is on the server; **modified**, that is your local copy is ahead of the server; or **staged**, which means that your changed copy is ready to be committed to the server, so that the server can have the same copy as the one in local.

Since more programmers can work on the same files, it can happen that some conflicts arise and Git, with its structure, simplifies the way in which these conflicts are solved, sometimes automatically merging the different versions of a file, otherwise it highlights the differences between the two versions of the same file.

GitFlow

GitFlow, created by Vincent Driessen, is the Git workflow "branching model", it's a set of guidelines the software development teams can follow to manage their projects. Once a programmer clones the central remote repository, he works locally maybe creating new branches. When he has finished, he pushes his branches to the central repository, that is he publishes his modifications and makes them available on the remote repository.
There are different kinds of branch:

- **master**: it's the main branch that stores the official release history
- **develop**: it is used as an integration branch for features, there is one for each repository and it diverges from *master* branch
- **feature**: it diverges from *develop* branch, it is used to develop new features and the "sprint" work, that is the workload usually assigned in two weeks, is committed/pushed here; at the end it merges back into *develop*
- **release**: it diverges from *develop* and it is created to fix bugs once the *develop* has acquired enough features for a release. It must be tested by a QA team to

make sure that all the changes are 100% stable; any bug picked up by the QA team will need to be addressed and this can be done on this branch. Once completed, it will be merged back into *develop* but also into *master*, adding a version number/tag, which is a way of tracking releases

- **hotfix**: it diverges from *master*, hot fixes are defined as minor fixes to a project, they could be spelling errors, or maybe a label that needs to be changed or something so small that it does not require a team of people to test the changes. At the end, it is merged back into *master*, creating a new version tag, and into *develop*

Fig. 2.7 is useful for a better understanding of the GitFlow branching model, it shows how the different types of branch are used.

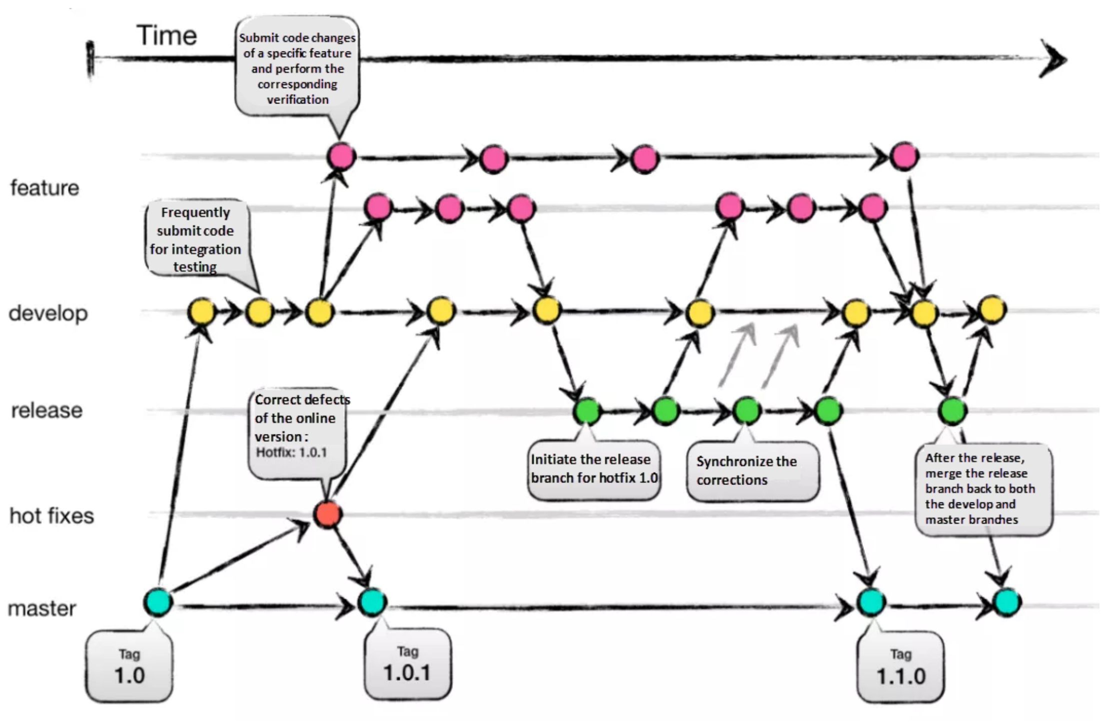

Time
Submit code changes of a specific feature and perform the corresponding verification
feature
Frequently submit code for integration testing
develop
release
Correct defects of the online version: Hotfix: 1.0.1
Initiate the release branch for hotfix 1.0
Synchronize the corrections
After the release, merge the release branch back to both the develop and master branches
hot fixes
master
Tag 1.0
Tag 1.0.1
Tag 1.1.0

Figure 2.7: GitFlow branching model [8]

Git on command line

There are a lot of different ways to use Git, for example the original command line tools or several GUIs (Graphical User Interfaces) with different potentialities, but the most important aspect is that from the command line you can run all Git commands, whereas with the GUIs you are able to execute only a subset of them. Moreover, using the command line leads to a better and deeper comprehension of the way Git works, in fact who runs the command line version he most likely comprehends how to run the GUI version, but the opposite is not necessarily true.

For all these reasons I have used Git from the command line and among all the available commands, the ones that I have needed are: **clone**, **checkout**, **status**, **add**, **commit**, **push**, **pull**, **branch**.
Before seeing them in detail and how I have used them, it is necessary to show Git structure, that is the areas where it virtually places the files.

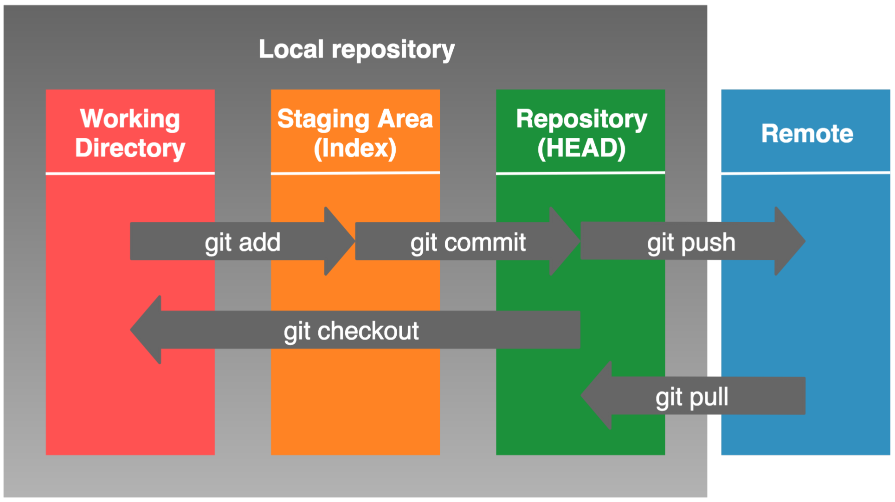


Figure 2.8: Git structure [**9**]

The **Working Directory**, also called **Working Tree** is the area where the files are actually present, i.e. where you are currently working; if there are some changes in the files, but you don't save them to Git, you will lose all the modifications. This is due to the fact that Git only recognizes the files or changes in the working directory, but it does not save them automatically. If you run the *status* command you will be able to see all the files present in the working directory and that Git labels as "untracked" files.

The **Staging Area** or **Index** is the area where Git tracks and saves the changes that occur in the files and these files are ready for the next commit, i.e. when the modifications are made available on the local copy of the repository.

The **Repository** or **HEAD** is the local copy of the remote repository and it's the area that saves everything; the files are moved here after the *commit* command, and after committing the staging area will be empty. Then the *push* command is needed in order to publish all the changes and to update the remote repository.

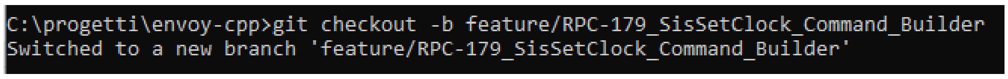
And finally, **Remote** is the remote repository itself, the shared folder on a server, accessible by all the programmers that collaborate to that project.

Git commands

Clone: I have used this command to make a local copy on my PC of the remote Git repository containing the Envoy project.

Now I am going to present the other Git commands in the exact order that I have run them, in order to show the code development flow that I have followed for the implementation of each machine command; practical examples, referred to the development of the *SetClock* command of the CM18 machine, will be provided.

Checkout: without options this command is used to move from one branch to another one, but adding the "-b" option it creates a new branch with the name specified after "-b".

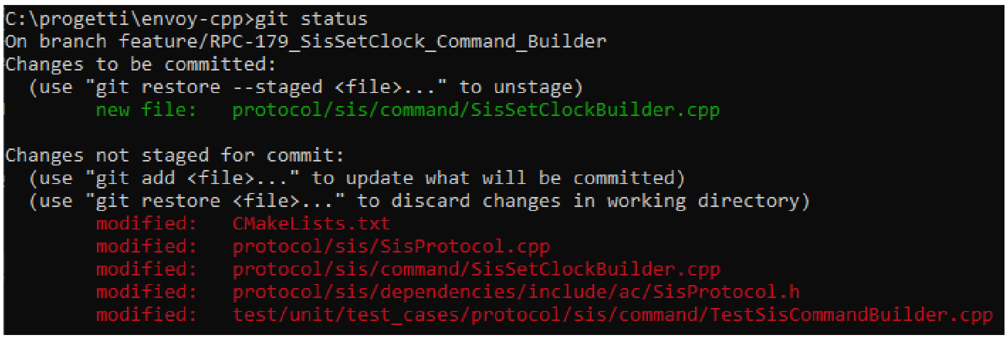


```
C:\progetti\envoy-cpp>git checkout -b feature/RPC-179_SisSetClock_Command_Builder
Switched to a new branch 'feature/RPC-179_SisSetClock_Command_Builder'
```

Figure 2.9: New branch creation

The "RPC code" that can be noted in the branch name is a reference code automatically assigned by **Jira**.

Status: it shows the status of the files within the current branch, it tells if there are modified, new or deleted files, and if they are already in the staging area ready to be committed, or still in the working directory.



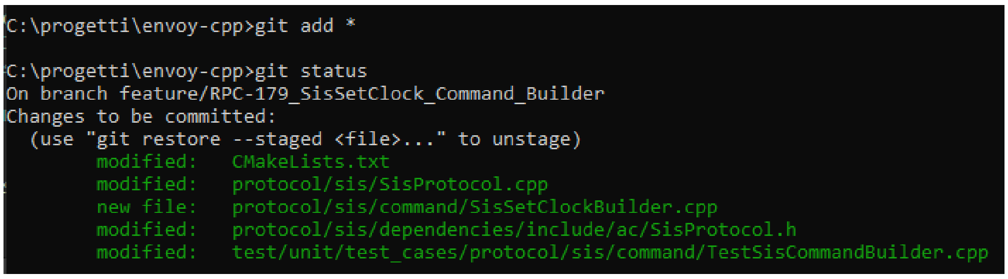
```
C:\progetti\envoy-cpp>git status
On branch feature/RPC-179_SisSetClock_Command_Builder
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   protocol/sis/command/SisSetClockBuilder.cpp

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   CMakeLists.txt
        modified:   protocol/sis/SisProtocol.cpp
        modified:   protocol/sis/command/SisSetClockBuilder.cpp
        modified:   protocol/sis/dependencies/include/ac/SisProtocol.h
        modified:   test/unit/test_cases/protocol/sis/command/TestSisCommandBuilder.cpp
```

Figure 2.10: Git status after coding

Fig. 2.10 shows the status after I have written the code to implement the *SetClock* command.

Add: it moves the specified files with changes from the working directory to the staging area, ready for the next commit; with the “-A” option or the "*" it adds all the modified or new files.

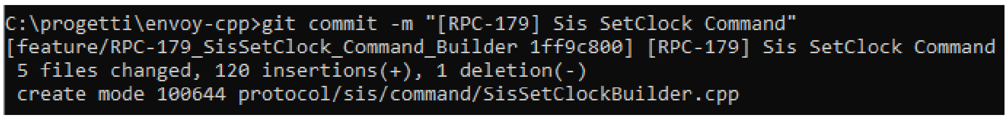


```
C:\progetti\envoy-cpp>git add *

C:\progetti\envoy-cpp>git status
On branch feature/RPC-179_SisSetClock_Command_Builder
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   CMakeLists.txt
        modified:   protocol/sis/SisProtocol.cpp
        new file:   protocol/sis/command/SisSetClockBuilder.cpp
        modified:   protocol/sis/dependencies/include/ac/SisProtocol.h
        modified:   test/unit/test_cases/protocol/sis/command/TestSisCommandBuilder.cpp
```

Figure 2.11: Git status after add command

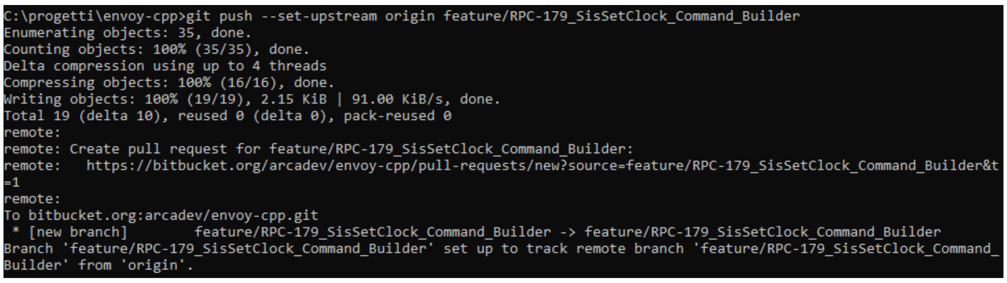
Commit: this command adds the modified or new files, that are in the staging area, to the local copy of the remote repository, with the “-m” option it is possible to add a comment; moreover, every commit has its own hash code to identify it.



```
C:\progetti\envoy-cpp>git commit -m "[RPC-179] Sis SetClock Command"
[feature/RPC-179_SisSetClock_Command_Builder 1ff9c800] [RPC-179] Sis SetClock Command
 5 files changed, 120 insertions(+), 1 deletion(-)
 create mode 100644 protocol/sis/command/SisSetClockBuilder.cpp
```

Figure 2.12: Git commit

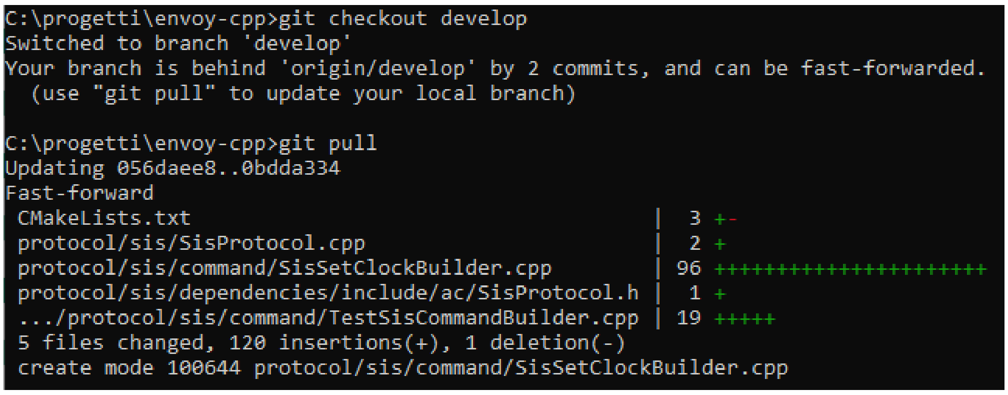
Push: used to upload all the committed files to the remote repository, in order to make all the modifications available for the other programmers collaborating to the project.



```
C:\progetti\envoy-cpp>git push --set-upstream origin feature/RPC-179_SisSetClock_Command_Builder
Enumerating objects: 35, done.
Counting objects: 100% (35/35), done.
Delta compression using up to 4 threads
Compressing objects: 100% (16/16), done.
Writing objects: 100% (19/19), 2.15 KiB | 91.00 KiB/s, done.
Total 19 (delta 10), reused 0 (delta 0), pack-reused 0
remote:
remote: Create pull request for feature/RPC-179_SisSetClock_Command_Builder:
remote:   https://bitbucket.org/arcadev/envoy-cpp/pull-requests/new?source=feature/RPC-179_SisSetClock_Command_Builder&t
=1
remote:
To bitbucket.org:arcadev/envoy-cpp.git
 * [new branch]        feature/RPC-179_SisSetClock_Command_Builder -> feature/RPC-179_SisSetClock_Command_Builder
Branch 'feature/RPC-179_SisSetClock_Command_Builder' set up to track remote branch 'feature/RPC-179_SisSetClock_Command_
Builder' from 'origin'.
```

Figure 2.13: Git push

Pull: this command is used to update the local copy of the repository to the last version of the remote repository and if there are conflicts Git highlights them, so that they can be fixed merging the two versions of the same file, or it can happen that Git fixes them automatically.



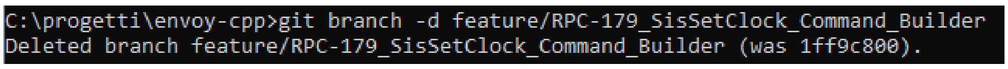
```
C:\progetti\envoy-cpp>git checkout develop
Switched to branch 'develop'
Your branch is behind 'origin/develop' by 2 commits, and can be fast-forwarded.
  (use "git pull" to update your local branch)

C:\progetti\envoy-cpp>git pull
Updating 056daee8..0bdda334
Fast-forward
 CMakeLists.txt                                          |  3 +-
 protocol/sis/SisProtocol.cpp                            |  2 +
 protocol/sis/command/SisSetClockBuilder.cpp             | 96 ++++++++++++++++++++++
 protocol/sis/dependencies/include/ac/SisProtocol.h      |  1 +
 .../protocol/sis/command/TestSisCommandBuilder.cpp      | 19 +++++
 5 files changed, 120 insertions(+), 1 deletion(-)
 create mode 100644 protocol/sis/command/SisSetClockBuilder.cpp
```

Figure 2.14: Git pull on branch develop

After having merged the *feature/RPC-179_SisSetClock_Command_Builder* branch, i.e. the one that I created for the implementation of the *SetClock* command, to the *develop* branch, I have moved to this branch (with the *checkout* command) and then I have run the *pull* command to update my local branch to the remote branch. (**Fig. 2.14**)

Branch: if this command is run without options it shows the list of the local branches only, but adding the “-a” option it shows also the remote branches; whereas adding a name it creates a new local branch; in addition, the “-d” option, followed by the branch name, is used to delete that branch locally.



```
C:\progetti\envoy-cpp>git branch -d feature/RPC-179_SisSetClock_Command_Builder
Deleted branch feature/RPC-179_SisSetClock_Command_Builder (was 1ff9c800).
```

Figure 2.15: Branch deletion

Once verified that the new command worked fine and with its addition everything compiled and built without errors, I have deleted the branch created for its implementation, since having merged it into the main branch (*develop*) there was no more need to keep it. (**Fig. 2.15**)

2.4 Code development

Envoy is an application developed in C++, one of the first OOP (Object-Oriented Programming) languages, whose purpose was to add the concepts of object orientation to the C programming language.
In fact, like all the OOP languages, C++ provides the concepts of class, object, method, attribute, inheritance, polymorphism etc., but we will see them in detail in **Sec. 2.4.1**.

Whereas, in order to compile the source code of Envoy and to link all the needed libraries I have used **CMake**, which is a cross-platform open-source software for the development automation (**Sec. 2.4.2**).

2.4.1 Object-oriented programming concepts

Before object-oriented programming there was procedural programming, that divided a program into a set of functions, so data were stored in a bunch of variables and functions operated on the data. This style of programming is very simple and straightforward, but it is suitable only for simple programs, because as the programs grow, it will end up with a group of functions that are all over the place and the programmer might find himself copying and pasting lines of code over and over, or he makes a change to one function and then several other functions break: this is what is called "spaghetti code", there is so much interdependency between all these functions and it becomes problematic.

OOP came to solve this problem, in this programming paradigm a group of related variables and functions that operate on them is combined into a unit called **object**, the variables are referred to as **attributes** and the functions as **methods**; an object is the instantiation of a **class**, and more objects belonging to the same class can be instantiated.
While in procedural code functions have so many parameters, in OOP methods end up having fewer and fewer parameters, the fewer the number of parameters the easier it is to use and maintain that method.
This way of combining related variables and functions into objects is called **encapsulation** and it is one of the OOP basic concepts.

Another concept of the OOP is **abstraction**: some of the attributes and methods can be hidden from the outside and this gives a couple of benefits.
The first one is that the interface of those objects is made simpler, in fact using and understanding an object with a few attributes and methods is easier than an object with several of them.

The second benefit is that it helps to reduce the impact of change, it means that if the programmer changes these inner or private methods, none of these changes will leak to the outside, because there isn't any code that touches these methods outside of their containing object. The programmer can delete a method or change its parameters but none of these actions will impact the rest of the application code.

The third core concept in OOP is **inheritance**. Inheritance is a mechanism that allows to eliminate redundant code, for example if there are objects having some attributes and methods in common, instead of redefining them for every object, they can be defined once in a generic object, the so called "parent" object, and then let the other objects, the "child" objects, inherit them, specifying it in their definition.

Finally, the concept of **polymorphism**. "Poly" means "many" and "morph" means "form", so polymorphism means "many forms" and in OOP is a technique that allows to get rid of long "if/else" or "switch/case" statements, determining what kind of function to run while the program is running.
For example, if more objects inherit a method from their parent class, but the implementation of this method is different for each object, i.e. this method is overridden, thanks to the power of polymorphism, the right implementation among all the possible ones is figured out during runtime.

So, summing up, the benefits of object-oriented programming are the following ones:

- Using encapsulation it is possible to group related variables and functions together and in this way the code complexity is reduced, moreover an object can be reused in different parts of a program or in different programs.
- With abstraction the details and the complexity can be hidden and only the essentials are shown. This technique reduces complexity and also isolates the impact of changes in the code.
- Inheritance, instead, allows to eliminate redundant code.
- With polymorphism it is possible to refactor ugly switch/case statements.

2.4.2 CMake

"CMake was developed by Bill Hoffman between 2000 and 2001, with the aim of creating a powerful cross-platform build environment. He took inspiration from **pcmaker**, an open-source 3D graphics and visualization system, adding many more features and implementing some of the functionalities of the Unix **configure** tool. Then, with the years, it has been improved thanks to other developers integrating it into their own build systems." [**12**]

In particular, this software is a build file generator, meaning it generates files, so it doesn't really build, which is something not necessarily easy to understand about it. The build files are written in a compiler-independent configuration language, platform-independent, using what is called CMake language, which is not the nicest thing but it is quite flexible.
CMake allows to do multiple things: first it allows to build the software by generating all the files; then, it allows to test this software and also to package it.

CMake itself is a cross-platform software, so it can run on Windows, MacOS, Linux and a lot of different platforms, because it is a very flexible software. It is open-source, anyone can always contribute, they can build it from source themselves if they want to use it on some unusual platforms.
It puts a lot of emphasis on backwards compatibility, so if someone has some old scripts, they will still work today probably, provided that they have no bugs, and this makes things really easy to upgrade to newer versions of CMake and to reuse old code.

Since CMake is a build file generator, it builds files for lots of different IDEs (Integrated Development Environments), such as **CodeLite**, which is the one that I have used to write the source code, and build tools, like **Make**, that I have used to build the code. It supports many platform targets, like Windows, MacOS, iOS, Linux, Android and many others.

"The configuration files are called "CMakeLists.txt" and are placed in each source directory and sub-directory that compose a project. Each *CMakeLists.txt* file contains a set of commands, whose form is *COMMAND(arg1 arg2 ...)*, where *COMMAND* is the command name and then, between brackets, there is a list of arguments separated by a blank space.
Starting from these files, CMake generates standard build files, e.g. makefiles on Unix and workspaces or projects on Windows, which can be used in the compiler environment that has been chosen.
So, CMake can generate native build environments that compile source code, create

libraries, generate wrappers and build executables or also static and dynamic libraries.
With CMake it is possible to manage complex directory hierarchies and applications that depends on multiple libraries, in fact CMake supports projects composed of several libraries, where each of them may contain multiple directories and the application may depend on the libraries plus additional code." [**12**]
This is precisely the case of **Envoy**.

In order to generate the CodeLite workspace, starting from the *CMakeLists.txt* file, I have run the following command from the command line:
cmake -G "CodeLite - MinGW Makefiles" -S . -B .\codelite

Whereas here below, the section of the *CMakeLists.txt* file that I have written to build the DLL (Dynamic-Link Library) corresponding to the SIS protocol (see **Sec. 3.2.2**), which is the protocol that I have implemented and integrated in the Envoy project.

```
project (SisProtocol)

# Setting source files
set(SOURCE_FILES
     ${CMAKE_CURRENT_LIST_DIR}/protocol/sis/
     SisProtocol.cpp
     ${CMAKE_CURRENT_LIST_DIR}/protocol/sis/command/SisExtendedStatusBuilder.cpp
     ${CMAKE_CURRENT_LIST_DIR}/protocol/sis/response/SisExtendedStatusResponseBuilder.cpp
     ${CMAKE_CURRENT_LIST_DIR}/protocol/sis/command/
     SisOpenBuilder.cpp
     ${CMAKE_CURRENT_LIST_DIR}/protocol/sis/response/SisOpenResponseBuilder.cpp
     ${CMAKE_CURRENT_LIST_DIR}/protocol/sis/command/
     SisCloseBuilder.cpp
     ${CMAKE_CURRENT_LIST_DIR}/protocol/sis/response/SisCloseResponseBuilder.cpp
     ${CMAKE_CURRENT_LIST_DIR}/protocol/sis/command/
     SisDepositBuilder.cpp
     ${CMAKE_CURRENT_LIST_DIR}/protocol/sis/response/SisDepositResponseBuilder.cpp
     ${CMAKE_CURRENT_LIST_DIR}/protocol/sis/command/SisAcceptDepositBuilder.cpp
     ${CMAKE_CURRENT_LIST_DIR}/protocol/sis/response/SisAcceptDepositResponseBuilder.cpp
     ${CMAKE_CURRENT_LIST_DIR}/protocol/sis/command/SisReadDateTimeBuilder.cpp
21      ${CMAKE_CURRENT_LIST_DIR}/protocol/sis/response/
        SisReadDateTimeResponseBuilder.cpp
22      ${CMAKE_CURRENT_LIST_DIR}/protocol/sis/command/
        SisGetUnitCoverButtonStateBuilder.cpp
23      ${CMAKE_CURRENT_LIST_DIR}/protocol/sis/response/
        SisGetUnitCoverButtonStateResponseBuilder.cpp
24      ${CMAKE_CURRENT_LIST_DIR}/protocol/sis/command/
25      SisCountBuilder.cpp
26      ${CMAKE_CURRENT_LIST_DIR}/protocol/sis/response/
        SisCountResponseBuilder.cpp
27      ${CMAKE_CURRENT_LIST_DIR}/protocol/sis/command/
28      SisSetClockBuilder.cpp
29      ${CMAKE_CURRENT_LIST_DIR}/protocol/sis/response/
        SisSetClockResponseBuilder.cpp)
30
31  # Creating dll
32  add_library(SisProtocol SHARED ${SOURCE_FILES})
33
34  # Linking libraries
35  target_link_libraries(SisProtocol Utility)
36
37  # Including directories
38  target_include_directories(SisProtocol PUBLIC
39      ${CMAKE_CURRENT_LIST_DIR}/protocol/sis/dependencies/include/ac
40      ${CMAKE_CURRENT_LIST_DIR}/protocol/sis/command/dependencies/
        include/ac
41      ${CMAKE_CURRENT_LIST_DIR}/protocol/sis/response/dependencies/
        include/ac)
```

Listing 2.2: SisProtocol section in CMakeLists.txt

In **List. 2.2** it is possible to see how the CMake commands are used:
in particular, at line 4, the *set* command is used to set the source files;
at line 32, the *add_library* command, with the *SHARED* option, indicates that this will be a DLL, i.e. a dynamic library; with the *STATIC* option, instead, this would have been a static library, or using the *add_executable* command, this would have been an executable;
then, at line 35, the *target_link_libraries* command sets the libraries that this project will link;
and lastly, at line 38, the *target_include_directories* command includes the directories where the header files are located.

2.5 Code testing

Code testing is one of the most important phases in the development of a software product. It is the last phase in the software development lifecycle, before deployment, so this step consists in verifying that the code has no bugs, that every piece of code works fine with all the other pieces of code, not generating any unwanted behaviour, and that all the requirements set in the initial design phase are met.

In other words, it is necessary to make sure that we developed the product right and also that we developed the right product, these concepts refer respectively to the more technical words **verification** and **validation** of the famous V-model (**Fig. 2.16**) that represents the software development phases. In fact, verification checks that the code works perfectly without any bugs, whereas validation makes sure that the software product matches all the client requirements. So, not only the software must work but it should also work efficiently.

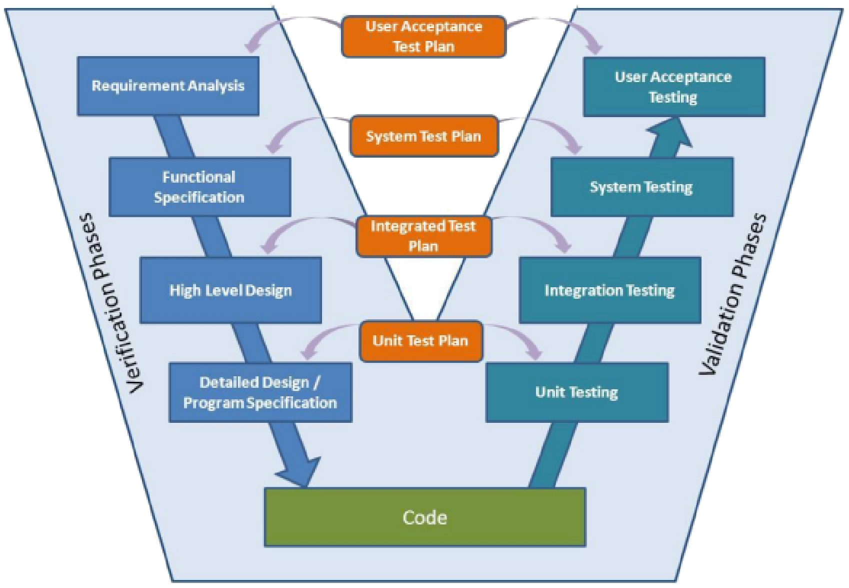


Figure 2.16: V-model in software development

There are two different types of testing, they are **functional testing** and **non-functional testing**: the first one consists in testing the code, i.e. the classes and the methods, so the actual working of the software; while, the second one tests the performance and the scalability of the software, for example a software for editing purpose should work smoothly and should not give any lags, or an application deployed on cloud should scale properly.

We can distinguish four different levels of testing:

1. **Unit testing**: in this first level, the smallest part of the software is tested, which is a class with all its methods. The classes are called units, from here the name "unit testing". Every single method is tested, giving in input all the possible meaningful values and checking if the real output matches the expected one.

2. **Integration testing**: once each component has been tested, they are all combined together to test this as a software, all the different classes are combined in order to verify that their interaction works well and does not produce any error.

3. **System testing**: this is the overall testing of the system, to test how it works with other software and with the actual data.

4. **Acceptance testing**: this is the last level, where it is verified that the software product meets the customer requirements.

Every level is responsibility of a different role, in particular the developer focuses on the unit testing and since I have dealt with it in my thesis project, I will go into more detail in the next section.

2.5.1 Unit testing

"The aim of unit testing is to check individual units of a source code separately, where for unit is intended the smallest part of a code that can be tested alone, like a function or a class method.
Unit testing helps to modularize the code, because breaking the code into several specialized pieces makes it easier to be tested; moreover, having a suite of unit tests that can be run iteratively ensures that everything keeps working every time a new functionality is added or something has been changed in the code." [**13**]

A unit test is a method that checks a specific functionality, it has clear pass/fail criteria and has this form:

```
Test (TestGroupName, TestName) {
    1 - setup block
    2 - running the under-test functionality
    3 - checking the results (assertions block)
}
```

Where the setup block is used to initialize all the variables needed to execute the test, then there is the section in which the test is effectively run, and lastly the last section contains the assertions, that are functions that check if the real output matches the expected one.

There are some guidelines that can be followed in order to write good unit tests:

- assuring the independence of each unit test, so that there are no tests blocking the execution of other tests;
- testing all the public methods, as well as class constructors and operators;
- making sure that the results are not affected by the order in which the tests are run.
- checking invalid input data, covering all the possible branches of the code and checking also the edge cases;

"In the scope of unit testing we can find the concepts of **suite**, **fixture**, and **mock object**: the first one refers to a group of tests that are logically connected or have common functionality, for example when the same function is tested in different cases; the fixture is a class that is used with groups of tests sharing the same data, to set up before and to clean up after the environment for each test in the group, thus avoiding code duplication; and the mock objects are objects that simulate the behaviour of those real objects that are difficult to be tested because of their complex dependencies.

Some frameworks were developed with the purpose of making it easier to carry out unit testing. Unit testing frameworks, in fact, are used for the automation of the operations involved in the creation of the tests, such as preparing the environment, writing the test code, printing the output messages etc...
Moreover, in addition to the usual **assert** macro, they also provide the **expect/check** macros that don't interrupt the test program after a single check failure: checkers compare the actual and expected results, including tolerances for floating point comparisons, and provide pre-implemented exception handlers.
Frameworks also allow to customize tests output messages: they can be simple pass/fail results (usually used for regression testing), user-defined messages, or more descriptive and verbose outputs.
With some of these frameworks it is also possible to create a report exporting the results in XML (Extensible Markup Language) format and this report can be passed to a continuous integration system like **Jenkins**." [**13**]

One of the most popular unit testing library for C++ is provided by the Boost framework and I am referring to **Boost.Test**, which is the one that I have used to implement my tests.

2.5.2 Boost.Test

Boost.Test is a unit testing library and it's part of the Boost C++ framework, it has a lot of interesting features, for example it provides several assertion macros and generates the output in XML format.

Checkers

"Checkers are macros with the format *BOOST_[level]_[checkname]*, where *level* is the severity level (optional) and *checkname* is the type of check, and they can take one or several arguments:

- **BOOST_WARN**: it produces a warning message if the check failed, but the error counter is not increased and the test case continues.
- **BOOST_CHECK**: it reports an error and increases the error counter when the check failed, but the test case continues.
- **BOOST_REQUIRE**: it is used for reporting fatal errors, when the execution of the test case should be aborted, for example to check whether an object that will be used later was created successfully." [**13**]

Suites

As previously stated, suites are groups of tests and in the case of Boost tests can be organized in suites using the pair of macros
BOOST_AUTO_TEST_SUITE(suite_name) and
BOOST_AUTO_TEST_SUITE_END().
Here below an example:

```
BOOST_AUTO_TEST_SUITE(ThreeThree_suite)
    BOOST_AUTO_TEST_CASE(testPlus) {
        BOOST_CHECK_EQUAL(3+3, 6);
    }
    BOOST_AUTO_TEST_CASE(testMult) {
        BOOST_CHECK_EQUAL(3*3, 9);
    }
BOOST_AUTO_TEST_SUITE_END()
```

Fixtures

In Boost fixtures are created using the macro *BOOST_FIXTURE_TEST_CASE()*:

```
struct SampleF {
    SampleF() : i(3) { }
    ~SampleF() { }
    int i;
};

BOOST_FIXTURE_TEST_CASE(SampleF_test, SampleF) {
    // accessing i from SampleF directly
    BOOST_CHECK_EQUAL(i, 3);
    BOOST_CHECK_EQUAL(i, 4);
    BOOST_CHECK_EQUAL(i, 5);
}
```

In **List. 2.3** one of the tests that I have written to test my code, in particular the one that checks the correct format of the JSON corresponding to the machine response to the *Open* command.

```
BOOST_AUTO_TEST_CASE(SisOpenRspBuilderTestCase_1) {

    unsigned char openResp[] = "Open,num,side,repcode";
    string method = "open";
    string LDN = "CM18_Test";
    string id = "1";

    string rspTemplate(R"({"jsonrpc":"2.0","method":"open",
    "description":"Opens a transaction session with a CM18/OM61
   machine", "result":{"LDN":null,"side":null,"replyCode":null,
    "description":null},"id":null})");

    unsigned char *rspPtr = openResp;
    SisProtocol sisProtocol;
    SisProtocol *sisProtocolPtr;
    sisProtocolPtr = &sisProtocol;
    string response = sisProtocolPtr->getResponse(LDN, method,
   rspTemplate, rspPtr, sizeof(openResp), id);
    Document doc;
    doc.Parse(response.c_str());
    StringBuffer buffer;
    Writer<StringBuffer> writer(buffer);
    doc.Accept(writer);
    string actual = buffer.GetString();
    string expected = R"({"jsonrpc":"2.0","method":"open",
    "description":"Opens a transaction session with a CM18/OM61
   machine", "result":{"LDN":"CM18_Test","side":"side","replyCode
   ":"1",
    "description":"OK"},"id":"1"})";

    BOOST_CHECK_EQUAL(expected, actual);
}
```

Listing 2.3: Open response test

Chapter 3

Environment analysis

3.1 Envoy framework

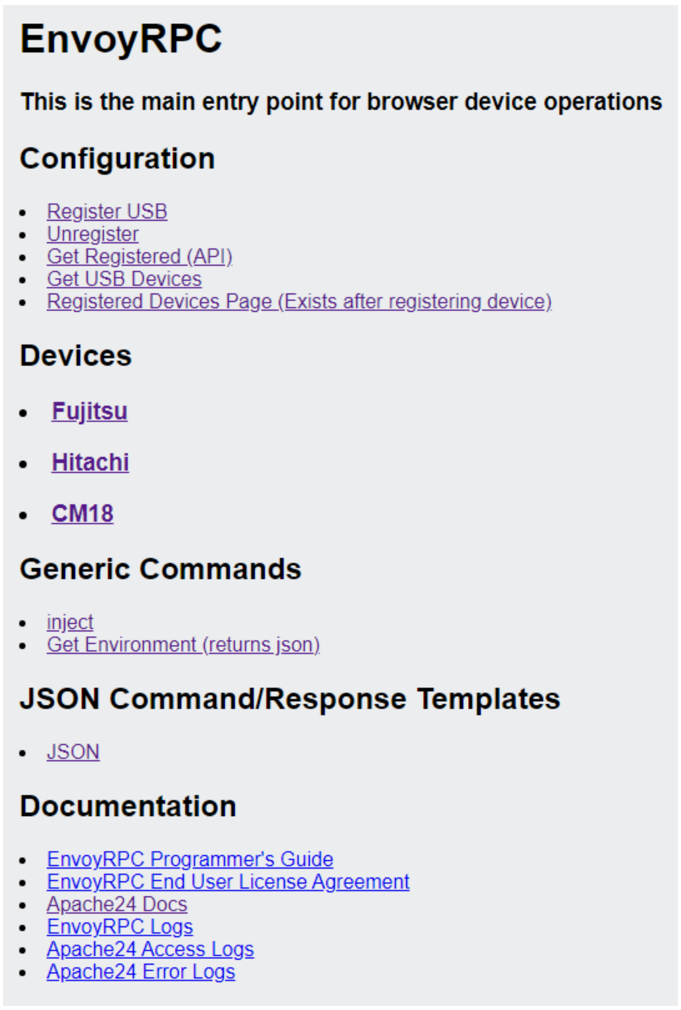
3.1.1 The Apache HTTP server and the web interface

As previously said in **Sec. 1.3**, Envoy supplies an Apache HTTP server accessible through port 8081 on localhost (localhost:8081), in fact to an application, Envoy is essentially an HTTP service with a JSON API, which is accessible to any application that can perform HTTP GET, POST, PUT or DELETE. The integrator customers run their cash automation devices from a single computer host and the Envoy API is intended to facilitate the issuing of commands and return of responses from those devices.

It can be considered a RESTful API. REST is the acronym of REpresentational State Transfer and it is a set of architectural constraints for a client-server architecture where a resource representational state is transferred. An API is considered RESTful if it is compliant with the following criteria:

- a client-server architecture composed of client, server and resources, with the requests managed by HTTP;
- a stateless client-server connection, so that the client information are not stored with the GET requests;
- data must be saved in cache, in order to optimize the client-server interactions;
- a uniform interface for the components, to transfer all the information in a standard form (the JSON format for example).

The EnvoyRPC web interface (**Fig. 3.1**), available by typing "localhost:8081" into the browser address field, is an application with a JavaScript library creating the JSON messages that get sent to the CGI back-end.
Configuration commands, device commands and links to documentation are included on the home page. In certain cases, forms are filled out to assign values to variables such as bill length and thickness, denominations etc. These are the same variables that correspond to the JSON API, and the same variables that should be filled out for a POST with data from the application.
All the commands available in the web interface are implemented as API endpoints in the Apache CGI directory and therefore can be run via methods that can send POST with JSON data.


EnvoyRPC

This is the main entry point for browser device operations

Configuration

- Register USB
- Unregister
- Get Registered (API)
- Get USB Devices
- Registered Devices Page (Exists after registering device)

Devices

- Fujitsu
- Hitachi
- CM18

Generic Commands

- inject
- Get Environment (returns json)

JSON Command/Response Templates

- JSON

Documentation

- EnvoyRPC Programmer's Guide
- EnvoyRPC End User License Agreement
- Apache24 Docs
- EnvoyRPC Logs
- Apache24 Access Logs
- Apache24 Error Logs

Figure 3.1: Envoy browser interface

3.1.2 CGI applications

The CGI (Common Gateway Interface) calls are the application's entry point to Envoy device command processing. POSTed JSON data result in the CGI executable, passing command request parameters to a JSON parser (such as a class method in a library), then to the protocol synthesizer module and finally to the low-level communication modules communicating with "libusb" library.
The response from the device is then translated by Envoy, converted into a JSON representation and sent back out through the CGI script as a JSON response.
Each CGI executable is the entry point for a separate device command. Thus, some modularity is achieved by separation of device-specific functionality.
These executables have been generated running a script, that, starting from the JSON templates located in *localhost:8081/json*, has created the corresponding .cgi files.
Then, every command is represented by an HTML file, inside which a javascript is run, creating the link to the CGI file.
In **Fig. 3.2** the representation of the flow.

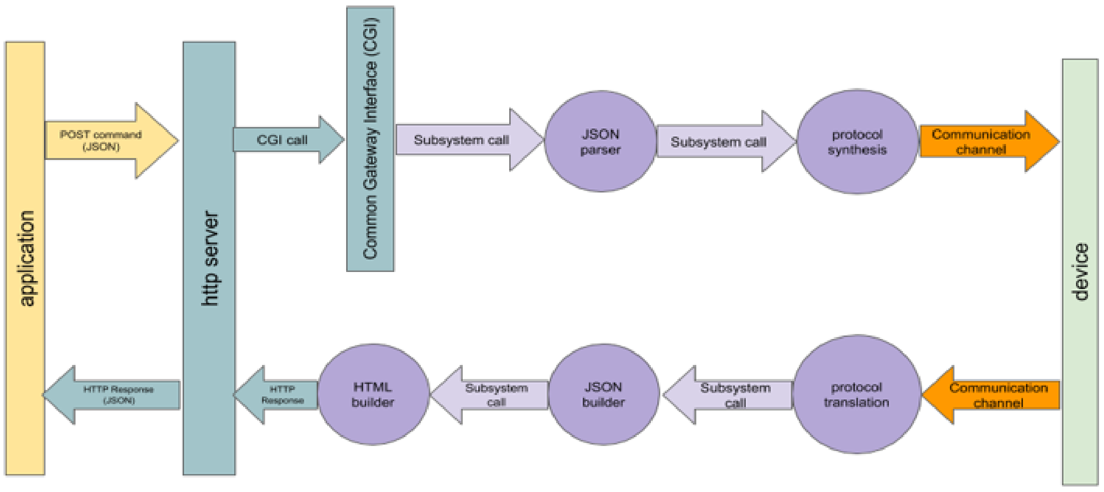


Figure 3.2: Envoy flow

3.1.3 JSON templates

JSON format

JSON stands for JavaScript Object Notation and it is a lightweight data-interchange format and also an open standard file format; it was originally specified by Douglas Crockford in 2000 and it is based on a subset of the JavaScript Programming Language Standard.
It is very easy both for humans to read and write, in fact it uses human-readable text to store and transmit data objects, and for machines to parse and generate. JSON is a text format that does not depend on the programming language used, but it has conventions for the C family languages, such as C, C++, C#, Python, Perl, Java, JavaScript, etc.

"Thanks to these properties JSON is an ideal data-interchange language, with a wide range of functionality, for example the communication of web applications with servers. Many modern programming languages include libraries to generate and parse JSON-format data, like "rapidjson" in C++. The extension for the JSON filenames is .json." [**14**]

"Since JSON is a data format interchangeable with programming languages, it is essentially based on two structures: a collection of name/value pairs, that can be realized through an object, record, struct, dictionary, hash table, keyed list, or associative array; and an ordered list of values, which can be realized as an array, vector, list, or sequence.
In this way, since these are universal data structures, practically all modern programming languages support them in one form or another.

The following ones are the elements that can be present in a JSON:

- *object*, it is an unordered set of name/value pairs, it begins with a left brace ({) and it ends with a right one(}). Each name is followed by a colon (:) and the name/value pairs are separated by a comma (,).
- *array*, it is an ordered collection of values. An array begins with a left bracket ([)and ends with a right one (]). The values are separated by a comma.
- *value*, it can be a lot of things, like a string in double quotes, a number, true, false, null, an object or an array. In addition, these structures can be nested.
- *string*, it is a sequence of zero or more Unicode characters, wrapped in double quotes." [**15**]

```
{
    "firstName": "James",
    "lastName": "Cork",
    "isAlive": true,
    "age": 33,
    "address": {
        "streetAddress": "25 3rd Street",
        "city": "Los Angeles",
        "state": "CA",
        "postalCode": "10066-3122"
    },
    "phoneNumbers": [{
            "type": "home",
            "number": "313 555-456"
        },
        {
            "type": "office",
            "number": "728 555-333"
        }
    ],
    "children": [],
    "spouse": null
}
```

Listing 3.1: JSON example

In the example in **List. 3.1** the different elements constituting a JSON can be spotted: e.g. "firstName":"James" is a name/value pair, "address" is an object, while "phoneNumbers" represents an array.

JSON-RPC

"In the Envoy project, a particular protocol built on JSON is used, we are talking about JSON-RPC, which is a simple remote procedure call protocol that defines only a few data types and commands and lets a system send notifications (information to the server that does not require a response) and multiple calls to the server that can be answered out of order.

In JSON-RPC protocol there is a server that implements the protocol itself and a client that is generally a software that calls only one method of a remote system. The input parameters in the request can be multiple and they can be passed as an array or an object, whereas also the method can return multiple output data.

The request to the specific method contains three fields:

- *method*, a string with the name of the method to be invoked.
- *parameters*, an object or array of values to be passed as parameters to the defined method.
- *id*, a string used for identification, to match the request with its corresponding response.

Also the response is composed of three elements:

- *result*, it contains the data returned by the invoked method. This element is formatted as a JSON object.
- *error*, an error object if there was an error invoking the method, otherwise this member must not exist.
- *id*, used to understand what response corresponds to what request, since the responses are given out of order." [**16**]

Here below an example of the request and response templates in JSON-RPC format used in Envoy (in particular the ones for the CM18 Open command).

```
{
    "jsonrpc": "2.0",
    "method": "open",
    "description": "Opens a transaction session with a CM18/OM61
   machine",
    "params": {
        "LDN": null,
        "side": null,
        "password": null
    },
    "id": null
}
```

Listing 3.2: CM18 Open request template

```
{
    "jsonrpc": "2.0",
    "method": "open",
    "description": "Opens a transaction session with a CM18/OM61
   machine",
    "result": {
        "LDN": null,
        "side": null,
        "replyCode": null,
        "description": null
    },
    "id": null
}
```

Listing 3.3: CM18 Open response template

3.2 Communication protocols

3.2.1 Protocol types

A communication protocol is a set of rules that define the syntax, the semantics and also the synchronization, allowing the communication between two or more entities, so that they are able to send and receive information through the variation of a physical quantity.

"In communicating systems, protocols have well-defined formats that specify the exact format and meaning that each command to be sent has to have, in order to obtain a certain response among all the possible predetermined responses for that specific situation.

Communication protocols represent an agreement between the entities involved into the communication, so that they are able to understand each other. To reach this agreement, a protocol has to become a technical standard. A programming language describes the same for computations, so there is a close analogy between protocols and programming languages: protocols are to communications what programming languages are to computations." [**17**]

Depending on how the information is represented, there are two types of communication protocols: **binary** and **text-based**.

In a **binary** protocol the information is represented by bytes, in fact this kind of protocol is suited to be read by a machine rather than a human being.
The advantages of binary protocols are fast transmissions and easy interpretation, since they facilitate the mechanical parsing.
Sometimes every byte is translated into its corresponding hexadecimal representation, dividing into two parts the byte and assigning to each group of four bits its hexadecimal value, thus reducing the size of the information. In this way, it becomes a hexadecimal protocol and this is the type of protocol used by some cash automation devices.

The other type of communication protocol is the **text-based** protocol or **plain text** protocol, where the information is represented by a human-readable format, usually in plain text. Commands and responses are strings of ASCII characters, terminated by a newline character and usually a carriage return character.
This kind of protocol is suited for human parsing and interpretation, in fact it is usually used when a human intervention to scan protocol contents is needed.
The ARCA machine protocol is a text-based protocol.

3.2.2 ARCA machine protocol

In order to communicate with the machines it is necessary to validate and parse the JSONs and then translate the commands from the JSON format to the specific protocol of the device.
In particular, I have studied the **SIS** protocol, which is the protocol used by ARCA machines, such as CM18, CM18T, CM18b, OM61, to name a few and now I'm going to explain it.

First of all, they use a master-slave communication, where the host connected to the machine is the master, whereas the slave is the machine controller; so the controller cannot generate commands but it can only reply to the external ones. The interfaces are standard RS232C serial interfaces with the following parameters:

baud-rate:	9600 baud
mode:	full-duplex
data-bit:	8 bit
stop-bit:	1 bit
parity:	none
number:	2 (one for each operator)
protocol:	DIN66348 data link protocol

In case two serial interfaces are used, if an host command arrives from both connectors the response will be sent on the same interface the command came in. If a communication problem happens during a procedure, the controller restarts on the same serial channel.
The protocol control procedure DIN 66348 is used to provide a secure communication between host and controller. This procedure is bi-directional, which means that the host can act as sender or receiver, but the unit will act as a sender only when replying to the host.

The communication in a DIN protocol is divided in three different phases: 1) the request, 2) the transmission and 3) the termination.

1) It is the procedure to start the communication between host and unit, it can be used by either of them and the other one can reply in different ways: the sender sends an ENQ (enquire) and the receiver can reply either NAK (not acknowledge), if it is busy, or DLE (data link escape) 0, if it is ready, or ENQ, if there is a conflict due to the fact that the target is still trying to send back some reply or info related to a previous command, in this case the host must go back to the receive mode and

link by sending DLE '0', or also a timeout can occur, so the unit retries connection twice or sends EOT (end of transmission) and returns to idle mode.

2) The host sends the command formatted in this way: "STX 'n' 'Text' ETB/ETX BCC", where STX (start of text) identifies the beginning of the message; "n" is equal to 1 for the first message block and then for the following blocks it alternates 0 and 1; "Text" is the real command for the machine (we will see later how it looks like); ETB (end of transmission block)/ETX (end of text) identifies the end of the block/message; and finally BCC (block check character) is the binary sum (module 2) of bit 0 to bit 6 of all chars in the block, excluding STX up and including ETB (or ETX), BCC parity-bit is always even and it is used for the checking of the transmitted data.
The reply can be a NAK, that means the block must be repeated, or a DLE n, the next block can be sent and "n" is equal to 1 or 0 according to the one received, or a timeout.

3) It represents the confirmation of the message transmission, in this last phase the host sends the end of transmission (EOT).

As said before, the "Text" block of the message represents the real command for the machine, it is essentially a string composed of a variable number, depending on the command, of fields separated by commas.
The first field identifies the kind of command, the second one is a sequential number, that goes from 1 to 9 and it is used in the communication to relate the host request to the device reply in case of missing synchronism, and then the other fields are the parameters needed by the machine to execute the command.

In particular, I have implemented these commands: **Open**, **Close**, **Extended-status**, **Deposit**, **Count**, **Accept Deposit**, **Set Clock**, **Read Date&Time**, **Get Unit Cover Button State**.

In the next section I will show both their format in the machine protocol and the templates that I have created to represent them in JSON format.

3.2.3 SIS commands

Open

The *Open* command is used to establish a link between the host and the machine. The operator (Left/Right) and the relative password are specified in order to define the side to be used for ejecting the banknotes and which cassettes can be used by the operator; at least one cassette should match the password.

Command syntax: *Open,num,side,code*
Where *side* is the operator side and *code* is the password.

In **List. 3.2** the JSON template for the *Open* command.

Reply syntax: *Open,num,side,repcode*
Where *repcode* is the reply code, it can be represented on 2 (CM18) or 3 (CM18T) digits and it indicates if everything is ok or if there are some errors or jams in the machine.

```
{
    "jsonrpc": "2.0",
    "method": "open",
    "description": "Opens a transaction session with a CM18/OM61
   machine",
    "result": {
        "LDN": "CM18_01",
        "side": "R",
        "replyCode": "1",
        "description": "OK"
    },
    "id": "1"
}
```

Listing 3.4: Open response example

It can be noted that the "LDN" field is always present, it stands for Logical Device Name and it is the name automatically assigned to the machine during the registration through the USB to identify it.

Close

The *Close* command closes the operator work session and removes the other serial channel from the busy status.

Command syntax: *Close,num,side*
Where *side* is always the operator side and it must coincide with the currently active side.

```
{
    "jsonrpc": "2.0",
    "method": "close",
    "description": "Closes a transaction session with a CM18/OM61
   machine",
    "params": {
        "LDN": null,
        "side": null
    },
    "id": null
}
```

Listing 3.5: Close command template

Reply syntax: *Close,num,side,repcode*
The command is executed only when the reply code is OK.

```
{
    "jsonrpc": "2.0",
    "method": "close",
    "description": "Closes a transaction session with a CM18/OM61
   machine",
    "result": {
        "LDN": "CM18_01",
        "side": "R",
        "replyCode": "1",
        "description": "OK"
    },
    "id": "1"
}
```

Listing 3.6: Close response example

Extended-status

This command is sent to get a detailed information about the status of every module of the machine, it can be sent without having sent the *Open* command before and even if the machine is in error condition.
The modules whose status is retrieved with this command are: the feeder, which is the input slot; the controller, which is the RTC (Real Time Controller), i.e. the module that guarantees the proper functioning of the machine, according to the commands received by the host; the reader, that is substantially the bill validator; the safe, that is the lower module of the machine, where there are the cassettes containing the cash; and finally the cassettes themselves.

Command syntax: *ExtStat,num*

```
{
    "jsonrpc": "2.0",
    "method": "getExtendedStatus",
    "description": "Gets the Extended Status of a CM18/OM61 machine",
    "params": {
        "LDN": null
    },
    "id": null
}
```

Listing 3.7: Extended-status command template

Reply syntax: *ExtStat,num,repcode[,1stat,2stat,3stat,4stat,Astat,Bstat,Cstat, . . . ,num_proto_cas_stat]*
Where all the fields after the *repcode* are composed of 3 digits, the first digit identifies the module (1 = feeder; 2 = controller; 3 = reader; 4 = safe; A,B,C,... = cassette A, cassette B, cassette C,...), whereas the other two show its status.

```
{
    "jsonrpc": "2.0",
    "method": "getExtendedStatus",
    "description": "Gets the Extended Status of a CM18/OM61 machine",
    "result": {
        "LDN": "CM18_01",
        "replyCode": "1",
        "description": "OK",
        "Feeder": {
            "statusCode": "40",
            "description": "Status OK"
        },
        "RTC": {
            "statusCode": "40",
            "description": "Status OK"
        },
        "Reader": {
            "statusCode": "40",
            "description": "Status OK"
        },
        "cassetteA": {
            "cassetteCode": "A",
            "statusCode": "40",
            "description": "Status OK"
        },
        "cassetteB": {
            "cassetteCode": "B",
            "statusCode": "40",
            "description": "Status OK"
        },
        "cassetteC": {
            "cassetteCode": "C",
            "statusCode": "40",
            "description": "Status OK"
        },
        "cassetteD": {
            "cassetteCode": "D",
            "statusCode": "40",
            "description": "Status OK"
        },
        "cassetteE": {
            "cassetteCode": "I",
            "statusCode": "00",
            "description": "Cassette present but switched OFF and the
    logical address matches the physical position"
        },
        "cassetteF": {
            "cassetteCode": "I",
            "statusCode": "00",
            "description": "Cassette present but switched OFF and the
    logical address matches the physical position"
        },
        "cassetteG": {
            "cassetteCode": "X",
            "statusCode": "00",
            "description": "Cassette not present"
        },
        "cassetteH": {
            "cassetteCode": "X",
            "statusCode": "00",
            "description": "Cassette not present"
        }
    },
    "id": "1"
```

```
}
```

Listing 3.8: Extended-status response example

Deposit

With this command it is possible to deposit the banknotes in safe, each banknote is deposited (if not rejected) in the cassette configured with the corresponding denomination.

Command syntax: *Dep,num,side*

```
{
    "jsonrpc": "2.0",
    "method": "deposit",
    "description": "Deposit cash into a CM18/OM61 machine",
    "params": {
        "LDN": null,
        "side": null
    },
    "id": null
}
```

Listing 3.9: Deposit command template

Reply syntax: *Dep,num,side,repcode,acc,rej,unk,[n.id,nnn], . . . ,[n.id,nnn]*
Where *acc* is the number of accepted and so deposited notes; *rej* the number of rejected notes, because they are not valid or there isn't a cassette configured with that denomination; *unk* the number of unknown notes, because the reader was not able to identify them; and then there is the list of the cassettes with their denomination and the number of deposited notes.

```
{
    "jsonrpc": "2.0",
    "method": "deposit",
    "description": "Deposit cash into a CM18/OM61 machine",
    "result": {
        "LDN": "CM18_01",
        "side": "R",
        "replyCode": "1",
        "description": "OK",
        "depositReport": {
            "accepted": "16",
            "rejected": "0",
            "unknown": "0"
        },
        "safeReport": {
                "cassetteA": {
                    "denomination": "CPC*",
                    "deposited": "7"
                },
                "cassetteB": {
                    "denomination": "CPD*",
                    "deposited": "7"
                },
                "cassetteC": {
                    "denomination": "CPE*",
                    "deposited": "0"
                },
                "cassetteD": {
                    "denomination": "CPG*",
                    "deposited": "2"
                },
                "cassetteE": {
                    "denomination": "CPH*",
                    "deposited": "0"
                },
                "cassetteF": {
                    "denomination": "CP**",
                    "deposited": "0"
                },
                "cassetteG": {
                    "denomination": "0000",
                    "deposited": "0"
                },
                "cassetteH": {
                    "denomination": "0000",
                    "deposited": "0"
                }
            }
    },
    "id": "1"
}
```

Listing 3.10: Deposit response example

Count

This command is used to only count the banknotes without depositing them. The notes are all directed either to the output slot (FIT and UNFIT) or to the reject slot (Not Recognized, Suspect, Not Authenticated).

Command syntax: *Count,num,side*

```
{
    "jsonrpc": "2.0",
    "method": "count",
    "description": "Counts cash for a CM18/OM61 machine",
    "params": {
        "LDN": null,
        "side": null
    },
    "id": null
}
```

Listing 3.11: Count command template

Reply syntax: *Count,num,side,repcode,acc,rej,unk,[n.id,nnn], . . . ,[n.id,nnn]*
Where *acc* is the number of accepted (FIT and UNFIT) notes; *rej* the number of rejected notes, because they are not valid; *unk* the number of suspect and not authenticated notes; and then there is the number of counted notes for each denomination.

With this command I have encountered a bit more difficulties due to the fact that not knowing a priori how many different denominations are present in the bundle of notes to be counted, a JSON template with all the possible denominations can't be created, so I have built a template only with the fields that are always present (**List. 3.12**) and then in the code I have written a section that fills in the JSON and adds on the fly as many fields as needed depending on the number of different denominations of the current bundle. An example of response is shown in **List. 3.13**.

```
{
    "jsonrpc": "2.0",
    "method": "count",
    "description": "Counts cash for a CM18/OM61 machine",
    "result": {
        "LDN": null,
        "side": null,
        "replyCode": null,
        "description": null,
        "countReport": {
            "accepted": null,
            "rejected": null,
            "suspect": null
        }
    },
    "id": null
}
```

Listing 3.12: Count response template

```
{
    "jsonrpc": "2.0",
    "method": "count",
    "description": "Counts cash for a CM18/OM61 machine",
    "result": {
        "LDN": "CM18_01",
        "side": "R",
        "replyCode": "1",
        "description": "OK",
        "countReport": {
            "accepted": "23",
            "rejected": "0",
            "suspect": "0"
        },
        "notesReport": {
            "CPCA": "1",
            "CPDA": "3",
            "CPGA": "8",
            "CPIA": "11"
        }
    },
    "id": "1"
}
```

Listing 3.13: Count response example

Accept Deposit

The *Accept Deposit* command allows to accept all the deposit operations done until that moment, so that it is not possible anymore to execute undo deposit and all the deposited banknotes are kept in safe until a withdrawal operation is performed.

Command syntax: *AccDep,num,side*

```
{
    "jsonrpc": "2.0",
    "method": "acceptDeposit",
    "description": "Accept Deposit for a CM18/OM61 machine",
    "params": {
        "LDN": null,
        "side": null
    },
    "id": null
}
```

Listing 3.14: Accept Deposit command template

Reply syntax: *AccDep,num,side,repcode,[n.id,nnn],...,[num_proto_cas.id,nnn]*
Where after the reply code (*repcode*), as for the *Deposit* reply, there is the list of the cassettes with their denomination, but this time with the total number of deposited notes until that moment.

```
{
    "jsonrpc": "2.0",
    "method": "acceptDeposit",
    "description": "Accept Deposit for a CM18/OM61 machine",
    "result": {
        "LDN": "CM18_01",
        "side": "L",
        "replyCode": "1",
        "description": "OK",
        "safeReportTotal": {
            "cassetteA": {
                "denomination": "CPC*",
                "deposited": "2"
            },
            "cassetteB": {
                "denomination": "CPD*",
                "deposited": "2"
            },
            "cassetteC": {
                "denomination": "CPE*",
                "deposited": "0"
            },
                "cassetteD": {
                    "denomination": "CPG*",
                    "deposited": "2"
                },
                "cassetteE": {
                    "denomination": "CPH*",
                    "deposited": "0"
                },
                "cassetteF": {
                    "denomination": "CP**",
                    "deposited": "0"
                },
                "cassetteG": {
                    "denomination": "0000",
                    "deposited": "0"
                },
                "cassetteH": {
                    "denomination": "0000",
                    "deposited": "0"
                }
            }
        },
        "id": "1"
}
```

Listing 3.15: Accept Deposit response example

Set Clock

This command is used to initialize the time keeper components of the RTC of the machine, setting date and time.

Command syntax: *SetClock,num,[reg],sec,min,hour,dow,dom,mon,year*
Where *reg* is an optional parameter and corresponds to the control register; *sec* are the seconds; *min* the minutes; *hour* the hours; *dow* the day of the week (01 = Monday,...,07 = Sunday); *dom* the day of the month; *mon* the month; and *year* the year.

```
{
    "jsonrpc": "2.0",
    "method": "setClock",
    "description": "Set clock with seconds, minutes, hours(24), day
   of week(Monday=1), day of month, month, year",
    "params": {
        "LDN": null,
        "seconds": null,
        "minutes": null,
        "hours": null,
        "dow": null,
        "dom": null,
        "month": null,
        "year": null
    },
    "id": null
}
```

Listing 3.16: Set Clock command template

Reply syntax: *SetClock,num,repcode*

The response is not very interesting to show since it only returns the reply code with the corresponding description.

Read Date&Time

With this command it is possible to read the date and time of the internal time keeper of the machine.

Command syntax: *ReadDT,num*
In this case there are only fixed parameters, except for the sequential number, so the command template will only have the "LDN" and the "id" fields.

Reply syntax: *ReadDT,num,repcode,reg,sec,min,hour,dow,dom,mon,year*
Where, after the reply code, all the fields are exactly the same parameters set in the *Set Clock* command.

```
{
    "jsonrpc": "2.0",
    "method": "readDateTime",
    "description": "Reads the date and time of a CM18/OM61 machine",
    "result": {
        "LDN": "CM18_01",
        "replyCode": "1",
        "description": "OK",
        "controlRegister": "00",
        "hour": "13",
        "minute": "59",
        "seconds": "48",
        "dayOfWeek": "Monday",
        "day": "06",
        "month": "09",
        "year": "21"
    },
    "id": "1"
}
```

Listing 3.17: Read DateTime response example

Get Unit Cover Button State

This command is used to check the status of all the unit covers/doors/buttons/input/output slots.

Command syntax: *GUCBS,num*
Like in the *Read Date&Time* command there are only fixed parameters.

Reply syntax: *GUCBS,num,repcode,*
Safe,Book,Cov,Feed,Inp,Rej,Lslt,Rslt,Lext,Rext,Cage,Esc,Bag,Fkp,Fks
Where *Safe* represents the status of the safe door (open/closed); *Book* (only for the CM18b) is the book status (open/closed); *Cov* the upper cover status (open/closed); *Feed* the feeder status (open/closed); *Inp* the input slot status (empty/full); *Rej* the reject slot status (empty/full); *Lslt* the left slot status (empty/full); *Rslt* the right slot status (empty/full); *Lext* the left external book button status (on/off); *Rext* the right external book button status (on/off); *Cage* (only for the CM18b) the cage status (open/closed); *Esc* (only for the CM18b) the escrow status (open/closed); *Bag* (only for the CM18b) the bag status (open/closed), because the peculiarity of the CM18b is the presence of a bag, where the banknotes are deposited, once full, this bag is sealed securely inside the unit and is ready for the CIT (Cash in Transit) pickup; *Fkp* indicates if the Cat. 2 Box is present or not (present/not present), the Cat. 2 Box is the box where the machine stores the suspect banknotes; *Fks* is the Cat. 2 Box status (empty/full).

```
{
    "jsonrpc": "2.0",
    "method": "getUnitCoverButtonState",
    "description": "Gets the Unit Cover Button State of a CM18/OM61
   machine",
    "result": {
        "LDN": "CM18_01",
        "replyCode": "1",
        "description": "OK",
        "safeDoor": "closed",
        "book": "closed",
        "cover": "closed",
        "feeder": "closed",
        "inputSlot": "empty",
        "rejectSlot": "full",
        "leftSlot": "empty",
        "rightSlot": "empty",
        "leftExternalBookButton": "off",
        "rightExternalBookButton": "off",
        "cage": "closed",
        "escrow": "closed",
        "bag": "closed",
        "cat.2Box": "present",
        "cat.2BoxStatus": "empty"
    },
    "id": "1"
}
```

Listing 3.18: Get Unit Cover Button State response example

Chapter 4

Development experience

In this chapter I will explain what I have carried out as my own contributions to the Envoy project. I will recap all what I have actually done from the very initial phases to the final obtained results, with the implementation of the code to manage the SIS protocol, thus integrating the CM18 into Envoy.

First of all, I have cloned locally on my PC the Git repository containing the source codes and all the files needed for the development of the Envoy project. Then, since I have chosen CodeLite as IDE to write code, it has been necessary to create the proper workspace for CodeLite, starting from the *CMakeLists.txt* file and to do this I have run the following command from the command line:
cmake -G "CodeLite - MinGW Makefiles" -S . -B .\codelite
This command executes CMake generating a CodeLite workspace in the folder "codelite", starting from the Makefiles that are present in the current folder and setting MinGW as software development environment to use GCC as compiler.

Once created the CodeLite workspace, I have started to have a look at the already existing code that was developed to integrate two machines produced by Fujitsu and Hitachi, in order to understand the structure of the application and in particular how the classes for the management of the Fujitsu protocol and the Hitachi protocol were built.
Moreover, I have run the application communicating with this two machines while looking at the log file, so that I have been able to see the order in which the methods are called, thus reconstructing the flow of the application.

During this "reconnaissance" phase I have noticed that a few methods to communicate with the CM18 were already implemented, but they were part of the general class that handles all the different types of protocols. So, I have implemented a, so

called, **code refactoring**, which is a technique, belonging to the Agile methodologies, applied to improve some non-functional features of the software, like the readability, maintainability, reusability, extendibility of the code and the reduction of its complexity.
With this code refactoring I have created a specific class for the SIS protocol, the *SisProtocol* class, moving those already existing methods into this new class and creating some methods for the protocol translation, because, while the Fujitsu and the Hitachi protocols are hexadecimal protocols, the SIS protocol is a text-based protocol, as seen in **Sec. 3.2.2**.

Then, once developed the class to handle the SIS protocol, I have implemented the *SisCommandBuilder* and *SisResponseBuilder* classes, which expose respectively the interface used by every class responsible for the builder of a command and the interface used by every class responsible for the builder of a response.

At this point, I have started to develop one by one the classes for each command and the ones for each response. The classes for the commands build the command to be sent to the machine, according to the machine protocol, starting from the JSON format; while the classes for the responses generate the JSON file, translating the response received from the machine.
Each of these classes was represented by a ticket on Jira and for each of them I have followed the same development flow showed in **Sec. 2.3.1** and that I am going to synthesize here below:

1. creation of a new branch for the new feature (*git checkout -b*)
2. code writing to implement the command/response
3. addition to the Stage Area of the modified/new files (*git add*)
4. commit creation for the new code (*git commit*)
5. publication of the changes to the remote repository (*git push*)
6. creation of a "pull request" on BitBucket to merge the new branch to the main branch, called *develop*
7. checking of the building executed by Jenkins to make sure that there are no errors
8. moving to the main branch locally (*git checkout develop*)
9. update of the main branch locally (*git pull*)
10. deletion of the branch created for the new feature (*git branch -d*)

The point 2 includes also the creation of the JSON templates. In **Sec. 3.2.3** it is possible to see all the JSON templates that I have built for the commands, with their relative responses, which I have implemented.

After having developed all the code necessary for the integration of the CM18 into Envoy, I have updated the *CMakeLists.txt* file, as shown in **Sec. 2.4.2**, to build the code, creating the DLL that manages the SIS protocol.
Then, in order to be able to run the application, I have written a script that copies the executables and all the DLLs, created with the build phase, into the installation path of Envoy:

```
@echo off
echo.
echo "Copy over libUsbLink.dll..."
copy /B /Y libUsbLink.dll C:\EnvoyRPC\api\lib\
echo.
echo "Copy over libCommLink.dll..."
copy /B /Y libCommLink.dll C:\EnvoyRPC\api\lib\
echo.
echo "Copy over libFujitsuProtocol.dll..."
copy /B /Y libFujitsuProtocol.dll C:\EnvoyRPC\api\lib\
echo.
echo "Copy over libHitachiProtocol.dll..."
copy /B /Y libHitachiProtocol.dll C:\EnvoyRPC\api\lib\
echo.
echo "Copy over libSisProtocol.dll..."
copy /B /Y libSisProtocol.dll C:\EnvoyRPC\api\lib\
echo.
echo "Copy over libProtocol.dll..."
copy /B /Y libProtocol.dll C:\EnvoyRPC\api\lib\
echo.
echo "Copy over libJSONTranslator.dll..."
copy /B /Y libJSONTranslator.dll C:\EnvoyRPC\api\lib\
echo.
echo "Copy over libutility.dll..."
copy /B /Y libutility.dll C:\EnvoyRPC\api\lib\
echo.
echo "Copy over cgi executable ..."
copy /B /Y api.exe C:\EnvoyRPC\api\api.cgi
echo.
echo "Copy over stub.exe ..."
copy /B /Y stub.exe C:\EnvoyRPC\api\ac\generic\stub.cgi
```

I have set this script as a post-build action within CodeLite, so that every time I built the code, this script was run automatically and I was able to run the application immediately, testing the new piece of code.

Regarding the unit testing phase, I have written the tests for all the commands and the responses using the Boost.Test library and then I have created a script that runs all the tests and generates a file, in XML format, with all the results:

```
@echo off

SET TEST_DIR=%cd%

echo "Copy over libUsb_1.0.dll ..."
cd
copy /B /Y ..\..\..\communications\usb\dependencies\lib\windows\x86\
    libusb-1.0.dll .
echo.
echo "Copy over rlm1402.dll ..."
copy /B /Y ..\..\..\utility\dependencies\lib\windows\x86\rlm1402.dll
    .
echo.

echo.
REM ########################################
REM # Run tests
REM ########################################
if exist test_all.exe (
echo "Running tests..."
%TEST_DIR%\test_all.exe --log_format=JUNIT --log_sink=
    unit_test_results_windows.xml
echo "After test, folder listing for %cd%:"
dir
) else (
echo "Build failed - exiting"
cd %TEST_DIR%
exit /B 1
)

if exist unit_test_results_windows.xml (
echo "Results written to %TEST_DIR%\unit_test_results_windows.xml"
) else (
echo "ERROR: Did not create Windows test result file. Check for
    runtime errors, such as missing libraries"
)

cd %TEST_DIR%

exit /B 0
```

Chapter 5

Customer support

During the development phase for the integration of the CM18 into the Envoy project, we have started the delivery of the software, to drive the other already integrated devices, to a couple of customers in the USA.

In this occasion, I have had the opportunity to collaborate with the customer support team of ARCA, experiencing an integration support with the final customer.

In particular, we have received an e-mail from a customer saying that he was experiencing an issue while sending the *Dispense* command to a Fujitsu machine through cURL, which is a command-line tool.

He was trying to send the following cURL script to communicate with the machine:

```
curl -X POST -d '{"jsonrpc":"2.0","method":"dispense","description":"
    foo","params":"LDN":"Fujitsu_01","retries":"1","rejects":"1","
    countnotes":[{"cassette position":"1","quantity":"1"}]},"id":"1"}'
     -H 'Content-Type: application/json' http://localhost:8081/api/ac/
    fujitsu/dispense.cgi
```

And he was getting the following error:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>500 Internal Server Error</title>
</head><body>
<h1>Internal Server Error</h1>
```

```
<p>The server encountered an internal error or
misconfiguration and was unable to complete
your request.</p>
<p>Please contact the server administrator at
 you@example.com to inform them of the time this error occurred,
 and the actions you performed just before this error.</p>
<p>More information about this error may be available
in the server error log.</p>
</body></html>
```

To find the source of the error we have analysed the log file of Envoy, following the flow of the application. We were looking for the point in which the application stopped, i.e. during the execution of what method.
Thanks to this accurate study phase, we have discovered that the problem was in the creation of the JSON. And using JSONLint, an online JSON validator, we have found out that the error was being caused simply by a missing curly bracket in the JSON sent through the cURL script.

This brief experience has been the occasion for me to prove the actual simplicity of the integration of this software product.

Chapter 6

Conclusions

With this thesis project I have had the opportunity to discover, learn and make use of all the most popular tools belonging to the Agile and DevOps methodologies. Methodologies that are becoming more and more widespread, especially in the context of the development of software products.

These tools have helped me to manage the whole workflow, from the initial planning phase, to the subsequent development phase, to the testing phase, till the final software release.
Jira has been very useful for planning the entire work, subdividing it in smaller tasks and week by week choosing which tasks to implement in order to complete the work.
Instead, in the development phase, I have made an intense use of Git, understanding how it works, thanks to the fact that I have run its commands from the command line, and realizing how it really simplifies the collaboration between programmers working on the same project.
Along with Git, I have experienced the power of Jenkins, in fact every time that I committed and pushed a new piece of code to the remote Git repository of the project, it built the entire project and ran the tests. In this way, Jenkins allows the continuous integration, automating the whole process and thus reducing the release time of the software product.
Moreover, since I have programmed in C++, I have had the chance to deepen my knowledge in object-oriented programming and to put in practice most of its concepts. I have also figured out a new way of building the code through CMake and Make: these tools, in fact, help to automate the build phase, optimizing times.
The testing phase is another important aspect of the software development flow that I have had the occasion to study, in particular creating the tests to perform unit testing with the Boost.Test library.

I have also had the opportunity to discover the cash automation devices, understanding how they actually improve the operational efficiency, the productivity and also the security in the cash handling within the financial institutions.
At the same time, studying how these machines work, I have run into the complexity of the various machine protocols, realizing the real advantages that an approach, like the one used in the Envoy project, can bring into the integration of these devices into software applications, with the communication performed through commands and responses in JSON format, a simple and human-readable format.

Furthermore, thanks to the collaboration done with the customer support team, in consequence of the first release of Envoy, I have been able to understand how the issues with the customers are addressed, once the software has been delivered.

In conclusion, thanks to the work done, starting from the development up to the delivery to the customer, I have really experienced how this approach allows the customer to reduce the integration times of these machines.
And for this reason, the future developments for the Envoy project, besides the completion of the CM18 integration with the implementation of the remaining commands, will certainly be the integration of other machines.
In fact, I am already working on the integration of a new machine, that differently from the CM18, which communicates over USB, it communicates via Ethernet.

Bibliography

[1] https://arca.com/company/arca-story
[2] https://arca.com/solutions/cash-recycling
[3] https://relevant.software/blog/agile-software-development-lifecycle-phases-explained/
[4] https://vintank.com/agile-software-development-life-cycle-explained/
[5] https://www.redhat.com/en/topics/devops/what-is-ci-cd
[6] https://italiancoders.it/wp-content/uploads/2018/01/devops-process.png
[7] https://www.jenkins.io/doc/book/pipeline/syntax/
[8] https://www.alibabacloud.com/blog/how-to-select-a-git-branch-mode_597255
[9] https://www.claudiobattaglino.it/2020/01/07/git-schema-di-funzionamento/
[10] https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow
[11] https://www.indeed.com/career-advice/career-development/what-is-object-oriented-programming
[12] https://cmake.org/overview/
[13] https://www.jetbrains.com/help/clion/unit-testing-tutorial.html#basics
[14] https://en.wikipedia.org/wiki/JSON
[15] https://www.json.org/json-en.html
[16] https://en.wikipedia.org/wiki/JSON-RPC
[17] https://en.wikipedia.org/wiki/Communication_protocol

Acknowledgements

I would like to thank my supervisor, Professor Massimo Violante, for giving me the chance to carry out this thesis project, putting me through to ARCA. I take this opportunity to thank all my colleagues, who made me feel very welcome and were always on hand to answer questions and offer help with any problems, since the first days.

I would like to express my sincere gratitude to my co-supervisor, Teodoro, who assisted and guided me throughout this project. He was always available to clear out any of my doubts from the study phase to the implementation phase, till the finalization of the project. Moreover, he gave me lots of valuable advice that will definitely be useful to me in my career.

I would like to offer my special thanks to my family, in particular I appreciate all the support I received from my mother in all these years of study. And I am deeply grateful to my maternal grandparents, who have always believed in me investing in my studies, thus allowing me to achieve this important goal in my life. A special thought goes to my father, who protects me from up there and gives me the strength to reach all my objectives.

I wish to show my appreciation to all the friends that I met at the university residence in Turin, especially to my roommate Nicolò, with whom I shared two years and a half of my life in Turin, and Daniele, with whom I shared a lot of fun moments and interesting experiences.

I wish also to extend my thanks to Ludovico, Massimo, Alessandro, Francesca and Giuseppe, friends and colleagues of the Bachelor's Degree, who shared with me all the moments at Politecnico, including afternoons of intensive study, but also funnier moments, like dinners and parties.

I would also like to thank all my friends since the days of the high school, in particular Davide, Federico, Simone, Samuele, Giulio, Francesco, Alberto and Nicolò, who, to a greater or lesser extent, contributed to making these five demanding years of university less tough.

And, finally, last but not least, I would like to offer my sweet thanks to Elisa, a special friend who, despite the distance that separates us, stood by me and supported me during these months of thesis.