POP-Python report

Release 1

Jonathan Stoppani

February 04, 2011

CONTENTS

CHAPTER

ONE

INTRODUCTION

Le secret du changement consiste à concentrer son énergie pour créer du nouveau, et non pas pour se battre contre l'ancien.

—Dan Millman

This document presents the work done during the semester project of the 5th semester at the College for Architecture and Engineering of Fribourg in Switzerland. This project was – since the beginning and as better explained later on – a moving target including different goals which evolved during the development process.

In this first chapter the initial problematic is described, the different intermediary and final goals are exposed and the structure of this report is explained in detail.

The citation opening this chapter wants to highlight the fact that the different goals modifications which occurred during the development aren't necessarily due to a wrong approach to the problematic but instead the result of a better understanding of the problem and the involved needs for a more detailed knowledge of it.

1.1 Problematic

The POP-C++ language and, most recently, its POP-Java counterpart, constitute a comprehensive object-oriented system to easily develop distributed systems on the Grid.

The goal of these languages is to offer an extension to the original language syntax (C++ and Java in these cases) which introduces a couple of new keywords to define different aspects bound to a transparent object distribution, such as the requirements of a certain object or the semantics of a method call (including, for example, the concurrency level).

The following code snippet shows the definition of a simple object which – once compiled – is automatically distributed thanks to the POP-C++ runtime:

```
parclass POPCobject {
public:
    POPCobject() @{od.memory(1024);};
    conc sync int add(int a, int b);
};
```

Although not complete, the preceding definition offers an insight of the few and small changes necessary to convert a C++ object into a POP-C++ one. In this case, we require the target node (i.e. the node on which the object will be executed) to offer at least 1024 MB of memory and define the semantics of the call to be *concurrent* and *asynchronous* [1].

[1] Refer to the POP-C++ manual for further information about the real syntax and the different call semantic meanings.

The initial goal of this project was to effectuate a feasibility study and provide some proofs-of-concept of the applicability of the POP model to the Python programming language. The Python counterpart should share the same syntactical concepts and be easily applicable to already existing programs without the need for large code changes.

Furthermore, compatibility between existing POP-C++ objects and POP-Python objects should be leveraged to be able, for example, to call a parallel object written in C++ from an object written in Python (and vice versa).

Although a little work was done in this direction, different aspects lead quickly to a redefinition of the goals. These iterations are better discussed in the following section.

1.2 A moving target

The previous section dealt with an high-level overview of the project application and its initial goal. As already anticipated, the goals of the project changed a couple of times during its development; this section better discusses the reasons for such changes and justifies the newly set project aims.

During the first phase of the project, namely the analysis of the actual POP-C++'s session and presentation OSI layers, the absence of any form of documentation of the actual protocol rapidly became a blocker and more and more hours were spent in the tentative to dissect the different TCP payloads.

As the POP model was conceived to be as general as possible and the effective underlying communications never analyzed, the number of different established TCP connections and the different method calls to offer a transparent high-level interface grew to a quite large number, making the manual analysis of the protocol a cumbersome and error prone task.

In the name of the "*Good programmers are lazy*" rule, a simplistic tool to automatically analyze the transmitted data was put in place. This first version of the tool simply analyzed and decoded the frame exchanges between some peers by reading an XML file exported from a manually executed wireshark[2] measure.

With this tool at hand, the real complexity of the different connections and method calls became evident: a simple application which instantiates one remote object and calls a method on it establishes more than 35 connections.

The *figure 1.1* shows an example of the visualization of a measure using this tool.

Instead of trying to build yet another POP implementation upon a not yet understood communication protocol which was never documented, the goals of the project were changed and adapted to the following: **Define a formal protocol specification for the POP model**.

The definition of a protocol and the documentation of the different transactions needed to execute a certain action opens the doors firstly to a simpler and faster implementation of the model in whatever language of choice and, secondly, allows to optimize the current POP-C++ implementation in order to reduce the number of TCP connections and method calls by removing the ones due to an excessive generalization of the model.

Sadly, once again, the effective complexity and the excessive verbosity of the transactions made even the simplistic XML based analyzer yet another tool introducing an additional manual step to the measuring process. Furthermore, the need to be able to differentiate between local-local and local-remote communications mades necessary the installation of the POP runtime on different machines, making this simplistic tool useless

To overcome these limitations and provide a more direct path between a POP application and the representation of its data exchanges, the simple measuring tool was rewritten from ground up to be able to do all the necessary tasks, from VM creation and their provisioning to the measuring of an application and the reporting of the results in an analysis oriented form.

With the developing and refinement of this tool taking some time, and after judging the results it was able to produce as interesting, the second goal was superseded by the development of the POP Analysis Suite; a suite of tools to measure and analyze POP communications between different and automatically set up virtual machines.

The *figure 1.2* shows an example of the visualization of a measure using the POP Analysis Suite.

[2] http://wireshark.org/

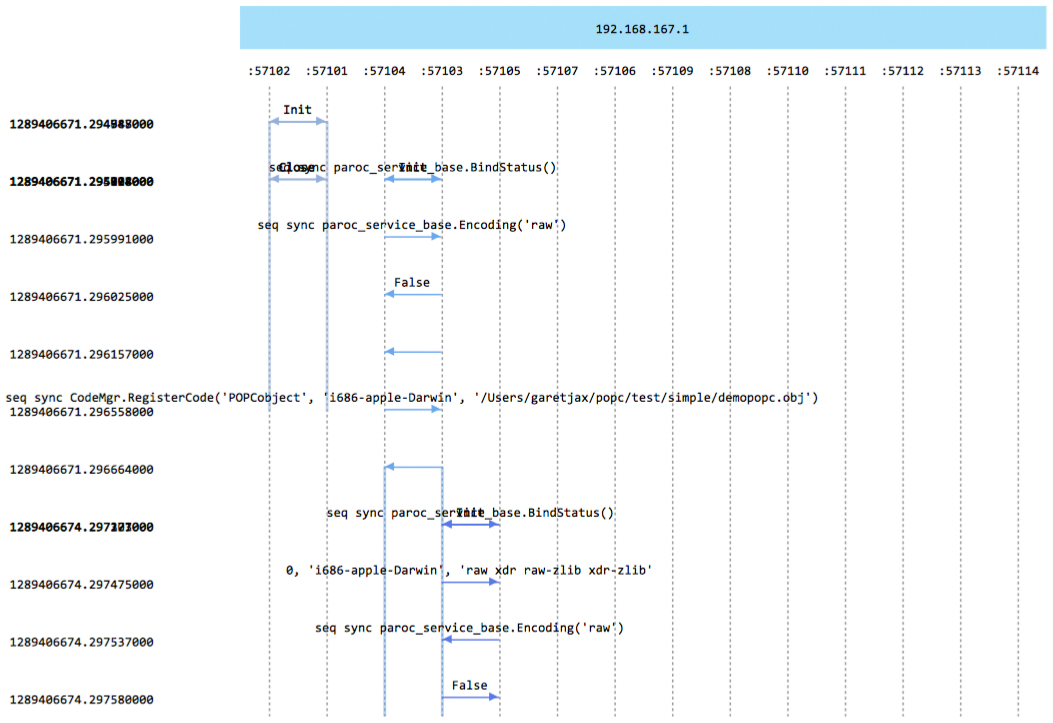
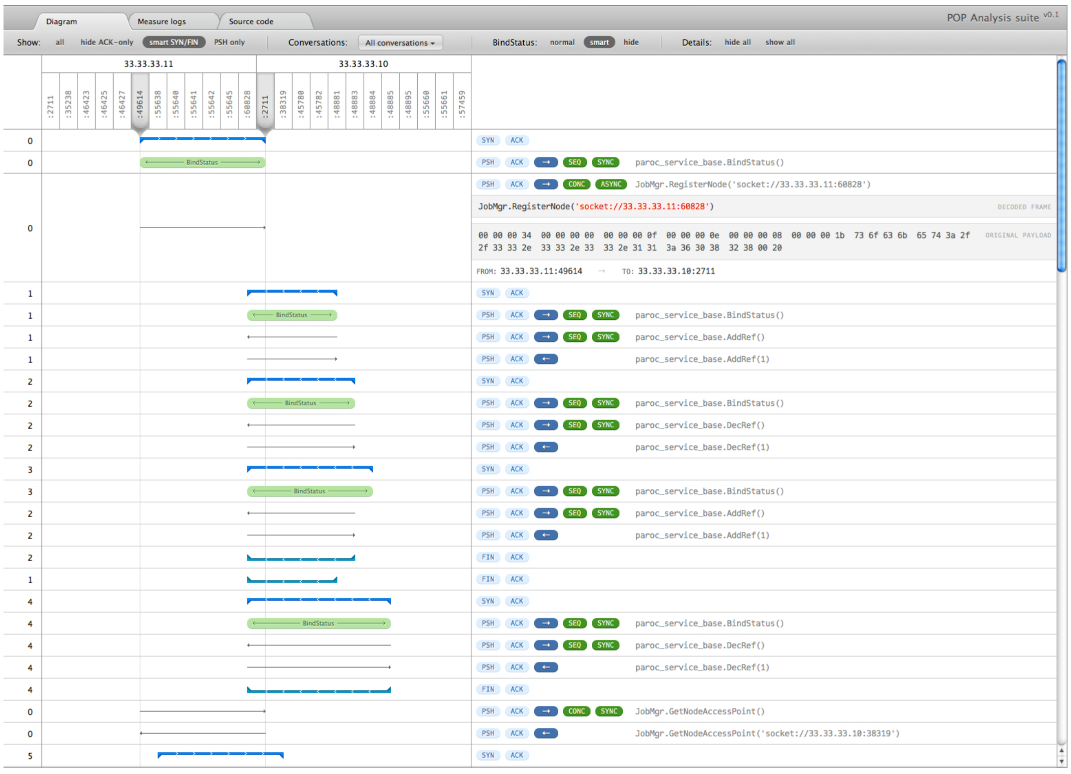


Figure 1.1: A measure as visualized by the simple XML based tool.



```
0  SYN ACK
0  PSH ACK → SEQ SYNC     paroc_service_base.BindStatus()
0  PSH ACK → CONC ASYNC   JobMgr.RegisterNode('socket://33.33.33.11:60828')
   JobMgr.RegisterNode('socket://33.33.33.11:60828')                                  DECODED FRAME
   00 00 00 34  00 00 00 00  00 00 00 0f  00 00 00 0e  00 00 00 08  00 00 00 1b  73 6f 63 6b  65 74 3a 2f   ORIGINAL PAYLOAD
   2f 33 33 2e  33 33 2e 33  33 2e 31 31  3a 36 30 38  32 38 00 20
   FROM: 33.33.33.11:49614  →  TO: 33.33.33.10:2711
1  SYN ACK
1  PSH ACK → SEQ SYNC     paroc_service_base.BindStatus()
1  PSH ACK → SEQ SYNC     paroc_service_base.AddRef()
1  PSH ACK ←              paroc_service_base.AddRef(1)
2  SYN ACK
2  PSH ACK → SEQ SYNC     paroc_service_base.BindStatus()
2  PSH ACK → SEQ SYNC     paroc_service_base.DecRef()
2  PSH ACK ←              paroc_service_base.DecRef(1)
3  SYN ACK
3  PSH ACK → SEQ SYNC     paroc_service_base.BindStatus()
2  PSH ACK → SEQ SYNC     paroc_service_base.AddRef()
2  PSH ACK ←              paroc_service_base.AddRef(1)
2  FIN ACK
1  FIN ACK
4  SYN ACK
4  PSH ACK → SEQ SYNC     paroc_service_base.BindStatus()
4  PSH ACK → SEQ SYNC     paroc_service_base.DecRef()
4  PSH ACK ←              paroc_service_base.DecRef(1)
4  FIN ACK
0  PSH ACK → CONC SYNC    JobMgr.GetNodeAccessPoint()
0  PSH ACK ←              JobMgr.GetNodeAccessPoint('socket://33.33.33.10:38319')
5  SYN ACK
```

Figure 1.2: A measure as visualized by the POP Anlysis Suite.

As you might have noticed, the comparison of the output of the previous XML based tool with the output of `pas` (the command line entry point for the POP Analysis Suite) immediately highlights the superiority in both layout and detail richness of the latter.

This tool, its usage and its extension capabilities are described in detail in the *PAS Manual*, which can be found in the appendices of the present report. The second chapter of this report, will focus on the development process and the decisions which were taken to led to the final result.

1.3 Structure of this document

The present report is structured in four main parts:

1. The introduction; offers an high-level view of the project, its goal and how they evolved during the development phase;
2. A chapter about the *POP Analysis Suite*, the tool built as final product of the project, containing informations about the development process and different aspects of its implementation;
3. A small chapter about the original project goal, *POP Python*, were the knowledge built all over the whole development process (i.e. the possible problematics of a Python implementation of the POP model) is assembled and which – though not anymore strictly relevant to the final project goal – can provide a good starting base for a future implementation;
4. The conclusions about the entire project and the acquired knowledge, a summary on how the different goals were reached and insights for future developments on these subject.

At the end, different appendices can be found, containing mainly technical documentation created to support the different tasks involved to reach the final goal.

The appendices contain also an index of the contents of the attached CD-ROM and the complete usage manual for the *POP Analysis Suite*.

Note: This document was created using sphinx[3] and can be built into different formats such as plain text, HTML files, PDF or many more by using the `sphinx-build` tool from the command line.

To create a local build, make sure to have the `sphinx` Python package installed and run the following commands:

```
$ cd report/rst/sources
$ make latexpdf  # or any other format; run make without arguments to find
                 # out about the supported ones
```

The documentation builds will then be placed in the `_build/latexpdf` subdirectory.

[3]http://sphinx.pocoo.org/

CHAPTER

TWO

POP ANALYSIS SUITE

N'attends pas que les événements arrivent comme tu le souhaites. Décide de vouloir ce qui arrive... et tu seras heureux.

—Epictète

In this chapter the final product of this project – already introduced in the first chapter – is presented in detail. Design choices are explained and the outcome described.

Note though that this chapter only approaches the problematic from the analysis point of view, the actual `pas` usage, its configuration, design, extensibility and every other aspect a possible user could have to deal with are not treated here. The *PAS Manual* which can be found in the appendices treats these subjects extensively from both a basic or advanced user point of view.

The present chapter is structured in three main parts: the first part describes why such a tool was needed, which problematics it has to solve and what to expect as a result; the second part deals with the actual resulting product, on how the encountered problems were solved and what it is actually capable to do; the third and last part contains a small conclusion relative to this chapter and its subject.

The citation above was chosen because it represents well the philosophy of the bottom-up approach which was used to solve the problem and reach the prefixed goals, as further explained in the next section.

2.1 The need for an analyzer

At the beginning of the project, when the goal to pursue was the development of a POP-Python implementation, a bottom-up approach was opted for: run a POP-C++ program, measure and analyze what happens on the network and try to mimic the same behavior with some Python objects.

This approach was chosen for two different reasons:

1. Firstly, no documentation about the POP protocol existed. This means that sooner or later what really happens on the network will need to be analyzed to really understand how the model is built;
2. Secondly, one of the partial requirements of the POP-Python implementation was that it had to be compatible with the existing POP-C++ infrastructure giving to a bottom-up approach – where the behavior is firstly defined and only then a suitable interface developed – much more sense than to a top-down approach, were we first define the interface and then adapt it to the real POP-C++ needs.

The natural next step in the capturing and analyzing phase was to fire up wireshark[1], execute a POP-C++ program and try to interpret the results.

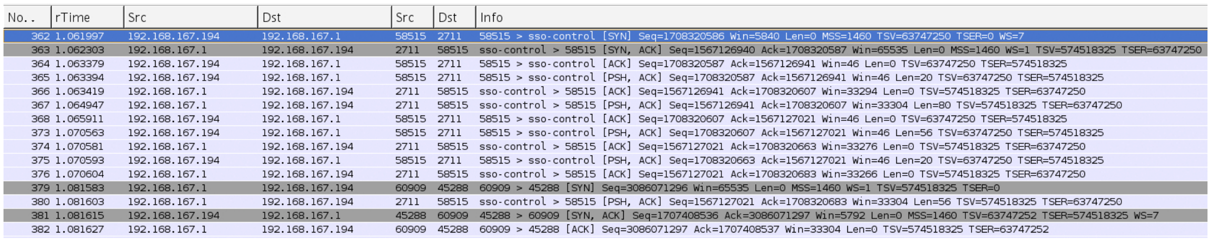
[1] http://wireshark.org/

Without a documentation of the actual structure and content of a POP message, looking at the hex encoded sequence of bytes as presented by wireshark didn't provide much informations about the semantic [2] content of the request.

Furthermore, wireshark is a great tool to analyze and debug particular details of a network transaction, but in this case an high level view of the different peers involved in the communication is needed and with many of them involved the wireshark GUI shows off its limitations (the *figure 2.1* shows an example of this visualization mode).

The first measures were done by distributing the different objects locally but this approach does not allow any-more to differentiate actual local-local (or `jobmgr` intern) communications from local-remote (involving thus the remotely created object) communications.

To solve this problem – caused by the particular POP architecture (which uses distributed objects internally too) and which cannot be overcome by a measuring tool – a virtual machine was put in place; its at this point that wireshark really reached its limits: it was not capable to discover the virtual network interfaces put in place by the virtual machines [3], making thus impossible to analyze the communications between local and distant objects.



No..	rTime	Src	Dst	Src	Dst	Info
362	1.061997	192.168.167.194	192.168.167.1	58515	2711	58515 > sso-control [SYN] Seq=1708320586 Win=5840 Len=0 MSS=1460 TSV=63747250 TSER=0 WS=7
363	1.062303	192.168.167.1	192.168.167.194	2711	58515	sso-control > 58515 [SYN, ACK] Seq=1567126940 Ack=1708320587 Win=65535 Len=0 MSS=1460 WS=1 TSV=574518325 TSER=63747250
364	1.063379	192.168.167.194	192.168.167.1	58515	2711	58515 > sso-control [ACK] Seq=1708320587 Ack=1567126941 Win=46 Len=0 TSV=63747250 TSER=574518325
365	1.063394	192.168.167.194	192.168.167.1	58515	2711	58515 > sso-control [PSH, ACK] Seq=1708320587 Ack=1567126941 Win=46 Len=20 TSV=63747250 TSER=574518325
366	1.063419	192.168.167.1	192.168.167.194	2711	58515	sso-control > 58515 [ACK] Seq=1567126941 Ack=1708320607 Win=33294 Len=0 TSV=574518325 TSER=63747250
367	1.064947	192.168.167.1	192.168.167.194	2711	58515	sso-control > 58515 [PSH, ACK] Seq=1567126941 Ack=1708320607 Win=33304 Len=80 TSV=574518325 TSER=63747250
368	1.065911	192.168.167.194	192.168.167.1	58515	2711	58515 > sso-control [ACK] Seq=1708320607 Ack=1567127021 Win=46 Len=0 TSV=63747250 TSER=574518325
373	1.070563	192.168.167.194	192.168.167.1	58515	2711	58515 > sso-control [PSH, ACK] Seq=1708320607 Ack=1567127021 Win=46 Len=56 TSV=63747250 TSER=574518325
374	1.070581	192.168.167.1	192.168.167.194	2711	58515	sso-control > 58515 [ACK] Seq=1567127021 Ack=1708320663 Win=33276 Len=0 TSV=574518325 TSER=63747250
375	1.070593	192.168.167.194	192.168.167.1	58515	2711	58515 > sso-control [PSH, ACK] Seq=1708320663 Ack=1567127021 Win=46 Len=20 TSV=63747250 TSER=574518325
376	1.070604	192.168.167.1	192.168.167.194	2711	58515	sso-control > 58515 [ACK] Seq=1567127021 Ack=1708320683 Win=33266 Len=0 TSV=574518325 TSER=63747250
379	1.081583	192.168.167.1	192.168.167.194	60909	45288	60909 > 45288 [SYN] Seq=3086071296 Win=65535 Len=0 MSS=1460 WS=1 TSV=574518325 TSER=0
380	1.081603	192.168.167.1	192.168.167.194	2711	58515	sso-control > 58515 [PSH, ACK] Seq=1567127021 Ack=1708320683 Win=33304 Len=56 TSV=574518325 TSER=63747250
381	1.081615	192.168.167.194	192.168.167.1	45288	60909	45288 > 60909 [SYN, ACK] Seq=1707408536 Ack=3086071297 Win=5792 Len=0 MSS=1460 TSV=63747252 TSER=574518325 WS=7
382	1.081627	192.168.167.1	192.168.167.194	60909	45288	60909 > 45288 [ACK] Seq=3086071297 Ack=1707408537 Win=33304 Len=0 TSV=574518325 TSER=63747252

Figure 2.1: Excerpt of a measure as presented by wireshark.

The following points indicate what do we need to know from a measure between some POP actors:

- A good overview of the different involved nodes: which node communicate with which other node.
- Details about the semantic content of the request: request type, which method on which class is called, which arguments are passed,. . .
- Being able to clearly differentiate between local-local and local-remote transactions.

2.2 Outcome

2.2.1 Simple XML based analyzer

The first version of what after became the POP Analysis Suite was a simple collection of XSL stylesheets which were manually applied to to an XML version of a measure as exported by wireshark.

The process was actually quite simple, but involved different manual steps and was thus pretty slow to re-execute:

1. Create a new measure
   (a) Start a measure in wireshark
   (b) Start the jobmgr
   (c) Run the program to test
   (d) Stop the jobmgr
   (e) Stop the measure in wireshark
2. Process the just created measure

[2] The term *semantics* is used in this context to indicate the meaning of the content encoded in the message and not the concept of *call semantics* introduced by the POP model.

[3] The tests and measures were all executed on a Mac OS X operation system. Maybe under a Linux based operating system this problem would not appear.

(a) Export the measure in XML

(b) Simplify it by applying an XSL stylesheet

(c) Generate an SVG image with a second XSL stylesheet

Further issues of this approach include the poor interactivity of the SVG output, the impossibility to automate all steps due to the different involved applications and the obvious configuration and management details such as POP-C++ installation, VM setup (startup, shutdown, provisioning,. . .).

Besides the problems described above, this tools served as a good starting point for the understanding of both the basic interactions between the different agents of a POP system and the actual construction and encoding of a POP message; the Python based parser used for this tool was indeed reused in the next iteration of the analyzer with only a few modifications.

2.2.2 POP Analysis Suite

The idea to further extend the capabilities of the simple XML based analyzer came from the reading of the documentaion of Vagrant[4], a tool for building virtualized environments with support for automatic provisioning.

Thanks to Vagrant it is possible to create one or more virtual machines based on a simple configuration script and to provision them through the utilization of chef[5] recipes.

The possibilities offered by a combination of this tools with the previous iteration promised to fully bridge the gaps left behind by the XML based tool. Thanks to Vagrant and chef it is now possible – with a single command invocation – to:

1. Create a new virtual machine from a defined base box [6];
2. Boot it up;
3. Configure different aspects such as networking and shared folders;
4. Provision it with chef, allowing thus a completely automatic installation of POP-C++ and all of its dependencies.

Building upon this solid base, the POP Analysis Suite is a tool providing command line commands to automate **all** aspects of a POP measure, from the creation of a testing environment all the long until the generation of a top of the notch measure report. The functionalities offered by the `pas` command line tool are explained in great detail in the *PAS Manual* to be found in the appendices, but the following list highlights some of the core aspects of this powerful tool:

Creation of isolated environments `pas` forces to setup a *testing environment* to host a group of one ore more measure cases, allowing to cleanly isolate VMs setups, measuring configurations and measure cases into distinct environments.

Automatic setup and provisioning of one or more VMs The default environment set up by `pas` creates and configures two VMs to run POP-C++ by installing all needed dependencies and compiling a configurable POP-C++ version from source.

It is possible to completely personalize the provisioning process through the use of the chef provisioning tool and to install different POP-C++ versions on different VMs to execute complex measuring scenarios.

Additionally, some preconfigured paths are automatically shared between the testing environment and the guest operating systems to exchange program sources, configuration files or measure results.

Built-int process management Different `pas` subcommands allow to start, stop or kill job managers on one or more virtual machines. Remote `tshark` [7] instances can also be managed through the use of `pas`.

[4] http://vagrantup.com/

[5] http://www.opscode.com/chef/

[6] A base box is a package containing a disk image with a preconfigured operating system on it. It can either already be present on the host system or automatically downloads from the internet when needed.

[7] `tshakr` is the command line equivalent of wireshark and comes bundled with each wireshark installation.

Custom measure case execution Measure cases, consisting of POP-C++ sources and a Makefile, can be deployed and compiled on each virtual machine and run on them by using simple `pas` subcommands such as `pas compile` or `pas execute`.

Advanced report generation Once a measure was run, `pas` offers tools to retrieve all relevant assets, to regroup them in a central location, to analyze them and to generate a complete and interactive, HTML based, report which can be used to analyze the measure or distributed to third parties.

Easily extensible Different commands can be composed into one for streamlined custom workflows or – if the built-in commands don't fulfill the needs for a complex measuring setup – custom commands can be created with the very same tools used all over the `pas` library.

Configurable Both the virtual machines setup and the `pas` behavior depend on configuration files which can either be left as they are and adhere to the `pas` environment conventions or customized to make `pas` behave as wanted.

As already anticipated before, all of this topics and more can be found in the *PAS Manual* at the end of the present report.

2.3 Conclusions

The final product offered by the POP Analysis Suite is a great achievement to deeply analyze the behavior of any POP-C++ program. As presented in the *Future developments* section, the analysis suite can even easily be customized to be able to measure a setup of real machines running POP-Java objects with a minimal effort.

The great flexibility and expansibility of the POP Analysis Suite lead to think that the tool was well conceived and implemented, but – apart from that – the results that this tool allows to achieve are a great step forwards in the documentation and definition of a formal POP protocol.

It is indeed not anymore needed to execute complex and tedious operations to know how exactly a given operation behaves, but the result can easily be read from a good structured report generated by running one or two shell commands.

Moreover the process is easily reproducible on the exact same setup thanks to the use of a standardized provisioning process allowing different peoples from different locations and working on disjoint systems to approach and work on the same problems; as a matter of fact, it suffices to distribute the VM configuration file and a measure case to let other people be able to reproduce exactly the same working environment.

CHAPTER

THREE

POP-PYTHON

Ce n'est pas parce que les choses sont impossibles qu'il faut les accepter.

—Alexandre Jardin

The goal of this short chapter is to resume the work done during the first part of the project to facilitate a future restart of the POP-Python project thanks to some existing documentation.

Unfortunately, the time was not enough to provide a working Python prototype, but the knowledge acquired during the project development process and the produced tools will lead to a much speedier implementation in the future.

The citation opening the chapter wants to be a fostering to implement a Python version somewhen in the future despite the problems presented in this chapter.

3.1 Possible implementation problems

A working Python version was never implemented, nonetheless, some of the problems were approached at least on a theoretical level. These are exposed in detail in the subsections below.

3.1.1 Dynamic typing

Contrary as to how happens with C++ and Java, Python uses dynamic typing and cannot thus know the data types until runtime. This is a major problem for direct compatibility with POP-C++ and POP-Java.

The two already existing implementations leverage static binding during the message encoding and decoding phase to know which type they are encoding or decoding. Unfortunately a running Python program cannot have this knowledge and is thus not able to decode a POP-C++ payload because it does not know the types of the values contained therein.

Take as an example the following method defined by an imaginary Java interface:

```java
String addToString(int a, int b);
```

Thanks to this signature, both Java and C++ know that the body of a `call` request [1] contains two integers, but as the respective Python signature would be the following:

```python
def addToString(a, b):
   pass
```

it is impossible for the Python interpreter to know the types of `a` or `b` until the invocation of the method.

Two solution seems to solve the problem:

[1] Refer to the appendices for detailed information about how a POP message is structured.

1. Adopt the same approach as used in the Python parser developed for the POP Analysis Suite and define the accepted and returned types for each method. This is the simpler approach and does not involve the modification of the POP-C++ and POP-Java sources, but requires to manually define the interfaces for all used methods.
2. Instruct the POP-C++ and POP-Java encoders to encode the data type before encoding the value itself. This approach requires the modification of the existing POP-C++ and POP-Java sources, but would solve the problem once for all dynamically typed languages.

   Furthermore this approach allows the interaction between dynamically typed objects by leveraging its full power and don't requiring the types to be statically defined somewhere.

   Note: From a rapid overview of the POP-C++ sources, it seems that the types are already passed to the encoders. If this is the case, this operation would simply mean to add a new encoder class which makes sure to write the encoded type of the data too. Furthermore, by choosing a different name for the encoding, this change would be backwards compatible.

3.1.2 Late binding

Another compatibility problem caused by the dynamic nature of Python is its leveraging of late (and dynamic) binding. In the current POP messages, the class and the method to call are identified by an ID defined statically at compilation time.

This is possible in C++ because the third method of an object will always be the third method while in Python methods can be bound or unbound from objects at runtime and this prevents to identify them with a number.

A similar problem was already encountered with the Java implementation but could be worked around because though partially supporting late binding, Java does not support dynamic binding. The workaround imposed methods to be ordered alphabetically on the POP-C++ side.

For this problem, two solutions are possible. The first being the same as the one of the previous problem: define a mapping between `classid`/`methodid` and the actual method. The second one consist in sending the class name and method signature instead of an ID to identify the method, but again, this solution involves the modification of the existing implementations and would not be backwards compatible.

Note: Probably it is possible to implement this change in a backwards compatible way by defining a special `classid`/`methodid` couple which resolves always to the same method and then adding the actual class name and method signature to the arguments. It will then be the job of this special method to read the class name and method name from the arguments and dispatch the request to the right object.

3.2 Syntax definition

The syntax definition involves mainly three aspects:

1. The definition of a `parclass`
2. The declaration of the object descriptors
3. The definition of the invocation semantics

All of these can be achieved in different ways and using standard Python techniques thanks to its dynamic nature. Each of these aspects is discussed in detail in the following subsections.

3.2.1 `parlcass` definition

There are three possible ways to mark a class as a `parclass`. There are advantages and disadvantages for each of these three syntaxes which are explained below:

1. `parclass` definition using inheritance:

   ```
   class MyClass(BaseClasse1, BaseClass2, Parclass):
      pass
   ```

   Simple to apply but pollutes the hierarchy tree and the constructor may not be called (may be worked around by merging this technique with the metaclass based one).

2. `parlcass` definition used metaclasses:

   ```
   class MyClass(BaseClasse1, BaseClass2):
      __metaclass__ = parclass
   ```

   Requires a deeper Python knowledge to be implemented correctly, but provides more explicit and better semantics as a `metaclass` is the type of a `class` and in this case we define `MyClass` to be of type `parclass`.

3. `parclass` definition using class decorators:

   ```
   @parclass
   class MyClass(BaseClasse1, BaseClass2):
      pass
   ```

   Is more limited than the metaclass approach if no metaclasses are used internally and... the syntax is ugly.

3.2.2 Object descriptors

As before, there are different ways to attach object descriptors to a class or an instance, but the different possibilities can coexists; the following snippet shows how to attach some object descriptors to a `class`:

```
class MyClass(object):

    __metaclass__ = parclass

    od.power(100, 50)
    od.memory(100, 50)
```

But things become more complicated when the descriptors of an instance to create have to be customized. The simplest way is to pass object descriptors to the `__init__` method and then let it do the initialization or let the metaclass intercept these types of arguments transparently to the user code:

```
class MyClass(object):
    __metaclass__ = parclass

    def __init__(self, myarg1, myarg2):
        self.myargs = (myarg1, myarg2)

c = MyClass(3, 5, od.memory(150, 20))
```

More exotic variations can – for example – use a class level `__getitem__` method (**this approach is highly discouraged**):

```
c = MyClass[od.power(100, 50), od.memory(100, 80)](3, 5)

# or, a variation using class attributes:

c = MyClass.power(100, 50).memory(100, 80)(3, 5)
```

Note: In Python new object descriptors can be added at runtime and newly created classes directly affected. In POP-C++ it was only possible to define values for already declared object descriptors.

3.2.3 Invocation semantics

Despite different variants can be imagined to describe method invocation semantics, a decorator based syntax seems the most logical one:

```
class MyClass(object):
    __metaclass__ = parclass

    @seq
    @sync
    def method(self):
        pass
```

To shorten the number of lines, a dynamically constructed semantics object variant could also be considered:

```
class MyClass(object):
    __metaclass__ = parclass

    @seq.sync
    def method(self):
        pass
```

As always, thanks to the dynamic features of Python, it is also possible to define the semantics at invocation time:

```
MyClass().method()          # Does a sequential synchronous call
MyClass().method.async()    # Overrides the defined semantics to do an asynchronous call
```

3.3 Conclusions

In this rapid discussion of a Python oriented view of the POP model we have seen the (probably) two major issues which will be encountered during the implementation of the model under Python but which apply to practical all dynamic languages such as Ruby, Erlang, Smalltalk or many others. Some pointers for one or more solutions are also provided, always trying to retain the maximal backwards compatibility. Solving these problems once should cover them for most languages.

After this section, an overview of the design choices made about the original syntax definition for the Python language is presented. As seen, for each of the three main annotation types more than one solution exists. It is not possible to decide about one or the other without a working implementation under the hood.

It is howsoever important to note that whatever syntax is chosen, it can always be wrapped in a function or a class to provide another variant. It is thus advised to start with the low-level stuff first and build the dynamic high-level syntax wrappers on top of it only at the end.

CHAPTER

FOUR

CONCLUSIONS

J'aime mieux dire la vérité en mon langage rustique que mensonge en un langage théorique.

—Bernard Palissy

In this last chapter the conclusions are finally draw. What was done? What wasn't? Were the prefixed goals reached or not? What can be done with this project to further pursue its development?

To these and more questions a response should be found in this chapter. Maybe its a good one or maybe not, but as introduced by the opening citation, this is my chapter. This is me speaking about my project and so, well... it will be a subjective thing.

4.1 Future developments

Future developments are both encouraged on the Python implementation side and on the POP Analysis Suite part.

The `pas` command line utility is really intended to be extended and adapted to more complex setups. Commands and utilities developed for this purpose can easily be merged back in in the tool for others to use.

Possible future developments for the `pas` tool include:

1. Built-in support for the POP-Java implementation;
2. Better streamlined integration with existing machines;
3. Automatic management of the `Vagrantfile` by `pas` through the use of specific commands (i.e. `pas vm add`);
4. Better decoupling of the different commands and lessen the assumptions about the guests (for example don't use shared folders for data transfer as they are hard to setup on real system, use `scp` instead);
5. Added functions to the visualization interface (for example, support searching inside the decoded messages or display decoding errors);
6. ...

On the other side, the bases for a successful Python implementation were laid out. It is now possible to exactly know which methods have to be called on the remote peers and in which sequence. Probable problems were also highlighted and different possibilities for the POP-Python syntax presented.

Probably, before continuing the work on a Python implementation, the first task to accomplish is the formal definition of the POP application level protocol, through the definition of use cases such as object registration, method calls, reference management,...

Once a formal protocol is available, it will be time to assemble the different building blocks into a working (and tested) Python implementation.

4.2 Work done

Despite changing the goal of the project multiple times, all done work always built on top of previously executed tasks, leading to almost no unnecessary discarded work. Thanks to this streamlined process a good result was however reached.

During the whole project, the following tasks were resolved:

- The current state of the POP model analyzed and some aspects condensed in technical documentation:
- A full featured dynamic decoder for the POP XDR encoding and supporting all complex POP types has been developed.
- On top of the decoder above, a basic protocol implementation [1] was built, with support for dynamically registered classes and methods.
- Different POP setups were explored and the respective virtual machine configurations created.
- A complete *POP environment* management solution was developed, based on existing virtual machine managers and configuration providers.
- On top of the decoder above, a system to generate visual reports of network level measures was put in place.
- On top of all these tasks, a complete, self-contained, integrated measuring solution was developed and documented.

By looking back and supposing to have to build a tool similar to the outcome of this project, I may safely affirm that I would need much less time.

Does that mean that some time was wasted? That unnecessary work was done? That the whole process was bad planned?

No! It simply means that during the execution of the project, I acquired new knowledge on a previously unknown topic; I dissected the informations at my disposal and assembled them to be able to produce what you can look at in the last appendix of this report.

Did I reach the project goals? Maybe, I don't know (the next section is dedicated to this topic). Did I reach my personal goals? Certainly yes!

4.3 Reached goals

In the previous section the work which was done was resumed and a short conclusion made about it. In this section the link to the original and modified project goals will be made and a more objective conclusion will be drawn.

4.3.1 Technical goals

Goal 1.1 (partially superseded by the new formulation):

> Provide a detailed analysis of the communication process between two remote POP-C++ endpoints and the underlying application level protocol(s). This study shall be detailed enough to be able to implement a distributed programming model compatible with the POP one.

This project surely provides a detailed analysis of the communication process between two (or more) POP-C++ endpoints and the tools which were built allow to easily demonstrate these concepts on real use cases.

On the other and, the application level protocols were not analyzed. Studies are provided for the lower layers of the OSI stack, such as the transport, session and presentation layers, but no analysis was done for the application level protocol.

I personally considers this goal as failed though the achieved products allow to perform a study of this type in much less time than before and with much more precise results.

[1] The protocol implementation is able to, starting from a complete POP message, invoke the right method on the right object by passing the good arguments to it.

Goal 1.2 (completely superseded by the new formulation):

> Study different possible approaches and tools to implement the POP model in Python and choose the one that best suits the task.

Although not anymore required by the specification, some results were achieved in this area both because needed for other parts for the project or because the task was approached before of the goal reformulation.

In both case the progresses made in this field are exposed in this report or in the *PAS Manual*.

Goal 1.3 (completely superseded by the new formulation):

> Conceive the Python-oriented syntax to apply the POP model and provide enough proofs of concept to illustrate the inner working.

A similar argument applies to this goal as to the previous one, this task was no more requested by the new specification but some resolved points are anyway included in the report (though no proofs of concept are provided).

Goal 1.4 (completely superseded by the new formulation):

> (if there is enough time) Begin to code a first Python implementation of the model and evaluate its performance.

For this goal, the same argument applies as for the previous twos. Different pieces which will be part of a possible future Python implementation are already used in the final project product (i.e. the decoder and the basic protocol implementation) while others are documented in the related section.

Goal 1.5 (part of the new formulation)

> Provide a fully functional and well documented implementation of the POP Analysis Suite.

Based on the previous work, this new goal sets the aim of the project to a complete implementation of the analysis suite. Both the final product and its documentation fully comply with the complete goal formulation. I consider this goal as fully reached.

4.3.2 Managerial goals

Goal 2.1:

> Exercise the project management skills learned in the frame of the different courses.

The acquired skills were exercised, but surely not in the best possible manner. Especially the time management and planning (further discussed in the next section) are skills for which there is always to learn. The difficulty here is that one can learn them only through experiences on real projects, such as this one.

For exactly this reason I consider this goal as achieved even though I don't have exploited my skills at their best.

Goal 2.2:

> Manage a project from its beginning to its end, while consolidating the acquired knowledge in both management and tools exploitation.

Personally I think the project was pretty well managed especially with regard to the flexibility to adapt it to a changing set of goals and the variable work load due to other parallel activities. The goal was to manage the whole project and from my point of view, I consider this goal as attained too.

4.3.3 A note about the planning

The project management process clearly includes a part of planning and time management. This task is made pretty hard from the variable workload during the whole semester and the impossibility to plan it all.

I think that while a good planning is important, not too much weight should be given to the time unit allocations. Some time management skills are surely necessary but mainly for short terms planning (maximum a week or two). For longer periods only times ratio between two tasks can give some more insight into the actual needed time (i.e. *task1* takes two times *task2* to complete), but any tentative to plan time in detail on a longer time period is intended to fail.

Especially in the academic scope, with 8 to 12 different courses to attend and each one with different expectations about the time taken by the student to revise or prepare a lab session, the coordination of the different loads in good advance becomes really difficult (life is hard eh...).

MESSAGE STRUCTURE

Every message exchanged by the POP-C++ protocol shares a common structure. This structure is illustrated for the different message types in the following images:

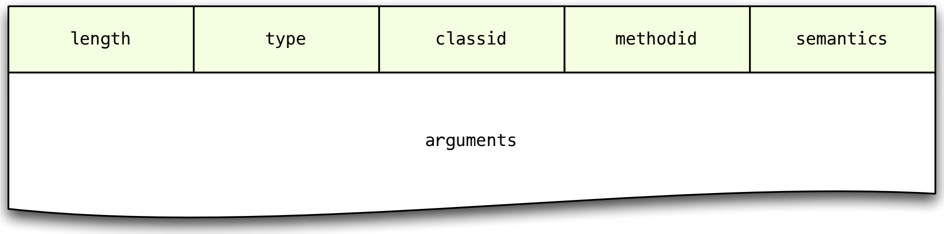


Figure A.1: Message structure for a message of type `request`.

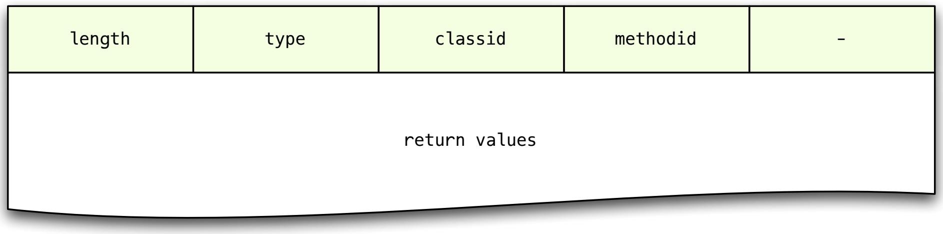


Figure A.2: Message structure for a message of type `response`.

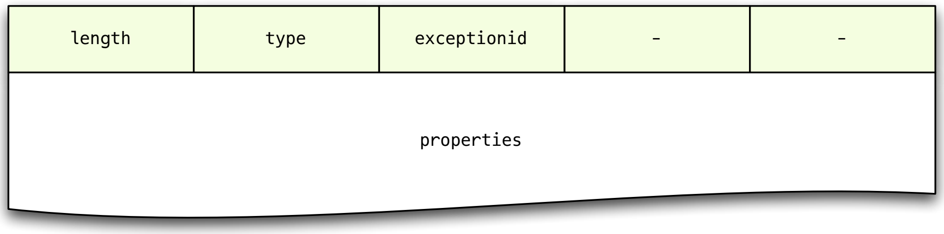


Figure A.3: Message structure for a message of type `exception`.

As anticipated above, the three message types share a common structure but there are some little differences in the interpretation of the different fields. Each header field is represented by an `integer` to which different meanings are attributed in accordance with the following descriptions:

- The `length` field contains the length of the complete message in bytes

Note: The length includes the 4 bytes of the prefix (the `length` field itself) too.

- The `type` field identifies the message type: 0 for a method call (or `request`), 1 for a return value (or `response`) and 2 for to propagate an exception (or `exception`).
- The third field contains the `classid` of the target class (or exception in the case of a message of type `exception`).
- The fourth field – only used in `request` or `response` messages – contains the ID of the method to call on the respective class.

Note: Method calls for `classid == 0` and `methodid < 10` are all handled by the `paroc_broker::ParocCall` method (which can be found at `broker_receive.cc:173`).

- The fifth and last header field is used only for method invocations and contains the semantics of the call. Refer to the appendix B for a detailed reference of how the `semantics` field is encoded.

APPENDIX

B

SEMANTICS ENCODING

The `semantics` field introduced in the appendix A is a bitmap of the different semantics flags ORed together. The defined flags are represented below along with their decimal and binary representations:

```
ASYNC         -           -     # Not defined
SEQ           0   0000'0000
SYNC          1   0000'0001
-             2   0000'0010     # No meaning
CONSTRUCT     4   0000'0100
CONC          8   0000'1000
MUTEX        16   0001'0000
```

The final representation of the `semantics` field is shown in the following representation:

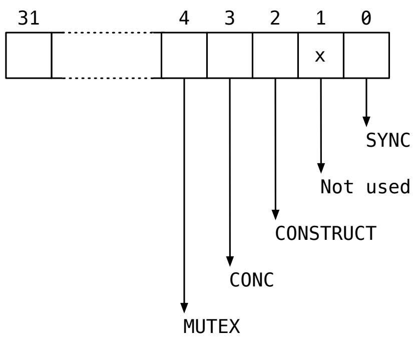


Figure B.1: Bitwise representation of the semantics `field`.

Note: A value of `0` for the semantics field, automatically implies an `ASYNC SEQ` call semantic.

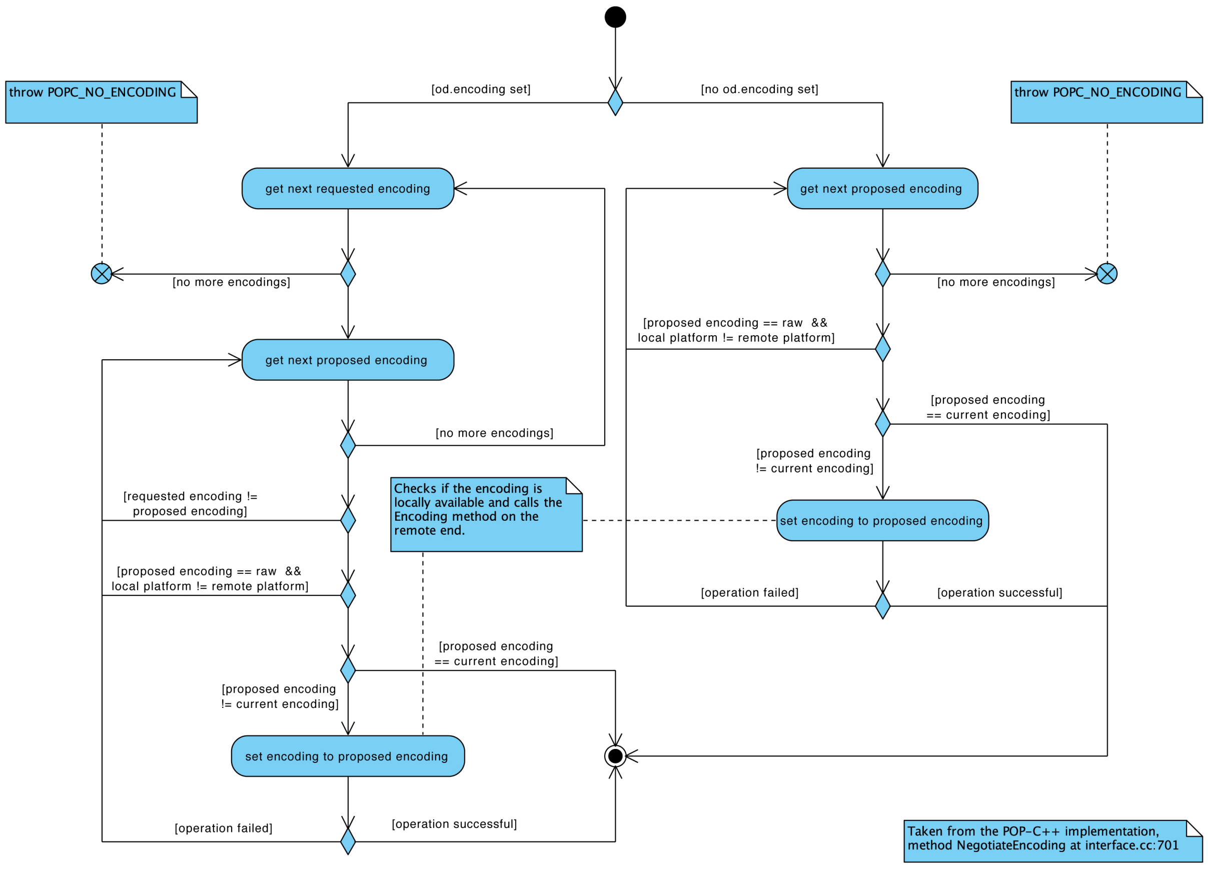
APPENDIX

C

ENCODING NEGOTIATION

The following activity diagram illustrates how the current POP-C++ implementation negotiates encoding during the `BindStatus` method call.

The actual code executing the negotiation can be found in the `interface.cc` file around the line 700, in the `NegotiateEncoding` method.



[od.encoding set]
[no od.encoding set]
throw POPC_NO_ENCODING
throw POPC_NO_ENCODING
get next requested encoding
get next proposed encoding
[no more encodings]
[no more encodings]
get next proposed encoding
[proposed encoding == raw && local platform != remote platform]
[proposed encoding == current encoding]
[no more encodings]
[proposed encoding != current encoding]
[requested encoding != proposed encoding]
Checks if the encoding is locally available and calls the Encoding method on the remote end.
set encoding to proposed encoding
[proposed encoding == raw && local platform != remote platform]
[operation failed]
[operation successful]
[proposed encoding == current encoding]
[proposed encoding != current encoding]
set encoding to proposed encoding
[operation failed]
[operation successful]
Taken from the POP-C++ implementation, method NegotiateEncoding at interface.cc:701

Figure C.1: Activity diagram of the encoding negotiation made by POP-C++.

APPENDIX

D

REMOTE METHOD INVOCATION

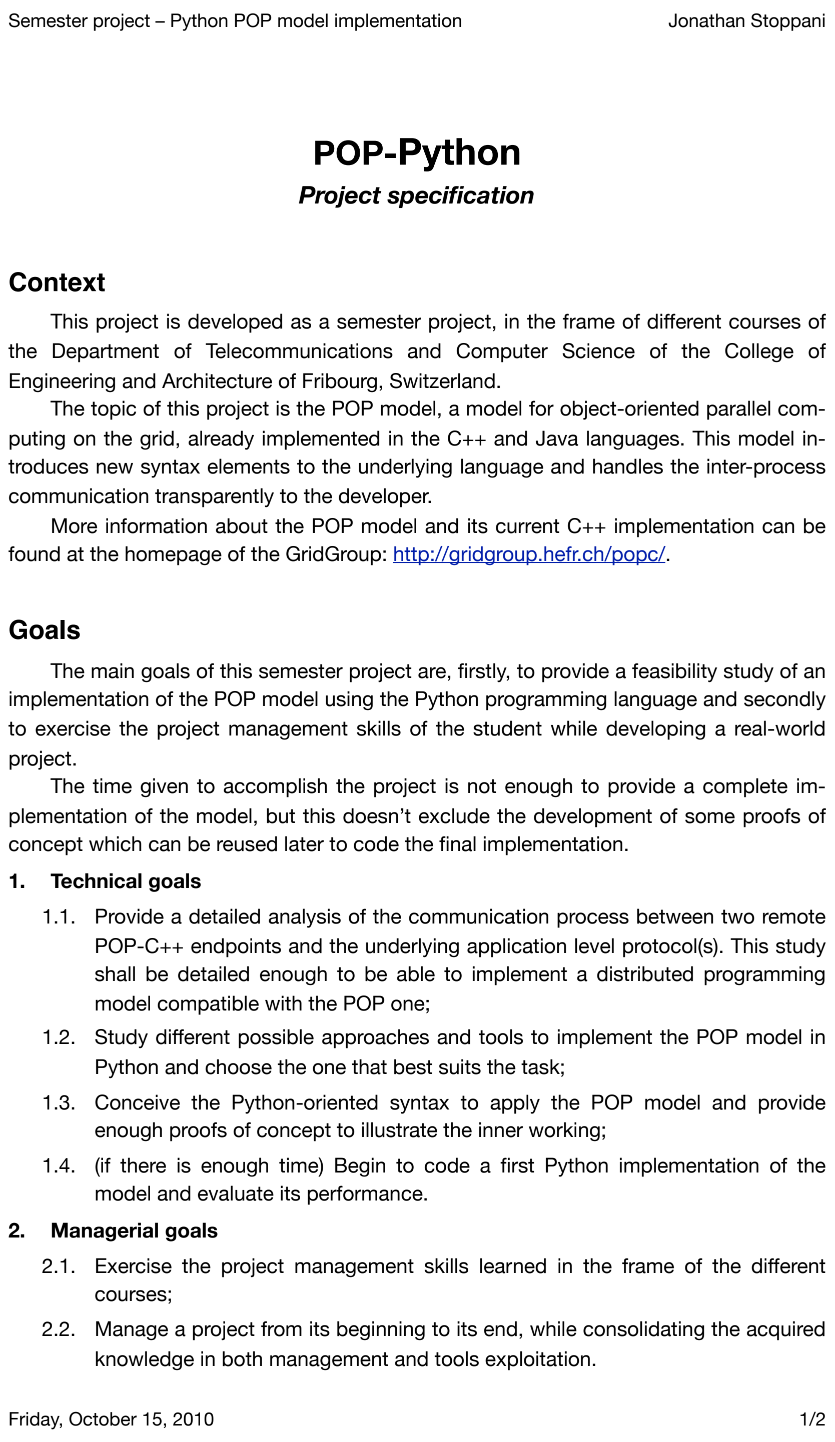
The following **pseudocode** illustrates how a remote method is invoked from the interface side point of view.

```
RTYPE Class::Method(ARGS) {
   paroc_mutex_locker __paroc_lock(_paroc_imutex);

   __paroc_buf->Reset();

   paroc_message_header __paroc_buf_header(CLASSUID, METHODID, SEMANTICS, METHOD_NAME);
   __paroc_buf->SetHeader(__paroc_buf_header);

   for (ARG in ARGS) {
      (*__paroc_buf).Push(ARG.NAME, ARG.TYPE, 1);
      (*__paroc_buf).Pack(ARG.REF, 1);
      (*__paroc_buf).Pop();
   }

   paroc_Dispatch(__paroc_buf);

   if (ASYNC) {
      __paroc_buf->Reset();
      return;
   }

   if (!__paroc_buf->Recv(*__paroc_combox)) {
      paroc_exception::paroc_throw_errno();
   }

   paroc_buffer::CheckAndThrow(*__paroc_buf);

   RTYPE _RemoteRet;

   (*__paroc_buf).Push("_RemoteRet", RTYPENAME, 1);
   (*__paroc_buf).Pack(&_RemoteRet, 1);
   (*__paroc_buf).Pop();

   __paroc_buf->Reset();

   return _RemoteRet;
}
```

APPENDIX

E

PROJECT SPECIFICATION



Semester project – Python POP model implementation Jonathan Stoppani

POP-Python

Project specification

Context

This project is developed as a semester project, in the frame of different courses of the Department of Telecommunications and Computer Science of the College of Engineering and Architecture of Fribourg, Switzerland.

The topic of this project is the POP model, a model for object-oriented parallel computing on the grid, already implemented in the C++ and Java languages. This model introduces new syntax elements to the underlying language and handles the inter-process communication transparently to the developer.

More information about the POP model and its current C++ implementation can be found at the homepage of the GridGroup: http://gridgroup.hefr.ch/popc/.

Goals

The main goals of this semester project are, firstly, to provide a feasibility study of an implementation of the POP model using the Python programming language and secondly to exercise the project management skills of the student while developing a real-world project.

The time given to accomplish the project is not enough to provide a complete implementation of the model, but this doesn't exclude the development of some proofs of concept which can be reused later to code the final implementation.

1. **Technical goals**
   - 1.1. Provide a detailed analysis of the communication process between two remote POP-C++ endpoints and the underlying application level protocol(s). This study shall be detailed enough to be able to implement a distributed programming model compatible with the POP one;
   - 1.2. Study different possible approaches and tools to implement the POP model in Python and choose the one that best suits the task;
   - 1.3. Conceive the Python-oriented syntax to apply the POP model and provide enough proofs of concept to illustrate the inner working;
   - 1.4. (if there is enough time) Begin to code a first Python implementation of the model and evaluate its performance.
2. **Managerial goals**
   - 2.1. Exercise the project management skills learned in the frame of the different courses;
   - 2.2. Manage a project from its beginning to its end, while consolidating the acquired knowledge in both management and tools exploitation.

Friday, October 15, 2010 1/2

Activities

This section lists a series of activities which shall be accomplished during the project development. This activities are inspired by the goals above and included in the planning.

1. Technical activities

1.1. Analyze the communication between two remote POP-C++ endpoints by defining well-know use cases and documenting the data exchange between the peers (milestone: *Protocol analysis*, due: November 4, 2010);

1.2. Collect information about different tools and libraries and compare them to identify the programming approach which best suits a POP-Python implementation (milestone: *Tools comparison*, due: October 29, 2010);

1.3. Define a POP-Python syntax in a way that integrates with the Python programming philosophy and code different proofs-of-concept to validate it (milestone: *Syntax definition*, due: November 8, 2010);

1.4. Design the overall POP-Python architecture and possibly begin to code a basic implementation of it (milestone: *Design*, due: December 13, 2010).

2. Managerial activities

2.1. Prepare the specification of the project, including the context, goals and activities needed for its completion (deliverable 7.1, due: October 15, 2010);

2.2. Prepare a planning for the whole project, based on the goals and activities exposed in the specification and keep it updated for the complete project timespan (deliverable 7.2, due: October 15, 2010);

2.3. Write all documents as well as the final report in english;

2.4. Prepare a final report in the form of a technical documentation containing a description of all the work done so that it can be used as a starting point for a future POP-Python implementation (deliverable 7.4, due: February 4, 2010);

2.5. Keep an up-to-date website were all the needed documents are available for viewing/download;

2.6. Prepare the agendas and reports for each weekly meeting.

Deliverables

15. October 2010	Delivery of the project specification.
20. October 2010	First presentation about the progress of the project.
4. February 2011	Delivery of the final documents.
Week EX1	Oral defense

APPENDIX

F

CD-ROM CONTENTS

The attached CD-ROM contains the following resources:

`report.pdf` The PDF version of the full report with all the appendices but the *PAS Manual* (the manual can be found as a separate PDF on the same CD-ROM).

`manual.pdf` The PDF version of the *PAS Manual* (an HTML version was also built and is available on the same CD-ROM).

`manual-html` The HTML version of the *PAS Manual* (a PDF version was also built and is available on the same CD-ROM).

`README` A text file containing the index of the CD-ROM contents.

`pas-source` A clone of the original pas git repository which can be found on github[1]. Contains the sources needed to build the *PAS Manual* too.

`report-source` The sphinx `rst` sources of the final report. Use these to built the report into other formats such as LaTeX or HTML.

`htdocs` A snapshot of the final version of the website of the project as it was published on the github page[2].

`documents` A folder containing all documents related to the project such as planning gantt diagrams, meeting agendas and reports, the original projects specification,...

[1]https://github.com/GaretJax/pop-analysis-suite/
[2]http://garetjax.github.com/pop-python/

APPENDIX

G

PAS MANUAL