POP-Java Documentation

Release 1.5

GRID & Cloud Computing Group

Oct 15, 2018

User manual

CHAPTER 1

Introduction and background

1.1 Introduction

Programming large heterogeneous distributed environments such as GRID or P2P infrastructures is a challenging task. This statement remains true even if we consider research that has focused on enabling these types of infrastructures for scientific computing such as resource management and discovery [FK97], [GFKH99], [CFK+88], service architecture [FKNT02], security [WSF+03] and data management [ABB+02], [SSA+01]. Efforts to port traditional programming tools such as MPI [FK98], [RFG+00], [KTIFoster03] or BSP [TDC03], [WP00], also had some success. These tools allow programmers to run their existing parallel applications on large heterogeneous distributed environments. However, efficient exploitation of performance regarding the heterogeneity still needs to be manually controlled and tuned by programmers.

POP-C++ and POP-Java are implementations of the POP (**P**arallel **O**bject **P**rograming) model first introduced by Dr. Tuan Anh Nguyen in his PhD thesis [Ngu04]. POP-C++ is an extension of the C++ programming language [KN07] and POP-Java is an extension of the Java programming language [Cle10]. The POP model is based on the very simple idea that objects are suitable structures to distribute data and executable codes over heterogeneous distributed hardware and to make them interact between each other.

Inspired by CORBA [Object Management Group01] and C++, the POP-C++ programming language extends C++ by adding a new type of **parallel object**, allowing to run C++ objects in distributed environments. With POP-C++, programming efficients distributed applications is as simple as writing a C++ programs. The POP-Java programming language extends Java and implements the same mechanisms as POP-C++.

1.2 The POP model

The POP model extends the traditional object oriented programming model by adding the minimum necessary functionality to allow for an easy development of coarse grain distributed high performance applications. When the object oriented paradigm has unified the concept of module and type to create the new concept of **class**, the POP model unifies the concept of class with the concept of **task** (or **process**). This is realized by adding to traditional sequential classes a new type of class: the **parallel class**. By instantiating parallel classes we are able to create a new category of objects we will call **parallel objects** in the rest of this document.

Parallel objects are objects that can be remotely executed. They coexist and cooperate with traditional sequential objects during the application execution. Parallel objects keep advantages of object-orientation such as data encapsulation, inheritance and polymorphism and adds new properties to objects such as:

- Distributed shareable objects
- Dynamic and transparent object allocation
- Various method invocation semantics

1.3 System overview

Although the POP-C++ programming system focuses on an object-oriented programming model, it also includes a runtime system which provides the necessary services to run POP-C++ and POP-Java applications over distributed environments.

An overview of the POP system (Both POP-C++ and POP-Java) architecture is illustrated in 1. In POP-Java, only the programming system is implemented and the runtime system is the same as the one used in POP-C++.

The POP-C++ runtime system consists of three layers: the service layer, the POP-C++ service abstractions layer, and the programming layer. The service layer is built to interface with lower level toolkits (e.g. Globus) and the operating system. The essential service abstraction layer provides an abstract interface for the programming layer. On top of the architecture is the programming layer, which provides necessary support for developing distributed object-oriented applications. More details of the POP-C++ runtime layers are given in a separate document [Ngu04].

1.4 Structure of this manual

This manual has 8 chapters, including this introduction:

- *Parallel Object Model* explains the POP model.

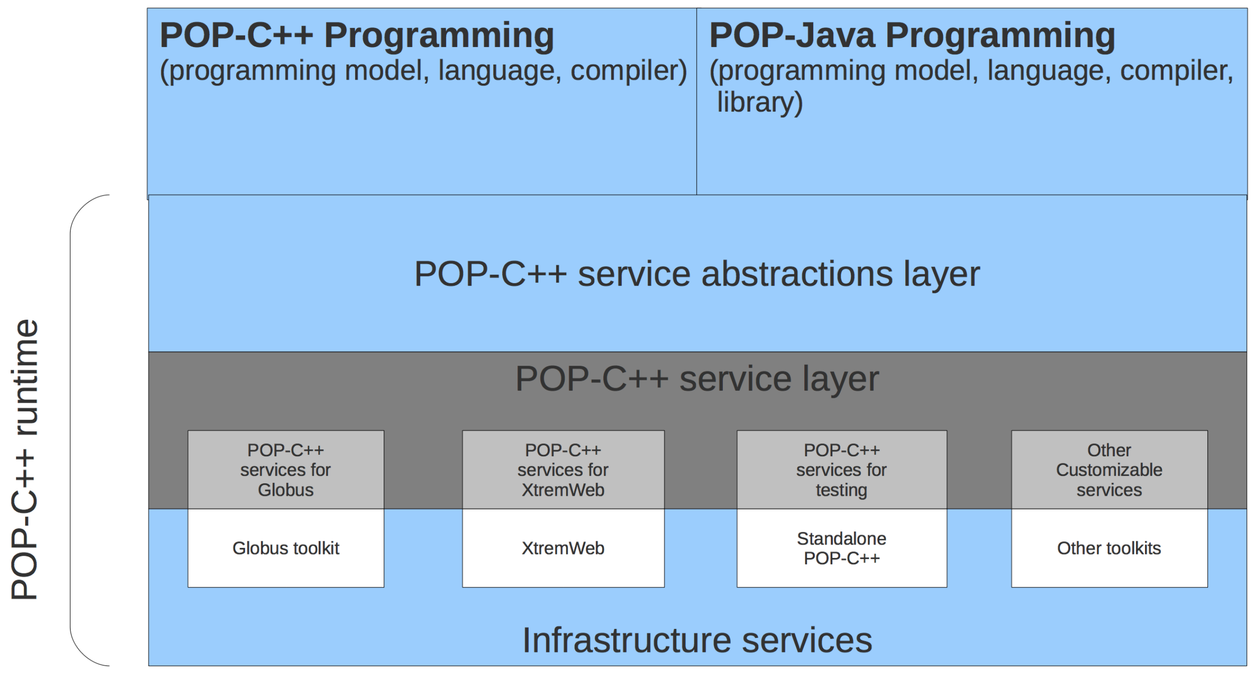

POP-C++ Programming
(programming model, language, compiler)
POP-Java Programming
(programming model, language, compiler, library)
POP-C++ runtime
POP-C++ service abstractions layer
POP-C++ service layer
POP-C++ services for Globus
POP-C++ services for XtremWeb
POP-C++ services for testing
Other Customizable services
Globus toolkit
XtremWeb
Standalone POP-C++
Other toolkits
Infrastructure services

Fig. 1: POP system architecture

- *Developing POP-Java applications* describes the POP-Java application development process.
- *Compile and run a POP-Java application* explains the compilation and the launch process of a POP-Java application.
- *Developing POP-Java and POP-C++ mixed applications* aims to describe and explain how to use of POP-C++ and POP-Java together in a same application.
- *POP-Java plugin* describes the POP-Java plugin system.
- *Installation* guides the user trough the installation process.
- Finally, *Troubleshooting* gives some hints to solve the main problems that can occur with a POP-Java application.

CHAPTER 2

Parallel Object Model

2.1 Introduction

Object-oriented programming provides high level abstractions for software engineering. In addition, the nature of objects makes them ideal structures to distribute data and executable codes over heterogeneous distributed hardware and to make them interact between each other. Nevertheless, two questions remain:

- Question 1: which objects should run remotely?
- Question 2: where does each remote object live?

The answers, of course, depend on what these objects do and how they interact with each other and with the outside world. In other words, we need to know the communication and the computation requirements of objects. The parallel object model presented in this chapter provides an object-oriented approach for requirement-driven high performance applications in a distributed heterogeneous environment.

2.2 Parallel Object Model

POP stands for *Parallel Object Programming*, and POP parallel objects are generalizations of traditional sequential objects. POP-Java is an extension of Java that implements the POP model. POP-Java instantiates parallel objects transparently and dynamically, assigning suitable resources to objects. POP-Java also offers various mechanisms to specify different ways to do method invocations. Parallel objects have all the properties of traditional objects plus the following ones:

- Parallel objects are shareable. References to parallel objects can be passed to any other parallel object. This property is described in *Shareable Parallel Objects*.
- Syntactically, invocations on parallel objects are identical to invocations on traditional sequential objects. However, parallel objects support various method invocation semantics: synchronous or asynchronous, and sequential, mutex or concurrent. These semantics are explained in section *Invocations semantics*.
- Parallel objects can be located on remote resources in separate address spaces. Parallel objects allocations are transparent to the programmer. The object allocation is presented in section *Parallel Object Allocation*.
- Each parallel object has the ability to dynamically describe its resource requirement during its lifetime. This feature is discussed in detail in section *Requirement-driven parallel objects*.

As for traditional objects, parallel objects are active only when they execute a method (non active object semantic). Therefore, communication between parallel objects are realized thank to remote methods invocation.

2.3 Shareable Parallel Objects

Parallel objects are shareable. This means that the reference of a parallel object can be shared by several other parallel objects. Sharing references of parallel objects are useful in many cases. For example, 1 illustrates a scenario of using shared parallel objects: `input` and `output` parallel objects are shareable among `worker` objects. A `worker` gets work units from `input` which is located on the data server, performs the computation and stores the results in the `output` located at the user workstation. The results from different `worker` objects can be automatically synthesized and visualized inside `output`.

To share the reference of a parallel object, POP-Java allows parallel objects to be arbitrarily passed from one place to another as arguments of method invocations.

2.4 Invocations semantics

Syntactically, method invocations on parallel objects are identical to those on traditional sequential objects. However, to each method of a parallel object, one can associate different invocation semantics. Invocation semantics are specified by programmers when declaring methods of parallel objects. These semantics define different behaviors for the execution of the method as described below (example of syntax in *Developing POP-Java applications*):

- **Interface semantics**, the semantics that affect the caller of the method:
  - **Synchronous invocation**: the caller waits until the execution of the called method on the remote object is terminated. This corresponds to the traditional method invocation.

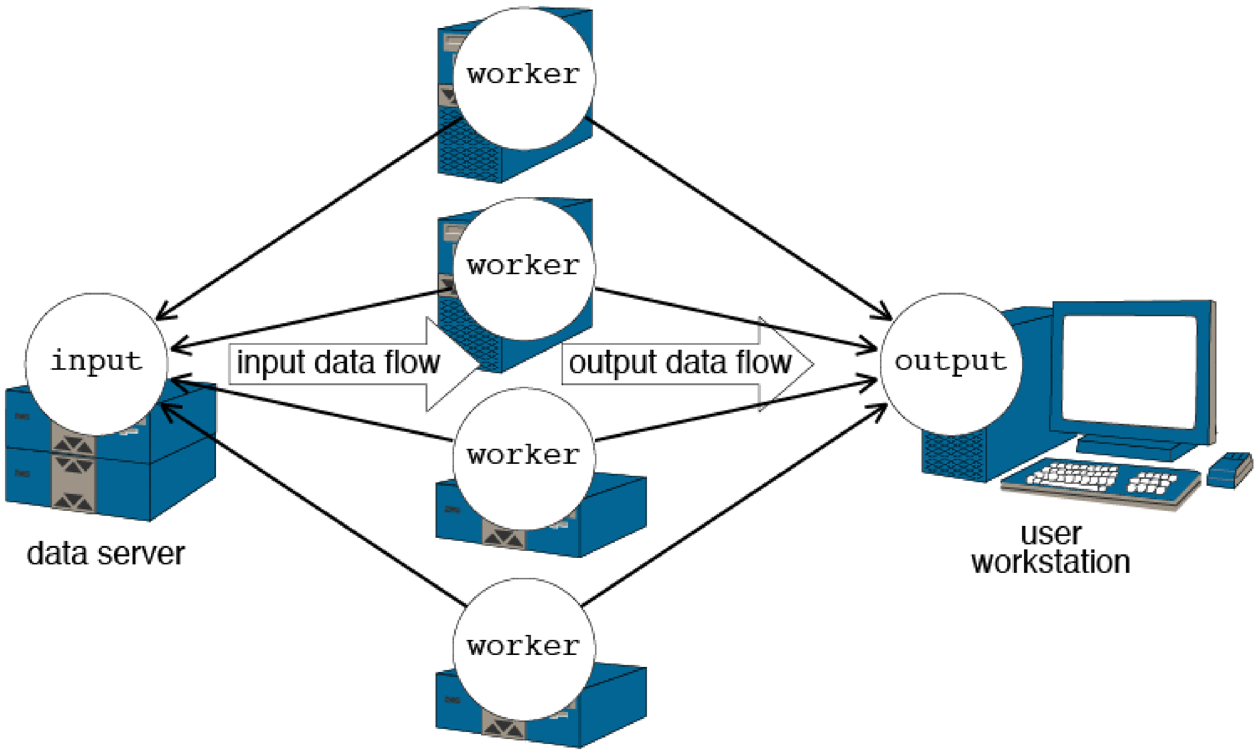


Fig. 1: A scenario using shared parallel objects

  - **Asynchronous invocation**: the invocation returns immediately after sending the request to the remote object. Asynchronous invocation is important to exploit the parallelism. However, as the caller does not wait the end of the execution of the called method, no computing result is available. This excludes asynchronous invocations from producing results. Results can be actively returned to the caller object using a callback to the caller. To do so the called object must have a reference to the caller object. This reference can be passed as an argument to the called method (see 2).

- **Object-side semantics**, the semantics that affect the order of the execution of methods in the called parallel object:
  - **A mutex call** is executed after completion of all calls previously arrived.
  - **A sequential call** is executed after completion of all sequential and mutex calls previously arrived.
  - **A concurrent call** can be executed concurrently (time sharing) with other concurrent or sequential calls, except if mutex calls are pending or executing. In the latter case the call is executed after completion of all mutex calls previously arrived.

In a nutshell, different object-side invocation semantics can be expressed in terms of atomicity and execution order. The mutex invocation semantics guarantees the global order and the atomicity of all method calls. The sequential invocation semantics guarantees only the execution order of sequential methods. Concurrent invocation semantics guarantees neither the order nor the atomicity.

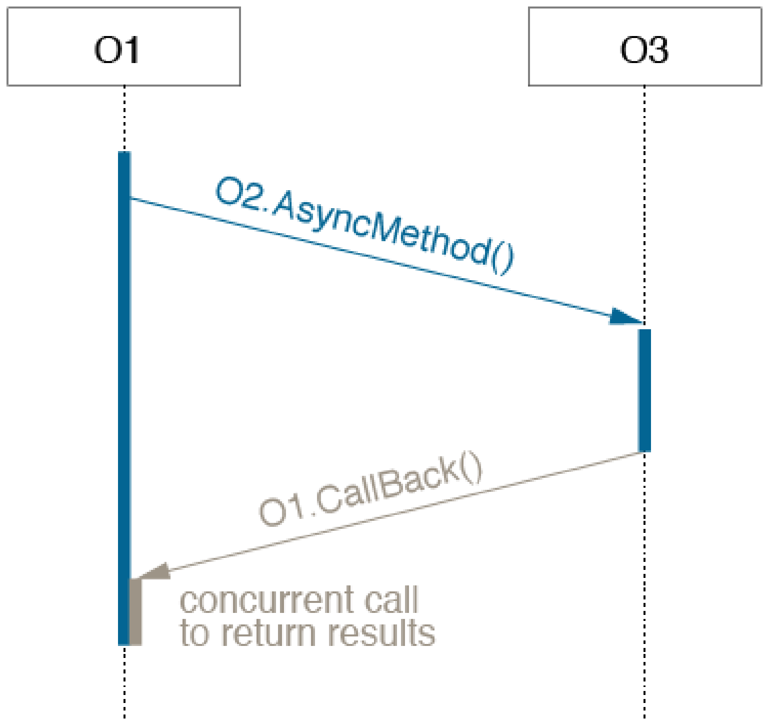


Fig. 2: Callback method returning values from an asynchronous call

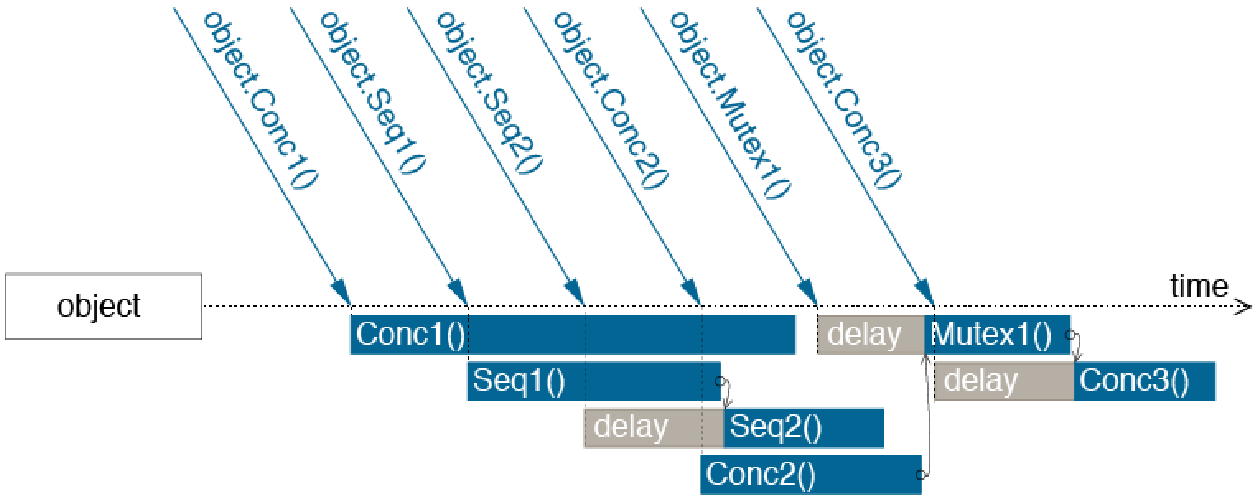


Fig. 3: Example of different invocation requests

3 illustrates different method invocation semantics. Sequential invocation `Seq1()` is served immediately, running concurrently with `Conc1()`. Although the sequential invocation `Seq2()` arrives before the concurrent invocation `Conc2()`, it is delayed due to the current execution of `Seq1()` (no order between concurrent and sequential invocations). When the mutex invocation `Mutex1()` arrives, it has to wait for other running methods to finish. During this waiting, it also blocks other invocation requests arriving afterward (`Conc3()`) until the mutex invocation request completes its execution (atomicity and barrier).

2.5 Parallel Object Allocation

The first step to allocate a new object is the selection of an adequate placeholder. The second step is the object creation itself. Similarly, when an object is no longer in use, it must be destroyed in order to release the resources it is occupying in its placeholder. The POP-C++ runtime system provides automatic placeholder selection, object allocation, and object destruction. Those automatic features result in a dynamic usage of computational resources and gives to the applications the ability to adapt to the changes in both the environment and the user behavior.

The creation of POP-Java parallel objects is driven by high-level requirements on the resources where the object should lie (see section *Requirement-driven parallel objects*). If the programmer specifies these requirements they are taken into consideration by the runtime system for the transparent object allocation. The allocation process consists of three phases: first, the system finds a suitable resource, where the object will lie; then the object code is transmitted and executed on that resource; and finally, the corresponding interface is created and connected to the object.

2.6 Requirement-driven parallel objects

Parallel processing is increasingly being done using distributed systems, with a strong tendency towards web and global computing. Efficiently extracting high performance from highly heterogeneous and dynamic distributed environments is a challenge today. POP-C++ and POP-Java were conceived under the belief that for such environments, high performance can only be obtained if the two following conditions are satisfied:

- The application should be able to adapt to the environment;
- The programming environment should somehow enable objects to describe their resource requirements.

The application adaptation to the environment can be fulfilled by multilevel parallelism, dynamic utilization of resources or adaptive task size partitioning. One solution is to dynamically create parallel objects on demand.

Resource requirements can be expressed by the quality of service that objects require from the environment. Most of the systems offering quality of service focus on low-level aspects, such as network bandwidth reservation or real-time scheduling. Both POP-C++ and POP-Java integrate the programmer requirements into parallel objects in the form of high-level resource descriptions. Each parallel object is associated with an object description that depicts the characteristics of the resources needed to execute the object. The resource requirements in object descriptions are expressed in terms of:

- Resource (host) name (low level description, mainly used to develop system services).
- The maximum computing power that the object needs (expressed in MFlops).
- The maximum amount of memory that the parallel object consumes.
- The expected communication bandwidth and latency.
- The preferred communication protocol.
- The preferred encoding protocol.

An object description can contain several items. Each item corresponds to a type of characteristics of the desired resource. The item is classified into two types: strict item and non-strict item. A strict item means that the designated requirement must be fully satisfied. If no satisfying resource is available, the allocation of parallel object fails. Non-strict items, on the other hand, give the system more freedom in selecting a resource. Resource that partially match the requirements are acceptable although a full qualification resource is preferable. For example, a certain object has a preferred performance 150MFlops although 100MFlops is acceptable (non-strict item), but it needs memory storage of at least 128MB (strict item).

The construction of object descriptions occurs during the parallel object creation. The programmer can provide an object description to each object constructor. The object descriptions can be parametrized by the arguments of the constructor. Object descriptions are used by the runtime system to select an appropriate resource for the object. Some examples of the syntax of object descriptions can be found in the section *Object description*.

It can occur that, due to some changes on the object data or some increase of the computation demand, an object description needs to be re-adjusted during the life time of the parallel object. If the new requirement exceeds some threshold, the adjustment could cause the object migration. The current implementations of POP-C++ and POP-Java do not support object migration yet.

CHAPTER 3

Developing POP-Java applications

The *POP model* is a suitable programming model for large heterogeneous distributed environments but it should also remain as close as possible to traditional object oriented programming. Parallel objects of the POP model generalize sequential objects, keep the properties of object oriented programming (data encapsulation, inheritance and polymorphism) and add new properties.

The POP-Java language is an extension of Java that implements the POP model. Its syntax remains as close as possible to standard Java so that Java programmer can easily learn it and existing Java libraries can be parallelized without much effort. Changing a sequential Java application into a distributed parallel application is rather straightforward. POP-Java is also very close to POP-C++ so that POP programmer can use both systems easily.

Parallel objects are created using parallel classes. Any object that instantiates a parallel class is a parallel object and can be executed remotely. To help the POP-C++ runtime to choose a remote machine to execute the remote object, programmers can add object description information to each constructor of the parallel object. In order to create parallel execution, POP-Java offers new semantics for method invocations. These new semantics are indicated thanks to new POP-Java keywords. This chapter describes the syntax of the POP-Java programming language and presents the main tools available in the POP-Java system.

3.1 Parallel objects

POP-Java parallel objects are a generalization of sequential objects. Unless the term sequential object is explicitly specified, a parallel object is simply referred as an object in the rest of this chapter.

3.1.1 Create a parallel class

Developing POP-Java application mainly consists of designing an implementing parallel classes. The declaration of a parallel class is the same as a standard Java class declaration, but it has to be annotated with the annotation `@POPClass`. The parallel class can extend another parallel class but not a sequential class.

Simple parallel class declaration

```
@POPClass
public class MyParallelClass {
    // Implementation
}
```

Parallel class declaration with an inheritance

```
@POPClass
public class MyParallelClass extends AnotherParallelClass {
    // Implementation
}
```

As Java allows only single inheritance, a parallel class can only inherit from **one** other parallel class. The Java language also imposes that the file including the parallel class has the same name than the parallel class.

Parallel classes are very similar to standard Java classes. As POP-Java has some different behavior, some restrictions applied to the parallel classes:

- All attributes in a parallel class must be protected or private
- The objects do not access any global variable
- A parallel class does not contain any static methods or non final static attributes

3.1.2 Creation and destruction

The object creation process consists of several steps: locating a resource satisfying the object description (resource discovery), transmitting and executing the object code, establishing the communication, transmitting the constructor arguments and finally invoking the corresponding object constructor. Failures on the object creation will raise an exception to the caller. *Exception handling* will describe the POP-Java exception mechanism.

As a parallel object can be accessible concurrently from multiple distributed locations (shared object), destroying a parallel object should be carried out only if there is no other reference to the object. POP-Java manages parallel objects' life time by an internal reference counter. A counter value of 0 will cause the object to be physically destroyed.

Syntactically, the creation of a parallel object is identical to the one in Java. A parallel object can be created by using the standard new operator of Java.

3.1.3 Parallel class methods

Like sequential classes, parallel classes contain methods and attributes. Method can be public or private but attribute must be either protected or private. For each public method, the programmer must define the invocation semantics. These semantics, described in *Invocations semantics*, are specified by an annotation.

- **Interface side**: These semantics affect the caller side. * `sync`: Synchronous invocation. * `async`: Asynchronous invocation.
- **Object side**: These semantics affect the order of incoming method calls on the object. * `seq`: Sequential invocation * `conc`: Concurrent invocation * `mutex`: Mutual exclusive invocation

The combination of the interface and object-side semantics defines the overall semantics of a method. There are 6 possible combinations of the interface and object-side semantics, resulting in 6 annotations:

```
@POPSyncConc
@POPSyncSeq
@POPSyncMutex
@POPAsyncConc
@POPAsyncSeq
@POPAsyncMutex
```

The following code example shows a synchronous concurrent method that returns an int value:

```java
@POPSyncConc
public int myMethod(){
   return myIntValue;
}
```

A method declared as asynchronous must have its return type set to void. Otherwise, the compiler will raise an error.

3.1.4 Object description

Object descriptions are used to describe the resource requirements for the execution of an object. Object descriptions are declared along with parallel object constructor declarations. The object description can be declared in a static way as an annotation of the constructor, or in a dynamic way as an annotation on the parameters of the constructor. First an example of a static annotation:

```
@POPObjectDescription(url="localhost")
public MyObject(){
}
```

and now a dynamic example:

```
public MyObject(@POPConfig(Type.URL) String host){
}
```

Currently only the url annotation is implemented, allowing to specify the URL/IP of the machine on which the POP-Object is executed. If the annotation is not set, POP-Java will use the POP-C++ jobmanager to find a suitable machine.

3.1.5 Data marshaling and IPOPBase

When calling a remote method, the arguments must be transferred to the object being called (the same happens for the return value and the exception). In order to operate with different memory spaces and different architectures, the data is marshaled into a standard format prior to be sent to remote objects. All data is serialized (marshaled) at the caller side an deserialized (unmarshaled) at the remote side.

With POP-Java all primitive types, primitive type arrays and parallel classes can be passed without any trouble to another parallel object. This mechanism is transparent for the programmer.

If the programmer wants to pass a special object to or between parallel classes, this object must implement the IPOPBase interface from the POP-Java library. This library is located in the installation directory (`POPJAVA_LOCATION/JarFile/popjava.jar`). By implementing this interface, the programmer will have to override the two following methods:

```
@Override
public boolean deserialize(POPBuffer buffer) {
   return true;
}

@Override
public boolean serialize(POPBuffer buffer) {
   return true;
}
```

These methods will be called by the POP-Java system when an argument of this type needs to be serialized or deserialized. As the object will be reconstructed on the other side and after the values will be set to it by the deserialize method, any class implementing the `IPOPBase` interface must have a default constructor.

The code below shows a full example of a class implementing the IPOPBase interface:

```java
import ch.icosys.popjava.core.buffer.Buffer;
import ch.icosys.popjava.core.dataswaper.IPOPBase;

public class MyComplexType implements IPOPBase {
   private int theInt;
   private double theDouble;
   private int[] someInt;

   public MyComplexType(){}

   public MyComplexType(int i, double d, int[] ia){
      theInt = i;
      theDouble = d;
      someInt = ia;
   }

   @Override
   public boolean deserialize(POPBuffer buffer) {
      theInt = buffer.getInt();
      theDouble = buffer.getDouble();
      int size = buffer.getInt();
      someInt = buffer.getIntArray(size);
      return true;
   }

   @Override
   public boolean serialize(POPBuffer buffer) {
      buffer.putInt(is);
      buffer.putDouble(ds);
      buffer.putIntArray(ias);
      return true;
   }
}
```

3.2 POP-Java behavior

This section aims to explain the difference between the standard Java behavior and the POP-Java behavior.

As in standard Java, the primitive types will not be affected by any manipulation inside a method as they are passed by value and not by reference. Objects passed as arguments tho methods will only be affected if the method semantic is “Synchronous”. In fact, POP-Java serializes the method arguments to pass them on the object-side. Once the method work is done, the arguments are serialized once again to be sent back to the interface-side. If the method semantic is “Synchronous”, the interface-side will deserialize the arguments and replace the local ones by the deserialized arguments. If the method semantic is "Asynchronous", the interface-side will not wait for any answer from the object-side. It's important to understand this small difference when developing POP-Java application.

3.3 Exception handling

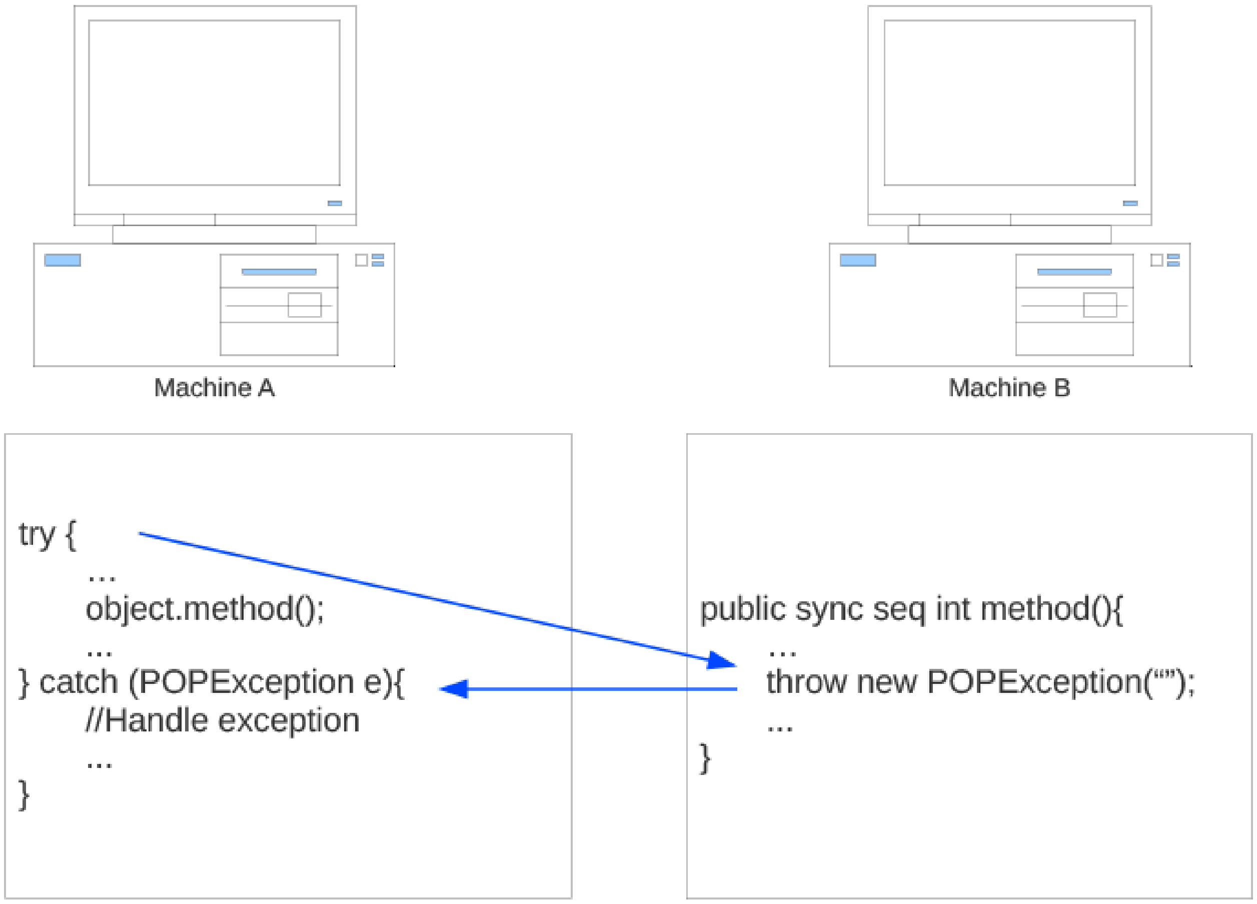
Errors can be efficiently handled using exceptions. Instead of handling each error separately based on an error code returned by a function call, exceptions allow the programmer to filter and centrally manage errors through several calling stacks. When an error is detected inside a certain method call, the program can throw an exception that will be caught somewhere else.

The implementation of exception in non-distributed applications, where all components run within the same memory address space is fairly simple. The compiler just need to pass a pointer to the exception from the place where it is thrown to the place where it is caught. However, in distributed environments where each component is executed in a separated memory address space (and the data is represented differently due to heterogeneity), the propagation of exception back to a remote component is complex.

POP-Java supports transparent exception propagation. Exceptions thrown in a parallel object will be automatically propagated back to the remote caller (1). The current POP-Java version allows the following types of exceptions:

- `Exception`
- `POPException`

The invocation semantics of POP-Java affect the propagation of exceptions. For the moment, only synchronous methods can propagate the exception. Asynchronous methods will not propagate any exception to the caller. POP-Java current behavior is to abort the application execution when such an exception occurs.



```
try {
        ...
        object.method();
        ...
} catch (POPException e){
        //Handle exception
        ...
}
```

```
public sync seq int method(){
        ...
        throw new POPException("");
        ...
}
```

Fig. 1: Exception handling example

CHAPTER 4

Compile and run a POP-Java application

This chapter explains the POP-Java compilation process, the POP-Java application launching process and the tools related to those processes. The structure of this chapter is as follows: The first section explains the compilation process. The second describes the application launching tools. The third one aims to help the programmer to understand the importance of the object map and the object map generator in the POP-Java application launching process. Finally, a full example is explained to pass through the whole process.

4.1 POP-Java compilation

POP-Java uses the standard Java compiler and can easily be integrated into an existing compilation process. As POP-Java java files use features from the POP-Java library, the POP-Java jar file needs to be included in the classpath during the compilation process. An example of how to compile POP-Java source-code using ant is shown below:

```
<property environment="env"/>

<target name="build" description="compile the source " >
        <javac srcdir="${source.folder}"
                destdir="${class.folder}"
                classpath="${env.POPJAVA_LOCATION}/JarFile/popjava.jar"
        />
</target>
```

4.2 The POP-Java application launcher

To help POP-Java programmer, POP-Java provides an application launcher that simplifies the launch of a POP-Java application. This application launcher is named `popjrun` and is used with the following syntax:

```
popjrun <options> [<objectmap>] <MainClass> <arguments>
```

Here is an explanation of the arguments to provide to the POP-Java application launcher:

- `options`: in the current version there is only one option `-c` or `--classpath` that allow the programmer to add some class path for the execution of the POP-Java application. The different class paths must be separated with a semicolon.
- `objectmap`: this informations is not mandatory. If it's provided, the object map informs the runtime system about the location of the different compiled parallel classes of the application. If it's not provided, the default object map (located at: `{POPJAVA_LOCATION}/etc/defaultobjectmap.xml`) will be used. More information give in *The POP-Java object map and object map generator*.
- `MainClass`: this is a main class of the POP-Java application.
- `arguments`: these are the arguments of the program.

4.3 The POP-Java object map and object map generator

The object map is an XML file that informs the POP-C++ runtime about the location of the different compiled parallel classes of the application. This file can be given to the "popjrun" tool. If the programmer does not specify this file, the default object map located at `{POPJAVA_LOCATION}/etc/` will be used.

The object map can be generated with the POP-Java application launcher. By using the option `-l` or `--listlong` and giving the class files or the JAR file, the object map will be printed to the standard output. The easiest way to save this file is to redirect the output into the desired file.

Here are the commands used for our example:

Compiled classes

```
popjrun --listlong Parclass1.class:Parclass2.class > objectmap.xml
```

JAR file

```
popjrun --listlong parclasses.jar > objectmap.xml
```

An example of a generated objectmap can be found here: *The POP-Java object map and object map generator*. The objectmap contains the path to the compiled classfile for every POP-Java class passed to the popjrun command. The path can either be a path to a folder containing the class file, or a jar file containing the class file. The path can either be a local path, or a url accessible by http. Keep in mind that all paths indicated need to be accessible by every machine that will create a POP-Java object.

4.4 Full example

This section shows how to write, compile and launch a POP-Java application by using a simple example. The POP-Java application used in this example includes only one parallel class. All sources of this example can be found in the directory `examples/integer` from the POP-Java distribution.

4.4.1 Programming

When we start to develop a POP-Java application the main part is the parallel classes. The following code snippet shows the parallel class implementation:

```java
import ch.icosys.popjava.core.annotation.*;

@POPClass
public class Integer {
    private int value;

    @POPObjectDescription(url="localhost")
    public Integer() {
        value = 0;
    }

    @POPSyncConc
    public int get() {
        return value;
    }

    @POPAsyncSeq
    public void set(int val) {
        value = val;
    }

    @POPAsyncMutex
    public void add(Integer i) {
        value += i.get();
    }
}
```

As we can see this class uses special POP-Java keywords. In the line 1, the parclass keyword specifies that this class is a parallel class. The constructor declaration includes an object description (line 4). The method declarations includes the invocation semantics (line 8, 12 and 16). The method `add` (line 16) receive another parallel object as a parameter and it's transparent for the programmer.

Once the parallel class is implemented, we can write a main class that use this parallel class. The following code snippet shows the code of the main class:

```
1  import ch.icosys.popjava.core.annotation.*;
2
3  @POPClass(isDistributable = false)
4  public class TestInteger {
5      public static void main(String[] args){
6          Integer i1 = new Integer();
7          Integer i2 = new Integer();
8          i1.set(23);
9          i2.set(25);
10         System.out.println("i1=" + i1.get());
11         System.out.println("i2=" + i2.get());
12         i1.add(i2);
13         int sum = i1.get();
14         System.out.println("i1+i2 = "+sum);
15         if(sum==48)
16             System.out.println("Test Integer Successful");
17         else
18             System.out.println("Test Integer failed");
19     }
20 }
```

The code of the main class is pure Java code. The instantiation (lines 3-4) and the method calls (lines 5-9) are transparent for the programmer.

4.4.2 Compiling

To manually compile the source files, use the following command:

Compiling as .class files

:: javac -cp $POPJAVA_LOCATION/JarFile/popjava.jar Integer.java TestInteger.java

4.4.3 Create the object map

Before running the example application, the programmer needs to generate the object map. The object map will be given to the POP-Java launcher which will inform the POP-C++ runtime system where to find the compiled files. The specified path needs to be accessible on every machine where an object of that type is initialized. The POP-Java launcher has a specific option to generate this file from the compiled files (`.class`) or the JAR file (`.jar`). Here is the command used for our example:

```
popjrun --listlong Integer.class > objmap.xml
```

The command will generate the XML file and print it on the standard output. To save this file, we redirect the output in a file named objmap.xml. This file contains the following XML code (the path specified in the element CodeFile will be different on your computer):

```
<CodeInfoList>
  <CodeInfo>
    <ObjectName>Integer</ObjectName>
    <CodeFile Type="popjava">
      /home/clementval/pop/popjava-1.0/example/integer/</CodeFile>
    <PlatForm>*-*</PlatForm>
  </CodeInfo>
</CodeInfoList>
```

4.4.4 Running

Once the POP-Java application is compiled and the object map is generated, the application can be run. A POP-Java application is a pure Java application at the end and could be run with the standard java program. In order to make this running easier for the programmer, POP-Java includes an application launcher. Here are the commands to use to run the POP-Java application example. At the end an example is given on how run the POP-Java application directly through Java.

POP-Java application compiled as .class files

```
popjrun objectmap.xml TestInteger
```

POP-Java application compiled as .jar file

```
popjrun -c myjar.jar objectmap.xml TestInteger
```

POP-Java application run directly through java

```
java -javaagent:$POPJAVA_LOCATION/JarFile/popjava.jar -cp myjar.jar␣
↪TestInteger -codeconf=objectmap.xml
```

Application output

Here is what we should have as the application output:

```
i1=23
i2=25
```

(continues on next page)

(continued from previous page)

```
i1+i2=48
Test Integer Successful
```

If the are any problems with the compilation or the launching of the application, please refer to the chapter *Troubleshooting*.

4.4.5 Misc

If you are running a POP-Java application on a computer with multiple network interfaces, make sure you specify the network interface to use. To specify the name of the network interface, set the `POPJ_IFACE` environment variable. If the specified name is not found, POP-Java will fall back to the same behaviour as if no network interface was specified as default.

CHAPTER 5

Developing POP-Java and POP-C++ mixed applications

5.1 POP-Java and POP-C++ interoperability

POP-Java can use POP-C++ parallel classes and POP-C++ can also use POP-Java parallel classes. This chapter will explain everything the programmer needs to know to develop mixed POP applications.

5.2 Restrictions

As Java and C++ are different languages, there are some restrictions. In this section, all the restrictions or programming tips will be given.

5.3 Java primitives

As Java primitives are always passed by value, the is no way to modify a Java primitive in a POP-C++ object. In pure POP-C++ the programmer can deal with the passing by reference but not in POP-Java.

5.3.1 Parameters passing

Some parameters cannot be passed from a POP-Java application to a POP-C++ parallel object and vice versa. The list below explain the restrictions on certain primitive types. The Java primitive types are taken as the basis.

- `byte`: This type does not exist in C++ so it's not possible to pass a byte.
- `long`: The Java long is coded on 8 bytes as it's coded on 4 bytes with C++. Some unexpected behavior can occur.
- `char[]`: The char array cannot be used in the current version of POP-Java with POP-C++ parallel classes.

5.3.2 Dealing with array

Passing arrays from POP-Java to POP-C++ is a bit tricky. As POP-Java and POP-C++ do not behave the same with arrays, the programmer must be aware of the way to pass the array. Here is an example of a method with an array as parameter.

The method declaration in POP-C++ In POP-C++, the programmer must give the array size to the compiler.

```
sync seq void changeIntArray(int n, [in, out, size=n] int *i);
```

Method declaration in POP-Java As POP-C++ will need the size of the array, POP-Java must declare this size as well.

```
@POPSyncSeq
public void changeIntArray(int n, int[] i){}
```

Method call from POP-Java In the POP-Java application, the programmer needs to give the array size in the method call.

```
p.changeIntArray(iarray.length, iarray);
```

5.4 POP-Java application using POP-C++ parallel objects

This section will teach the programmer how to develop a POP-Java application with a POP-C++ parallel class. The same example of the parallel class Integer will be used. For more details about the POP-C++ programming please have a look to "Parallel Object Programming C++ - User and Installation Manual" [Grid and Ubiquitous Computing Group, EIA-FR10]. In the following example, the main class used is the same as the one shown in the *previous chapter*. All the sources can be found in the directory `example/mixed1` of the POP-Java distribution.

5.4.1 Develop the POP-C++ parallel class

First, the programmer needs to write the parallel class in POP-C++ as it should be for a POP-C++ application. The code snippet below shows the header file of the parclass Integer:

```
parclass Integer
{
   classuid(1000);
public:
   Integer();
   ~Integer();

   mutex void Add(Integer &other);
   conc int Get();
   seq async void Set(int val);

private:
   int data;
};
```

There are two rules to follow when the programmer develop a POP-C++ parallel class for POP-Java usage.

- The parclass must declare a classuid.
- The methods must be declared in alphabetics order.

The next code snippet shows the implementation of the parallel class `Integer`. There is no important information in this file for the POP-Java usage.

```
#include <stdio.h>
#include "integer.ph"
#include <unistd.h>

Integer::Integer() {
   printf("Create remote object Integer on %s\n",
          (const char *)POPSystem::GetHost());
}

Integer::~Integer() {
   printf("Destroying Integer object...\n");
}

void Integer::Set(int val) {
   data=val;
}

int Integer::Get() {
```

(continues on next page)

(continued from previous page)

```
   return data;
}

void Integer::Add(Integer &other) {
   data += other.Get();
}
@pack(Integer);
```

Compilation of the parallel class Once the parclass implementation is finished, it can be compiled with the POP-C++ compiler. The following command will create an object executable of our parclass Integer.

```
popcc -object -o integer.obj integer.cc integer.ph
```

5.4.2 Create the partial POP-Java parallel class

To be used in a POP-Java application, a POP-C++ parallel class must have its partial implementation in POP-Java language. A partial implementation means that all the methods must be declared but does not need to be implemented.

The next code snippet shows the partial implementation of the parallel class `Integer`. All the methods are just declared. This partial implementation is a translation of the POP-C++ source code to POP-Java source code.

```
@POPClass
public class Integer {
   private int value;

   public Integer() {
   }

   @POPSyncMutex
   public void add(Integer i) {
   }

   @POPSyncConc
   public int get() {
      return 0;
   }

   @POPAsyncSeq
   public void set(int val) {
   }
}
```

Note: In the future version of POP-C++ and POP-Java, the partial implementation would be generated by the compiler. For the moment, the programmer will need to do it by hand.

5.4.3 Special compilation

To compile the partial POP-Java parallel class, the compiler needs some additional information. The POP-Java compiler has an option to generate an additional information XML file. To generate this file use the following command line:

```
popjc -x Integer.pjava
```

This command will generate a file (`additional-infos.xml`) in the current directory. This file is incomplete. The programmer will need to edit it with the information of the POP-C++ parallel class. The following snippet shows the file generated by the POP-Java compiler:

```
<popjparser-infos>
   <popc-parclass file="Integer.pjava" name="" classuid=""
                  hasDestructor="true"/>
</popjparser-infos>
```

The two empty attributes `name` and `classuid` must be completed with the value of the POP-C++ parallel class. An example of how the complete file must look like is given below:

```
<popjparser-infos>
   <popc-parclass file="Integer.pjava" name="Integer" classuid="1000"
                  hasDestructor="true"/>
</popjparser-infos>
```

All the information to compile the POP-Java application is now known. Here is the command to compile it:

Compilation as .class files

```
popjc -p additional-infos.xml Integer.pjava TestInteger.pjava
```

Compilation as .jar file

```
popjc -j myjar.jar -p additional-infos.xml Integer.pjava TestInteger.
↪pjava
```

5.4.4 Generate the object map

An object map is also needed for a POP-Java application using POP-C++ parallel classes. The programmer can generate this object map with the POP-Java application launcher and the option `--listlong`. This option also accepts the POP-C++ executable files. Here is the command used for the example application:

```
popjrun --listlong integer.obj > objmap.xml
```

Generated objmap.xml file (path and architecture can differ from the ones shown here):

```
<CodeInfoList>
   <CodeInfo>
      <ObjectName>Integer</ObjectName>
      <CodeFile>/home/clementval/pop/popjava-1.0/example/mixed/
      integer.obj</CodeFile>
      <PlatForm>i686-pc-Linux</PlatForm>
   </CodeInfo>
</CodeInfoList>
```

5.4.5 Running the application

To run the mixed application, the programmer needs to use the POP-Java application launcher. As the application main class is written in POP-Java, only this tool can run this application. Here is the command used to run the application:

```
popjrun objmap.xml TestInteger
```

The output of the example application should be like the following:

```
i1=23
i2=25
i1+i2=48
Test Integer Successful
```

If any problems occurred with the compilation or the launching of the application, please see the chapter *Troubleshooting*.

5.5 POP-C++ application using POP-Java parallel objects

A POP-C++ application can also use POP-Java parallel classes. The following chapter shows how to develop, compile and run a POP-C++ using POP-Java parallel objects.

5.5.1 Developing and compiling the POP-Java parallel class

The POP-Java parallel class will be the same as the one shown in the *previous chapter*. The compilation will be a little bit different. As for a POP-Java application using a POP-C++ parclass, the POP-Java will need some additional informations during the compilation process. These additional information must be given in a XML file. The POP-Java compiler can generate a canvas of this file with the option "-x". Here is the command we used:

```
popjc -x Integer.pjava
```

The generated file will be similar to the one shown in the *Special compilation section*. This time the attribute `name` must stay empty as we want to keep the real name of the POP-Java parallel class. The completed file should look like in the following snippet:

```
<popjparser-infos>
   <popc-parclass file="Integer.pjava" name="" classuid="1000"
                  hasDestructor="true"/>
</popjparser-infos>
```

This file can be given to the compiler to compile the parallel class with the following command:

```
popjc -p additional-infos.xml Integer.pjava
```

5.5.2 The POP-C++ partial implementation

As for the POP-Java application using POP-C++ parallel objects, the POP-C++ application will need a partial implementation of the parallel class in POP-C++. The header file will stay the same as the one shown *previously*. The code snippet below shows the partial implementation of the POP-C++ parallel class. Once again, the methods are declared but not implemented.

```
#include <stdio.h>
#include "integer.ph"
#include <unistd.h>

Integer::Integer() {
   printf("Create remote object Integer on %s\n",
          (const char *)POPSystem::GetHost());
}

Integer::~Integer() {
}

void Integer::Set(int val) {
}

```

(continues on next page)

(continued from previous page)

```
int Integer::Get() {
   return 0;
}

void Integer::Add(Integer &other) {
}
@pack(Integer);
```

5.5.3 The POP-C++ main

To be able to run the application, a `main` function must be written. An example of such a function is given below:

```
#include "integer.ph"
#include <iostream>
using namespace std;
int main(int argc, char **argv)
{
   try{
      // Create 2 Integer objects
      Integer o1;
      Integer o2;
      o1.Set(1); o2.Set(2);
      cout << endl << "o1="<< o1.Get() << "; o2=" << o2.Get() << endl;
      cout<<"Add o2 to o1"<<endl;
      o1.Add(o2);
      cout << "o1=o1+o2; o1=" << o1.Get() << endl << endl;
   } catch (POPException *e) {
      cout << "Exception occurs in application :" << endl;
      e->Print();
      delete e;
      return -1;
   } // catch
   return 0;
}
```

The main is very similar to the one used in POP-Java but this time it is written in POP-C++.

5.5.4 Object map

As the current version of POP-C++ is not able to generate the object map for a POP-Java parallel class, the programmer needs to edit the object map manually.

The code below is the canvas of the line to add in a POP-C++ object map for a POP-Java parallel class.

```
POPCObjectName *-* /usr/bin/java -cp POPJAVA_LOCATION
popjava.broker.Broker -codelocation=CODE_LOCATION
-actualobject=POPJAVAObjectName
```

Here is the line for the example (the path will be different on your computer):

```
Integer *-* /usr/bin/java -cp /home/clementval/popj
popjava.broker.Broker
-codelocation=/home/clementval/pop/popjava-1.0/example/mixed2
-actualobject=Integer
```

5.5.5 Compile and run the POP-C++ application

The POP-Java parallel class is compiled and the object map is complete. The main and the partial implementation of the parallel class in POP-C++ must be compiled. The following command will compile our application:

```
popcc -o main integer.ph integer.cc main.cc
popcc -object -o integer.obj integer.cc integer.ph main.cc
```

Everything is compiled and we can run the application with the "popcrun" tool:

```
popcrun obj.map ./main
```

The output of the application should look like this:

```
popcrun obj.map ./main

o1=1; o2=2
Add o2 to o1
o1=o1+o2; o1=3
```

CHAPTER 6

POP-Java plugin

The POP-Java system can be augmented by its users. If the programmer feels the need of a new network protocol or a new encoding protocol, he can create a POP-Java plugin and add it to the system easily. This chapter aims to present the combox and the buffer plugin systems.

6.1 Combox plugin

The Combox is the component responsible for the network communication between an application and a parallel object or between two parallel objects. In the current version of POP-Java, only the protocol socket is implemented. If the programmer needs another protocol, he can create his own Combox.

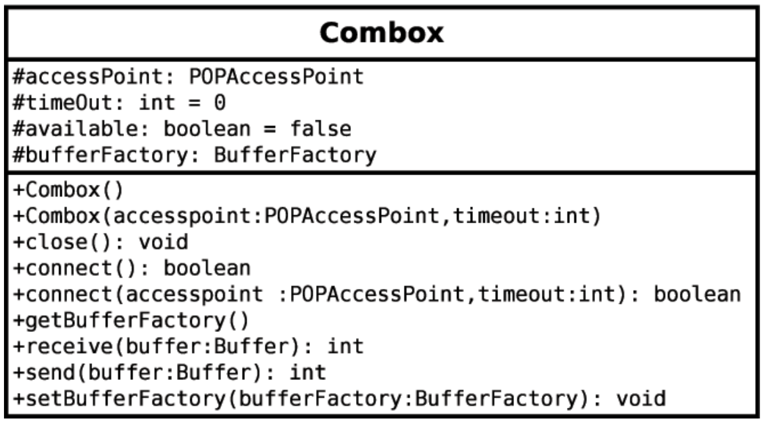
To create a new protocol for POP-Java, the programmer needs to create three different classes : a combox, a combox server and a combox factory.

The combox must inherit from the super class ComboxPlugin located in the package `popjava.combox` in the POP-Java library. The 1 shows the `ComboxPlugin` class signature.

The combox server must inherit from the super class ComboxServer located in the package popjava.combox in the POP-Java library. The 2 shows the ComboxServer class signature.

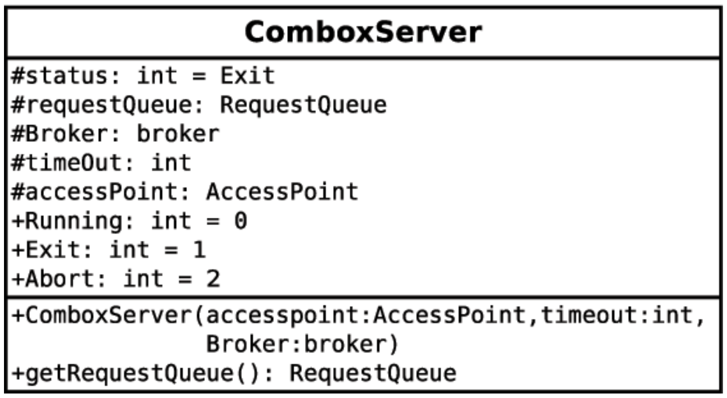
The combox factory must inherit from the super class ComboxFactory located in the package `popjava.combox` in the POP-Java library. The 3 shows the ComboxFactory class signature.

Once all the classes are implemented, the programmer needs to compile them as standard Java code and create a JAR file. This JAR file can be added in the system by editing the file `pop_combox.`



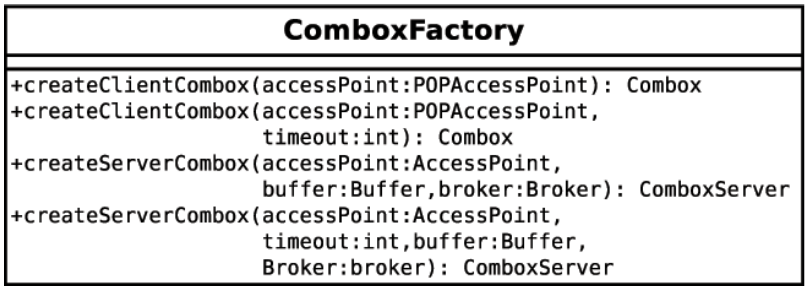
```
Combox
#accessPoint: POPAccessPoint
#timeOut: int = 0
#available: boolean = false
#bufferFactory: BufferFactory
+Combox()
+Combox(accesspoint:POPAccessPoint,timeout:int)
+close(): void
+connect(): boolean
+connect(accesspoint :POPAccessPoint,timeout:int): boolean
+getBufferFactory()
+receive(buffer:Buffer): int
+send(buffer:Buffer): int
+setBufferFactory(bufferFactory:BufferFactory): void
```

Fig. 1: ComboxPlugin class signature



```
ComboxServer
#status: int = Exit
#requestQueue: RequestQueue
#Broker: broker
#timeOut: int
#accessPoint: AccessPoint
+Running: int = 0
+Exit: int = 1
+Abort: int = 2
+ComboxServer(accesspoint:AccessPoint,timeout:int,
              Broker:broker)
+getRequestQueue(): RequestQueue
```

Fig. 2: ComboxServer class signature



```
ComboxFactory
+createClientCombox(accessPoint:POPAccessPoint): Combox
+createClientCombox(accessPoint:POPAccessPoint,
                    timeout:int): Combox
+createServerCombox(accessPoint:AccessPoint,
                    buffer:Buffer,broker:Broker): ComboxServer
+createServerCombox(accessPoint:AccessPoint,
                    timeout:int,buffer:Buffer,
                    Broker:broker): ComboxServer
```

Fig. 3: ComboxFactory signature

`xml` located in the directory `POPJAVA_LOCATION/plugin`. The XML code below is the current XML file with the socket protocol.

```
<ComboxFactoryList>
  <Package JarFile="popjava.combox.jar">
    <ComboxFactory>popjava.combox.ComboxSocketFactory</ComboxFactory>
  </Package>
</ComboxFactoryList>
```

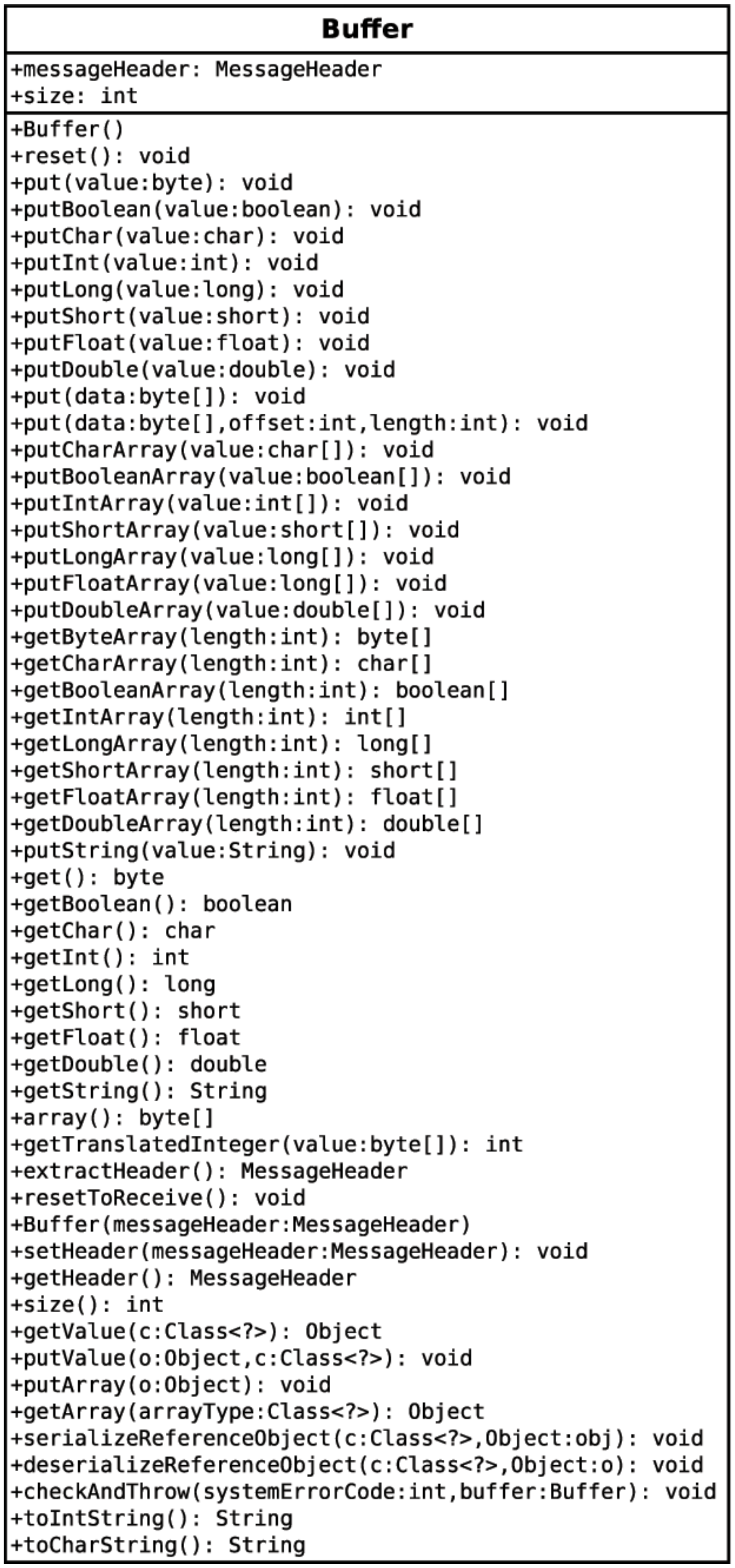
6.2 Buffer plugin

The buffer is the component in charge of the data encoding. In the current implementation of POP-Java, two buffers are available. One is using the RAW encoding and the other is using the XDR encoding. If the programmer needs a special encoding protocol, he can also create his own and add it to the POP-Java system as a plugin.

To implement a new encoding protocol, the programmer needs to create two classes. A buffer and a buffer factory.

The buffer must inherit from the class BufferPlugin located in the package `popjava.buffer` in the POP-Java library. The 4 shows the `BufferPlugin` class signature.

The buffer factory must inherit from the super class `BufferFactory` located in the package `popjava.buffer` in the POP-Java library. The 5 shows the `BufferFactory` class signature.



```
Buffer
+messageHeader: MessageHeader
+size: int
+Buffer()
+reset(): void
+put(value:byte): void
+putBoolean(value:boolean): void
+putChar(value:char): void
+putInt(value:int): void
+putLong(value:long): void
+putShort(value:short): void
+putFloat(value:float): void
+putDouble(value:double): void
+put(data:byte[]): void
+put(data:byte[],offset:int,length:int): void
+putCharArray(value:char[]): void
+putBooleanArray(value:boolean[]): void
+putIntArray(value:int[]): void
+putShortArray(value:short[]): void
+putLongArray(value:long[]): void
+putFloatArray(value:long[]): void
+putDoubleArray(value:double[]): void
+getByteArray(length:int): byte[]
+getCharArray(length:int): char[]
+getBooleanArray(length:int): boolean[]
+getIntArray(length:int): int[]
+getLongArray(length:int): long[]
+getShortArray(length:int): short[]
+getFloatArray(length:int): float[]
+getDoubleArray(length:int): double[]
+putString(value:String): void
+get(): byte
+getBoolean(): boolean
+getChar(): char
+getInt(): int
+getLong(): long
+getShort(): short
+getFloat(): float
+getDouble(): double
+getString(): String
+array(): byte[]
+getTranslatedInteger(value:byte[]): int
+extractHeader(): MessageHeader
+resetToReceive(): void
+Buffer(messageHeader:MessageHeader)
+setHeader(messageHeader:MessageHeader): void
+getHeader(): MessageHeader
+size(): int
+getValue(c:Class<?>): Object
+putValue(o:Object,c:Class<?>): void
+putArray(o:Object): void
+getArray(arrayType:Class<?>): Object
+serializeReferenceObject(c:Class<?>,Object:obj): void
+deserializeReferenceObject(c:Class<?>,Object:o): void
+checkAndThrow(systemErrorCode:int,buffer:Buffer): void
+toIntString(): String
+toCharString(): String
```

Fig. 4: BufferPlugin class signature

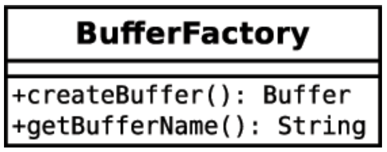


Fig. 5: BufferFactory class signature

CHAPTER 7

Installation

To use POP-Java and POP-C++ on a computer we need to install them. This chapter helps the programmer to perform the correct installation of the POP system on a computer.

7.1 POP-C++ installation

In order to use POP-Java we need to install the latest version of POP-C++. This section will help us to get through th installation process and make sure the installation is fine for a POP-Java usage.

7.1.1 Requirements

In order to install POP-C++ we need to install additional software. The following packages are required before compiling:

- a C++ compiler (g++ or equivalent)
- zlib-devel (package name depends on distribution)
- GNU Bison (optional)
- Globus Toolkit (optional)

7.1.2 Before installing

Before installation we should make the following configuration choices. In case of doubt the default values can be used.

- The compilation directory that should hold roughly 50MB. This directory will contain the distribution tree and the source files of POP-C++. It may be erased after installation.
- The installation directory that will hold less than 40M. It will contain the compiled files for POP-C++, include and configuration files. This directory is necessary in every computer executing POP-C++ program (default location `/usr/local/popc`).
- A temporary directory will be asked in the installation process. This directory will be used by POP-C++ to hold file during the application execution (default `/tmp`).
- Resource topology. The administrator must choose what computer form the grid.

For more information about the POP-C++ installation and configuration process, please see "Parallel Object Programming C++ - User and Installation Manual" [Grid and Ubiquitous Computing Group, EIA-FR10].

7.1.3 Installation process

This section will guide us through the POP-C++ installation process. In the POP distribution we find a directory including POP-C++. First, we need to configure the installation. If we use the configure script without any option, POP-C++ will be installed in the default directory (`/usr/local/popc`). We can also specify the directory by using the option –prefix.

Default directory

```
./configure
```

Specific directory

```
./configure --prefix=/home/user/popc
```

Once the configuration script is done, we will need to compile the source of POP-C++ for our architecture. For this, we just need to run the make command in the root directory of POP-C++.

```
make
```

Finally, to install POP-C++, we need to run the install target of the make file. This script will guide us through the installation. To be sure that our installation will fit the requirements of POP-Java, please follow the instructions below.

Answer "y" to the first question. We need to configure POP-C++ services.

```
make install
...
DO YOU WANT TO CONFIGURE POP-C++ SERVICES? (y/n)
y
```

We need to make a special installation so answer "n" to the second question:

```
...
DO YOU WANT TO MAKE A SIMPLE INSTALLATION ? (y/n):
n
```

The answers to the questions below are up to our configuration but if we don't know our configuration just pass every question.

```
=====================================================
GENERATING SERVICE MAPS...
CONFIGURING POP-C++ SERVICES ON YOUR LOCAL MACHINE...
Enter the full qualified master host name (POPC gateway):

Enter the child node:

Enter number of processors available (default:1):

Enter the maximum number of POP-C++ jobs that can run concurrently
(default: 100):

Enter the available RAM for job execution in MB (default 1024) :

Which local user you want to use for running POP-C++ jobs?

CONFIGURING THE RUNTIME ENVIRONMENT
Enter the script to submit jobs to the local system:

Communication pattern:

SETTING UP RUNTIME ENVIRONMENT VARIABLES
Enter variable name:
```

We need the startup script to use the global runtime service with POP-Java so answer "y" to the question "Do you want to generate the POP-C++ startup scripts?".

```
=====================================================
CONFIGURATION POP-C++ SERVICES COMPLETED!
=====================================================
Do you want to generate the POP-C++ startup scripts? (y/n)
y
```

Depending on our configuration, we can modify the default values of the startup script or just keep them. One important thing is to copy the environment variables on the .bashrc or equivalent file.

```
=====================================================
CONFIGURING STARTUP SCRIPT FOR YOUR LOCAL MACHINE...
Enter the service port[2711]:

Enter the domain name:

Enter the temporary directory for intermediate results:

=====================================================
CONFIGURATION DONE!
=====================================================

IMPORTANT : Do not forget to add these lines to your .bashrc
file or equivalent :
---------
    POPC_LOCATION=/home/clementval/popc
    PATH=$PATH:$POPC_LOCATION/bin:$POPC_LOCATION/sbin

Press <Return> to continue
```

The POP-C++ installation is done. We can now use POP-C++ and also install POP-Java.

7.1.4 System startup

Before executing any POP-C++ application, the runtime system (Job manager and resource discovery) must be started. There is a script provided for that purpose, so every node must run the following command:

```
POPC_LOCATION/sbin/SXXpopc start
```

SXXpopc is a standard Unix deamon control script, with the traditional start, stop and restart options.

7.2 POP-Java installation

This section will guide us through the POP-Java installation process.

7.2.1 Requirements

In order to install POP-Java, some packages are required. Here is the list of required packages:

- JDK 8 or higher
- Apache ANT (optional)

7.2.2 Installation process

To install POP-Java we need to launch the command `./gradlew build` int the POP-Java directory. Once the source code is compiled, launch the installation with the install script: `sudo ./install`. This script will guide us through the installation by asking us some questions. Be aware that if we install POP-Java in the default location we need the administrator rights. Please use the option `-E` with the sudo command to keep the environment variables.

Here is the output we should have on our shell:

```
[POP-Java installation]: Detecting java executable ...
[POP-Java installation]: Java executable detected under
  /usr/bin/java
[POP-Java installation]: Please enter the location of your desired
  POP-Java installation (default: /usr/local/popj ) :
/home/clementval/popj
[POP-Java installation]: Installing POP-Java under
  /home/clementval/popj ? (y/n)
y
[POP-Java installation]: Copying files ...
[POP-Java installation]: Generating configuration files ...
[POP-Java installation]: Generating object map file for the test suite
[POP-Java installation]: POP-Java has been installed under
  /home/clementval/popj. Please copy the following lines into your
  .bashrc files or equivalent

POPJAVA_LOCATION=/home/clementval/popj
export POPJAVA_LOCATION
POPJAVA_JAVA=/usr/bin/java
export POPJAVA_JAVA
PATH=$PATH:$POPJAVA_LOCATION/bin

[POP-Java installation]: Installation done.
```

At the end of the installation, the script asks to copy some environment variable declarations in the .bashrc or equivalent file. This step is mandatory to make POP-Java work correctly.

7.3 Test the installation

POP-Java includes a test suite. We can run this test suite to check if our POP system is correctly installed. To run this test suite, we need to launch the `launch_testsuite` script located in the

POP-Java installation location.

Here is the output we should get after the completion of the test suite:

```
./launch_testsuite
########################################
#   POP-Java 1.0 Test Suite started    #
########################################
POP-C++ detected under /home/clementval/popc
POP-C++ was not running. Starting POP-C++ runtime global services ...
Starting POPC Job manager service:
POPCSearchNode access point: socket://172.28.10.67:38331
Starting Parallel Object JobMgr service
socket://172.28.10.67:2711POP-C++ started
##############################
#   POP-Java standard test   #
##############################
Starting POP-Java test suite
Launching passing arguments test (test 1/6)...
Arguments test successful
Passing arguments test is finished ...
Launching multi parallel object test (test 2/6)...
Multiobjet test started ...
Result is : 1234
Multiobjet test finished ...
Multi parallel object test is finished...
Launching callback test (test 3/6)...
Callback test started ...
Identity callback is -1
Callback test successful
Callback test is finished...
Launching barrier test (test 4/6)...
Barrier: Starting test...
Barrier test successful
Barrier test is finished...
Launching integer test (test 5/6)...
i1 = 23
i2 = 25
i1+i2 = 48
Test Integer Successful
Integer test is finished...
Launching Demo POP-Java test (test 6/6)...
START of DemoMain program with 4 objects
Demopop with ID=1 created with access point : socket://127.0.1.1:39556
Demopop with ID=2 created with access point : socket://127.0.1.1:60575
Demopop with ID=3 created with access point : socket://127.0.1.1:50088
Demopop with ID=4 created with access point : socket://127.0.1.1:39475
```

(continues on next page)

(continued from previous page)

```
Demopop:1 with access point socket://127.0.1.1:39556 is sending his ID␣
↪to object:2
Demopop:2 receiving id=1
Demopop:2 with access point socket://127.0.1.1:60575 is sending his ID␣
↪to object:3
Demopop:3 receiving id=2
Demopop:3 with access point socket://127.0.1.1:50088 is sending his ID␣
↪to object:4
Demopop:4 receiving id=3
Demopop:4 with access point socket://127.0.1.1:39475 is sending his ID␣
↪to object:1
Demopop:1 receiving id=4
END of DemoMain program
Demo POP-Java test is finished...

####################################
#   POP-C++ interoperability test   #
####################################
popcc -o main integer.ph integer.cc main.cc
popcc -object -o integer.obj integer.cc integer.ph main.cc
./integer.obj -listlong > obj.map
Launching POP-C++ integer with POP-Java application test (test 1/2)
POPC Integer test started ...
o1 = 10
o2 = 20
10 + 20 = 30
POPC Integer test successful
POP-C++ integer with POP-Java application test is finishied ...
popcc -parclass-nobroker -c integer2.ph
popcc -o main integer2.stub.o integer.ph integer.cc main.cc
popcc -parclass-nobroker -c integer2.ph
popcc -object -o integer.obj integer2.stub.o integer.cc integer.ph
popcc -object -o integer2.obj integer2.cc integer2.ph
./integer.obj -listlong > obj.map
./integer2.obj -listlong >> obj.map
Launching Integer mix (POP-C++ and POP-Java) with POP-Java application␣
↪test(test 2/2)
i=20
j=12
i+j=32
Integer mix (POP-C++ and POP-Java) with POP-Java application test is␣
↪finishied ...
########################################
#   POP-Java 1.0 Test Suite finished   #
########################################
```

(continues on next page)

(continued from previous page)

```
Stopping POPC Job manager service...
Connecting to 172.28.10.67:2711....
POPCSearchNode stopped
JobMgr stopped
```

CHAPTER 8

Configuration

After installation of POP-Java we need to take some steps if we want to enable some advanced features.

To do this a small dedicated shell was created. To run it go into the POP installation directory and run:

```
$ java -javaagent:JarFile/popjava.jar -cp JarFile/popjava.jar popjava.
↪scripts.POPJShell
This shell is not interactive, you must type every command.
Use ``help`` to know the available commands.
Every command has a --help (-h) flag which print its options.
```

This will open the shell. To execute a command simply write it and press enter. No history is available at this time.

```
$ help
Available options:
  help                      print this help
  jm                        configuration of the local job manager
  debug                     toggle system debug option
  keystore                  all keystore related operations.
```

8.1 About the Shell

Every command in the shell has a `help` method, usually by adding `-h` or `--help` to it. When it asks for a missing value is because that value could have been given by an option. See below.

```
$ jm node add -h
usage: jm node add [OPTIONS]
add a new node to a network
Available options:
  --type, -t          The type of node we are working with (jobmanager,
↪ tfc, direct)
  --uuid, -u          The UUID of the network to add the node into
  --host, -H          The destination host of the node
  --port, -p          The destination port of the node
  --protocol, -P      The node specific protocol (socket, ssl, daemon)
  --certificate, -c   The certificate for the SSL connection
Node specific options will be asked.
```

8.1.1 TLS Configuration

In case you want to use secure connections, you first have to create a keystore. Using the `keystore create` command the command will ask us to insert all the needed values. It will also save the keystore information in the POP-Java's global configuration so users will be able to use secure connection too.

```
$ keystore create
missing value for 'file': global.jks
missing value for 'storepass':
missing value for 'keypass':
missing value for 'alias': localNodeOnly
missing value for 'rdn': OU=PopJava,CN=testNode
Generating keystore...
Saving configuration...
```

8.1.2 Job Manager Configuration

If there is not a third party application to configure the Job Manager, the shell also partially give this capability.

The first thing to do is start the Job Manager.

```
$ jm start
Job Manager started.
```

With this we can now interact with it.

8.2 Network creation

To create a new network you will have to execute the `jm network create` command. Its output should something like the folowing snippet.

```
$ jm network create
missing value for 'name': friendly net
missing value for 'uuid':
Network 'friendly net' created with id [d3fe0096-e582-4b85-bdc0-
↪a429b169d24f]
Network certificate available at '/home/dosky/pop-java-dist/friendly␣
↪net@d3fe0096-e582-4b85-bdc0-a429b169d24f.cer'
```

The command will also export a `.cer` file which can be shared with trusted parties to communicate with them.

Note: The UUID value is what really identify the network, if someone else want to communicate with you it has to create a network matching the generated UUID in the command above. This means not leaving it blank.

You can see the existing network by running `jm network list`

```
$ jm network list
Note that networks are identified by their UUID.
+------------------------------------------+---------------------------
↪-----+
| UUID                                     | Friendly name            ␣
↪     |
+==========================================+================================+
| d3fe0096-e582-4b85-bdc0-a429b169d24f     | friendly net             ␣
↪     |
+------------------------------------------+---------------------------
↪-----+
```

8.3 Adding friendly nodes

Similarly to how we add network, a command exists in order to add friendly nodes.

```
$ jm node add
missing value for 'type': jobmanager
missing value for 'uuid': d3fe0096-e582-4b85-bdc0-a429b169d24f
missing value for 'host': <host>
missing value for 'port': <port>
missing value for 'protocol': ssl
missing value for 'certificate': other certificate.cer
Node added to network 'd3fe0096-e582-4b85-bdc0-a429b169d24f'
```

Note: Currently there exists three `type` of node: tfc, jobmanager, direct.

Currently there exists two `protocol`: socket, ssl.

When working with `ssl` a certificate is needed and the connection will be encrypted, while `socket` will be unencrypted.

8.3.1 Executing object as another user

Generally speaking the Job Manager on a machine has access to sensitive information like the content of the keystore. We don't want anyone except the system administrator to be able to modify those files.

8.3.2 Other options

POP-Java is very flexible, most of its options can be user configurable.

The shell by itself doesn't give the possibility of setting most of those options, bu they can be manually modified by adding the keyword and the value in the `popjava.properties` file situated in the `etc` directory of the POP installation.

A use can potentially modify those option for its own application by adding a `-configfile=<file>` option at the program execution.

For more information in regards of the options, check the `popjava.util.Configuration` class in the Javadoc or the developer Configuration section.

CHAPTER 9

Troubleshooting

9.1 POP-Java exception

This section lists some of the main POP-Java exception that can occurred during the application execution and gives the cause of the problem.

9.1.1 Cannot bind to access point: socket://your-computer-name:2711

This exception occurs when the application cannot contact the POP-C++ runtime system. To fix this problem, we need to start the POP-C++ runtime system with the following command:

```
POPC_LOCATION/sbin/SXXpopc start
```

9.1.2 Error message: `OBJECT_EXECUTABLE_NOTFOUND`

This exception occurs when the executable file is not found. This might be due to a bad object map or the deletion of the object executable file. To fix this problem we should generate a new object map with the object executable.

9.1.3 Error message: `NO_RESOURCE_MATCH`

This exception occurs when no resource match the requirements of a specific object. To fix this problem we should check the object descriptions in the parallel objects. We might have put a too high requirement for a parallel object creation.

9.1.4 Error message: `Cannot run program "/usr/local/popc/services/appservice"`

If we get an error with "cannot run program" and the path contains the appservice of POP-C++, you have certainly reinstalled POP-C++ and the configuration file of POP-Java is now wrong. The easiest way to fix this problem is to reinstall also POP-Java. We can also edit the configuration file under `POPJAVA_LOCATION/etc/popj_config.xml`. The item `popc_appcoreservice_location` must be modified with the good path.

9.1.5 Test suite frozen

If the test suite seems to be frozen, we should abort the test suite and restart the POP-C++ global service with the following command:

```
POPC_LOCATION/sbin/SXXpopc restart
```

CHAPTER 10

The TFC model

TFC (Trusted Friend Computing) is a POP-Java object sharing model, which can be implemented in any context.

The model allows to share objects and resources of a given program or module between nodes in a network, without having to run any third party code on a given machine (see 1).

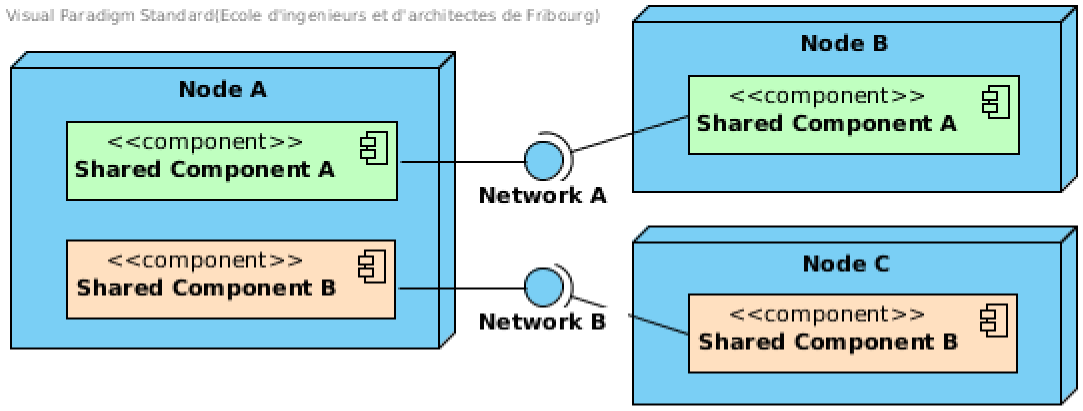


Fig. 1: Nodes sharing the same program/module

A typical use case of TFC appears e.g. when someone wants to publish an object and someone else to find and use that object (see basic and advanced examples in the next sections).

Todo: explain better

CHAPTER 11

Basic example

In this section, we explain how to publish an object and how to retrieve one inside a friend network.

11.1 Publishing TFC objects

TFC requires that a connected *friend* shares a known application consisting of a POP object.

Let's take the following class as an example:

```
@POPClass
public class A {

        private int n;

        public A() { }

        @POPObjectDescription(url = "localhost", protocols = "ssl")
        public A(int n) {
                this.n = n;
        }

        @POPSyncConc
        public int get() {
                return n;
        }
}
```

The class A is very basic. It only offers one method to retrieve a stored value *n*.

The following application shows how to publish an instance of A to our network:

```
@POPClass(isDistributable = false)
public class TFCPublish {
        private static final String NET = "abc-123-def";
        public static void main(String[] args) throws IOException {
                A a = new A();
                System.out.println("Object at " + PopJava.
↪getAccessPoint(a));

                System.out.println("Publishing via JMC...");
                JobManagerConfig jmc = new JobManagerConfig();
                jmc.publishTFCObject(a, NET, "mysecret");

                System.out.println("Press enter to destroy the object..
↪.");
                System.in.read();
        }
}
```

At line 5, we can see a standard POP object creation. At line 9, we connect to the Job Manager using its configuration API class. At line 10, we publish the created object by giving it to the Job Manager and setting which network should be able to search for the object. The secret allows to remove the object from the published list without killing it.

Note: We took an unique network identifier abc-123-def for sake of simplicity. See *Configuration* to learn how to create networks.

11.2 Searching for a TFC object

To search for an object, we require to know its interface (and not necessarily its implementation). In our example, we have its implementation.

Note that friend nodes may be offline at the time of search, meaning that multiple searchs for the same object might yield different results.

Retrieving objects require the following steps:

1. Setup of the search parameters (lines 6,7);
2. Initialization of the search (line 8);
3. Connection to the obtained results (line 12).

```
@POPClass(isDistributable = false)
public class TFCRetrieve {
        private static final String NET = "abc-123-def";
        public static void main(String[] args) {
                System.out.println("Retrieving from network...");
                ObjectDescription od = new ObjectDescription();
                od.setNetwork(NET);
                POPAccessPoint[] aps = PopJava.newTFCSearch(A.class,␣
↪10, od);
                System.out.println("Got " + Arrays.toString(aps));

                for (POPAccessPoint ap : aps) {
                        A r = PopJava.connect(A.class, NET, ap);
                        System.out.println(ap + " -> " + r.get());
                }
        }
}
```

The *ObjectDescriptor* at line `6,7` sets the network in which we will search for objects. There we can also specify some options such as `setSearch` for the depth of the search, or `setSearchHosts` to select the hosts which are allowed to answer us from the host list.

The API call to `PopJava.newTFCSearch` requires the class we are looking for, the maximum number of instances we want and the *ObjectDescriptor* as parameters.

In order to connect to an existing object, we use the `PopJava.connect` method which requires the network, the remote object and its address.

Note: The current way to publish and retrieve objects require several API calls, which is not in the best spirit of a POP model (KISS). This might be simplified in the future.

CHAPTER 12

Advanced example

In this section, we detail a more complex example of a TFC use case. In this example, we don't have any default network and we publish only partially on the available networks.

We also introduce some advanced annotations such as `@POPPrivate` that can be used in more complex applications, and new parameters such as `localhost`, `tracking` and `localJVM`.

```
@POPClass
public static class A {

        private int n;

        public A() { }

        @POPObjectDescription(url = "localhost", protocols = "ssl",
                tracking = true, localJVM = true)
        public A(int n) {
                this.n = n;
        }

        @POPSyncSeq
        public int get() {
                return n;
        }

        @POPAsyncMutex(localhost = true)
        public void set(int n) {
```

(continues on next page)

(continued from previous page)

```
                this.n = n;
        }

        @POPSyncSeq
        @POPPrivate
        public void divide() {
                n /= 2;
        }
}
```

This class does the same as the one in the preceding example: it exposes a value to whoever wants to know it, except that now we have the possibility to modify the value even after the object creation. **However, this feature is not for everyone!**

12.1 Local JVM objects

We can see in the `@POPObjectDescription` annotation that we have two new attributes:

1. `tracking`, which allows to know who used the object;
2. `localJVM`, which allows to integrate the object in the current JVM instead of spawning a new one (see 1).

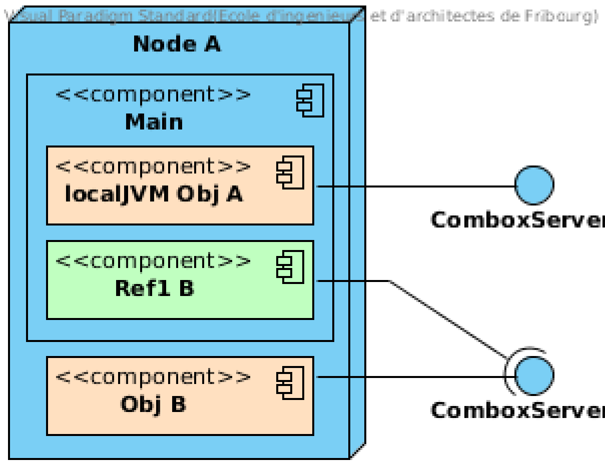


Fig. 1: Main create two POP objects: the first is a `localJVM` and the second a classic JVM.

Why creating an object this way? There are multiple reasons, the main one being that a POP object spawned locally does not require the data to be transmitted via a Combox and has access to all non-POP objects created in the JVM. This notably permits to make data accessed from a non-POP platform available to a POP application.

That said, this not an all in one solution: `localJVM` should be used with care! The annotations used to achieve synchronicity may not work (particularly `async`), unless we treat the object as if it were remote.

```
A al = new A(10); // local JVM
A ar = PopJava.getThis(al); // connect to the local JVM object
```

In the example above `al` is a `localJVM` object and is treated as such. `ar` also points to the same object `al` but must pass through a `Combox` to make calls. Thus, it also loses access to some methods.

12.2 Local special access method

`localJVM` is generally used to make a hybrid application working with non-POP objects. Thus, some object methods might not be available for every connecting client but only for the JVM which created the object itself.

The `@POPPrivate` annotation is meant for keeping a method accessible to the JVM which created the object and not exposing it remotely.

```
// Node A
A local = new A(10); // local JVM
A ref1 = PopJava.getThis(local); // Connect to the local JVM object

// Node B
A ref2 = PopJava.connect(...) // Connect to nodeA -> local remotely
```

In the code above, we create a `localJVM` object and connect to it by creating a reference. Then we have a remote machine which is also connected to it. Figure 2 shows the situation.

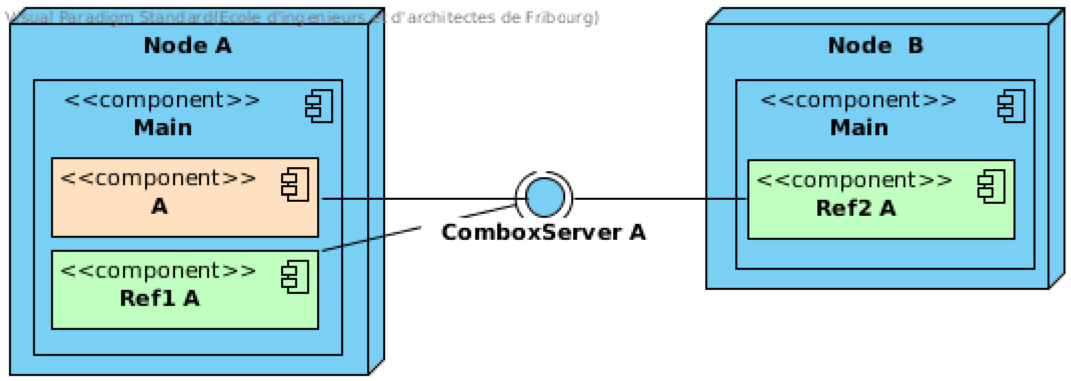


Fig. 2: Local JVM with local and remote connections

In this `local` example, we can call the method `divide`; `ref1` and `ref2` do not have this method exposed because it is annotated with `@POPPrivate`.

12.3 Remote special access method

`@POPPrivate` is not the only restriction that we can make. The `set` method has an attribute in its annotation: `localhost = true`. This attribute automatically checks that the calls to this method come from someone on the same machine than the object.

In the same preceding example, we can see that the `set` method is not accessible by everyone, but only by objects on Node A. Table below shows the access to the three methods of `A`.

Method	local	ref1	ref2
get			
set			
divide			

12.4 Tracking

Tracking allows to know how long an object was used and by who. Calls to a specific API with a POP object as a target permit to obtain this information.

Let's take the same example used in the two previous chapters: one object receives two connections from two different sources and is also used locally.

Note: It's important to know that we can not track the usage of a `localJVM` object, unless we are connected to it via a Combox. This means that we will never know how `local` uses A.

The person who created the POP object has access to its usage statistics. In fact, only the owner of the object knows all the users who used it.

The following information is usually extracted from a connecting user: the certificate (if present) used to identify the user, the IP address and the network used for the connection.

Note: POP-Java does not handle the real identification of a user. It's the job of the one creating an application to provide this ability.

To access the statistics of a POP object, we use the API provided by the class `POPAccounting`, which can

- Check if an object has tracking enabled
- Retrieve the users which used the object
- Ask the statistics of a given user

- Ask the statistics of a current connection

12.4.1 Own statistics

Accessing your own statistics can be useful to check how much you used another person's‘ shared object before closing a connection. The usage is stacked and connection independent, meaning that the statistics cumulate and are not reset between two connections to an object.

```
POPTracking own = POPAccounting.getMyInformation(a);
```

`POPTracking` contains information that the owner of the object can see about you, in order to identify you and see your usage of the methods in the object.

12.4.2 Object statistics

As the owner of an object, you may be interested in knowing who used your object. To do that, you first need to request a list of users of the object. Then you can successively ask detailed information about each user.

```
POPRemoteCaller[] users = POPAccounting.getUsers(a);
for (POPRemoteCaller user : users) {
        POPTracking info = POPAccounting.getInformation(a, user);
        // do something
}
```

12.4.3 Tracked information

We generally track three things done by the user:

- the methods he used;
- the number of times he called each method;
- the duration of the method execution (usage);
- the size of the method input data;
- the size of the method output data.

With those information, the owner of an object can gather enough information e.g. to fill in an invoice.

CHAPTER 13

Contribution guidelines

Contributing to POP-Java is very simple. All the code is hosted in a public git repository hosted on gitlab. It can be found at <https://github.com/pop-team/pop-java> and is open to merge requests for new developments.

13.1 Coding conventions

When writing new code for POP-Java you should always:

- Indent with 4 spaces
- Always surround blocks with `{ }`
- . . .
- . . .

13.2 Creation of a new release

1. Update changelog file

    All notable changes to the POP-Java project will be documented in the `CHANGELOG.md` file as follows (source: SemVer). Given a version number **MAJOR.MINOR.PATCH**, increment the:

    - **MAJOR version**, when incompatible API changes are made;

- **MINOR version**, when functionalities in a backwards-compatible manner are added;
- **PATCH version**, when backwards-compatible bug fixes are made.

Additional labels for pre-release (e.g. Beta, RC1) and build metadata may be added as extensions to the MAJOR.MINOR.PATCH format.

For each new version released, the related section will list its novelties under the following potential sub-sections: Features, Bug Fixes and BREAKING CHANGES.

New functionalities, which are not yet released, will be listed at the top of the CHANGELOG.md under the so-called UNRELEASED section.

2. Update version

   Increment the version number of POP-Java in the `build.gradle` file.

   Note: This step is mandatory in order to publish a new Maven release.

3. Update author file

   All current and/or previous authors (*core committers*) shall be listed in the `AUTHORS` file as follows:

```
Current core committers:

* Beat Wolf
* Jonathan Stoppani
* Christophe Gisler


Previous core commiters:

* Beat Wolf
* Davide Mazzoleni
* Valentin Clément
```

4. Build Jar

   A fat Jar version of POP-Java must be built locally in order to run the tests by using the following command:

```
$ ./gradlew fatJar
```

   Note: Make sure you use Java JDK 8 (not 9) in order to build POP-Java. Otherwise it will not run under Java 8.

5. Run tests locally

   POP-Java must be tested locally by using the following command:

   ```
   $ ./gradlew test
   ```

   All tests must pass before going to the next step.

6. Build and upload Maven package to OSSRH

   Build the POP-Java Jar files and signing files required for the Maven package, and upload (deploy) them to the OSSRH repository by using the following commands:

   ```
   $ ./gradlew clean
   $ ./gradlew uploadArchives
   ```

   Note:

   - We first clean the build directory to get rid of the fat Jar bundle, which must not be deployed to the OSSRH repository.
   - To perform this step, one must have a Sonatype JIRA login and credentials in his gradle.properties file (generally stored in `~/.gradle/`) like this:

     ```
     signing.keyId=YourKeyId
     signing.password=YourPublicKeyPassword
     signing.secretKeyRingFile=PathToYourKeyRingFile

     ossrhUsername=your-jira-id
     ossrhPassword=your-jira-password
     ```

   - The signing data must be generated, e.g. with GnuPG.
   - Newer (2.1+) GPG versions use a new keyring file format (.kbx). You need to convert/export your key to the old format.

     ```
     gpg --export-secret-keys -o secring.gpg
     ```

   - With the 2.1+ GPG version you also need to use a special command to list the available keys to get the correct id.

     ```
     gpg --list-keys --keyid-format short
     ```

   - More information about the Maven packaging process is given on the OSSRH Guide.

7. Commit, tag and push

Commit your changes to the project, tag your version and push them:

```
$ git commit -m "My commit message"
$ git tag -a v2.1.0 -m "my version 2.1.0"
$ git push origin master
$ git push --tags
```

8. Wait for tests to pass and documentation to build

   Here nothing to do but wait. While one or more tests fail, please fix the related bugs and go back to previous step.

9. Update release details on GitHub

   Please follow these steps:

   1. Go to the GitHub release page;
   2. Click on the new release link;
   3. Click on the `Edit tag` button (on the top right of the page);
   4. Fill in the related fields;
   5. Click on the `Publish release` button.

10. Release deployed Maven package from OSSRH to the Central Repository

    Automatically close and release the staging version from OSSRH to the Central Repository by using the following command:

    ```
    ./gradlew closeAndReleaseRepository
    ```

    Note:

    - To pass this step, the deployed files are verified and thus must fulfil some requirements.
    - This step was fully automatized thanks to the Gradle Nexus Staging Plugin. However, it can manually be done on the OSSRH website as described here.
    - It takes about 2 hours to synchronize between OSSRH and the Central Repository

CHAPTER 14

Documentation management

This chapter describes all the processes related to the documentation of the POP-Java project.

14.1 Documentation types

The POP-Java project currently has two main types of documentation:

Prose documentation The documentation you are reading right now. It contains manuals, guides and how-tos about the project itself. The prose documentation is written by humans for humans. This documentation is currently edited using Sphinx.

API reference The API Reference contains the documentation for each functional unit of the project (packages, classes, methods, . . .) and is generated by parsing the source code. The API reference is currently being generated using Doxygen.

14.2 Editing the prose documentation

As mentioned in the previous section, the prose documentation is managed by Sphinx. Sphinx is a tool to create documentation from reStructuredText sources and can produce a variety of output formats (e.g. HTML, LaTeX -> PDF, ePub, Texinfo, man pages, plain text, . . .).

The prose documentation resides in the `docs/` directory at the root of the repository and can be edited with any text editor. The complete reference for the reStructuredText syntax and the addons added by Sphinx are available on the reStructuredText (see also http://sphinx-doc.org/rest.html) and Sphinx Markup Constructs pages, respectively.

For quick and small edits, it is possible to use the GitHub editing interface instead of a complete local clone + edit + commit + push process. The documentation files are available on GitHub in the gridgroup/pop-java repository.

14.3 Building and publishing the prose documentation

The building process transforms the input documents in reStructuredText format into one of the output formats supported by Sphinx (HTML, PDF, ePub, . . .).

There are two different ways to build the documentation:

- Locally on the development machine, by installing the necessary tools and executing the right command.
- Remotely, via a custom post-commit hook on a CI server or through a specialized service, such as http://readthedocs.org/.

In the next two subsections we will discuss these two possibilities. In the second case we will limit ourselves to building through http://readthedocs.org/.

14.3.1 Locally

To build the documentation locally we need a working Sphinx installation.

Sphinx requires at least Python 2.5 or 3.1; you can read more about additional details regarding the requirements on the Sphinx introduction page.

To check which version of python you have installed, run the following command:

```
python -V
```

If python is not installed or does not meet Sphinx's version requirements, you can either install or update it by using your distribution package manager (`apt-get` on Debian/Ubuntu, `yum` on CentOS/Fedora, `emerge` on Gentoo, . . .).

Below we report some easy steps to install Sphinx on your system. For a more complete walk-through (including platforms such as OS X and Windows), you can always refer to the Sphinx installation instructions.

To install Sphinx, you can use either `easy_install`:

```
easy_install sphinx
```

or `pip` (recommended):

```
pip install sphinx
```

Depending on the platform and your setup, you probably have to run these commands with administration privileges:

```
sudo pip install sphinx
```

You can check if Sphinx was successfully installed by running the following command:

```
python -c 'import sphinx'
```

In addition to Sphinx itself, this documentation makes use of one additional required and one additional optional packages:

- The `bibtex` extension allows to use BibTeX databases to manage references. You can install it by executing:

  ```
  pip install sphinxcontrib-bibtex
  ```

- The `sphinx_rtd_theme` is the theme used on Read The Docs. If the corresponding package is installed, it will be used for local builds as well. The theme can be installed by running:

  ```
  pip install sphinx_rtd_theme
  ```

Note: If you had to use `sudo` When installing Sphinx, then the commands to install these additional packages will have to be prefixed with it as well.

Once you have a working Sphinx installation on your system, it's time to start building your documentation. As the initial setup is already done for the POP-Java project, you don't have to configure anything.

Different ways exist to build the documentation. The easiest is by far to use the generated `Makefile` present in the `docs/` directory:

```
cd path/to/pop-java/docs/
make html
```

If successfull, the last line of the command output will indicate the location of the build. In this case, the HTML files will be located in `_build/html`.

To publish the generated documentation, it suffices to upload the `html` directory (or the equivalent artifact for a different output format) to a publicly accessible web server.

In order to support the other output formats managed by Sphinx, the `Makefile` has different additional targets. You can find out more about them by running the `help` target, whose output is shown below:

```
Please use `make <target>' where <target> is one of
  html       to make standalone HTML files
  dirhtml    to make HTML files named index.html in directories
  singlehtml to make a single large HTML file
  pickle     to make pickle files
  json       to make JSON files
  htmlhelp   to make HTML files and a HTML help project
  qthelp     to make HTML files and a qthelp project
  devhelp    to make HTML files and a Devhelp project
  epub       to make an epub
  latex      to make LaTeX files, you can set PAPER=a4 or PAPER=letter
  latexpdf   to make LaTeX files and run them through pdflatex
  latexpdfja to make LaTeX files and run them through platex/dvipdfmx
  text       to make text files
  man        to make manual pages
  texinfo    to make Texinfo files
  info       to make Texinfo files and run them through makeinfo
  gettext    to make PO message catalogs
  changes    to make an overview of all changed/added/deprecated items
  xml        to make Docutils-native XML files
  pseudoxml  to make pseudoxml-XML files for display purposes
  linkcheck  to check all external links for integrity
  doctest    to run all doctests embedded in the documentation (if␣
↪enabled)
```

14.3.2 readthedocs.org

Read the Docs (RTD) is a free service to build and host Sphinx documentation sets. It supports polling any GIT or Mercurial repository and re-running a build each time a new commit is detected. The POP-Java documentation is currently available on RTD at the following link: http://pop-java.readthedocs.org/en/latest/.

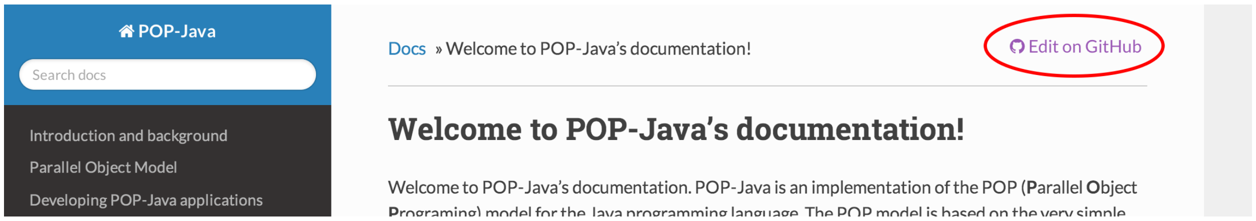
In order to update the documentation hosted on RTD, it suffices to commit the changes to the GIT repository and push them to the GitHub remote:

```
git commit -m 'Documentation update'
git push origin master
```

The repository hosted on GitHub is configured with a post-commit hook to trigger a new RTD build and the updated version should be available in a short time (usually < 2 minutes).

Thanks to RTD's integration with GitHub, an even easier way to carry out small, self-contained edits to the documentation is directly through the GitHub editing interface:

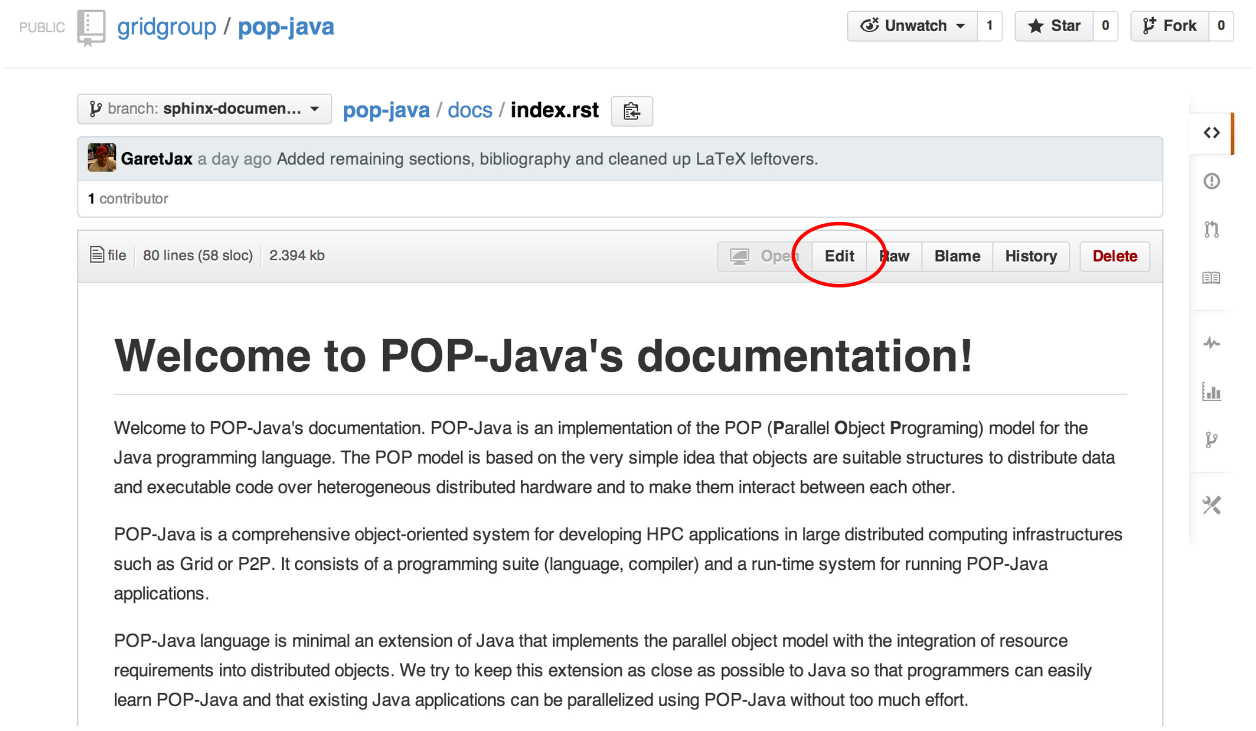
- Each HTML page generated on RTD contains a GitHub edit link in the right corner which brings up the GitHub interface browsing.


POP-Java
Search docs
Introduction and background
Parallel Object Model
Developing POP-Java applications
Docs » Welcome to POP-Java's documentation!
Edit on GitHub

Welcome to POP-Java's documentation!

Welcome to POP-Java's documentation. POP-Java is an implementation of the POP (**P**arallel **O**bject Programing) model for the Java programming language. The POP model is based on the very simple

- We can then click on the edit button to enter the editing interface.


PUBLIC gridgroup / pop-java
Unwatch 1
Star 0
Fork 0
branch: sphinx-documen...
pop-java / docs / index.rst
GaretJax a day ago Added remaining sections, bibliography and cleaned up LaTeX leftovers.
1 contributor
file 80 lines (58 sloc) 2.394 kb
Open
Edit
Raw
Blame
History
Delete

Welcome to POP-Java's documentation!

Welcome to POP-Java's documentation. POP-Java is an implementation of the POP (**P**arallel **O**bject **P**rograming) model for the Java programming language. The POP model is based on the very simple idea that objects are suitable structures to distribute data and executable code over heterogeneous distributed hardware and to make them interact between each other.

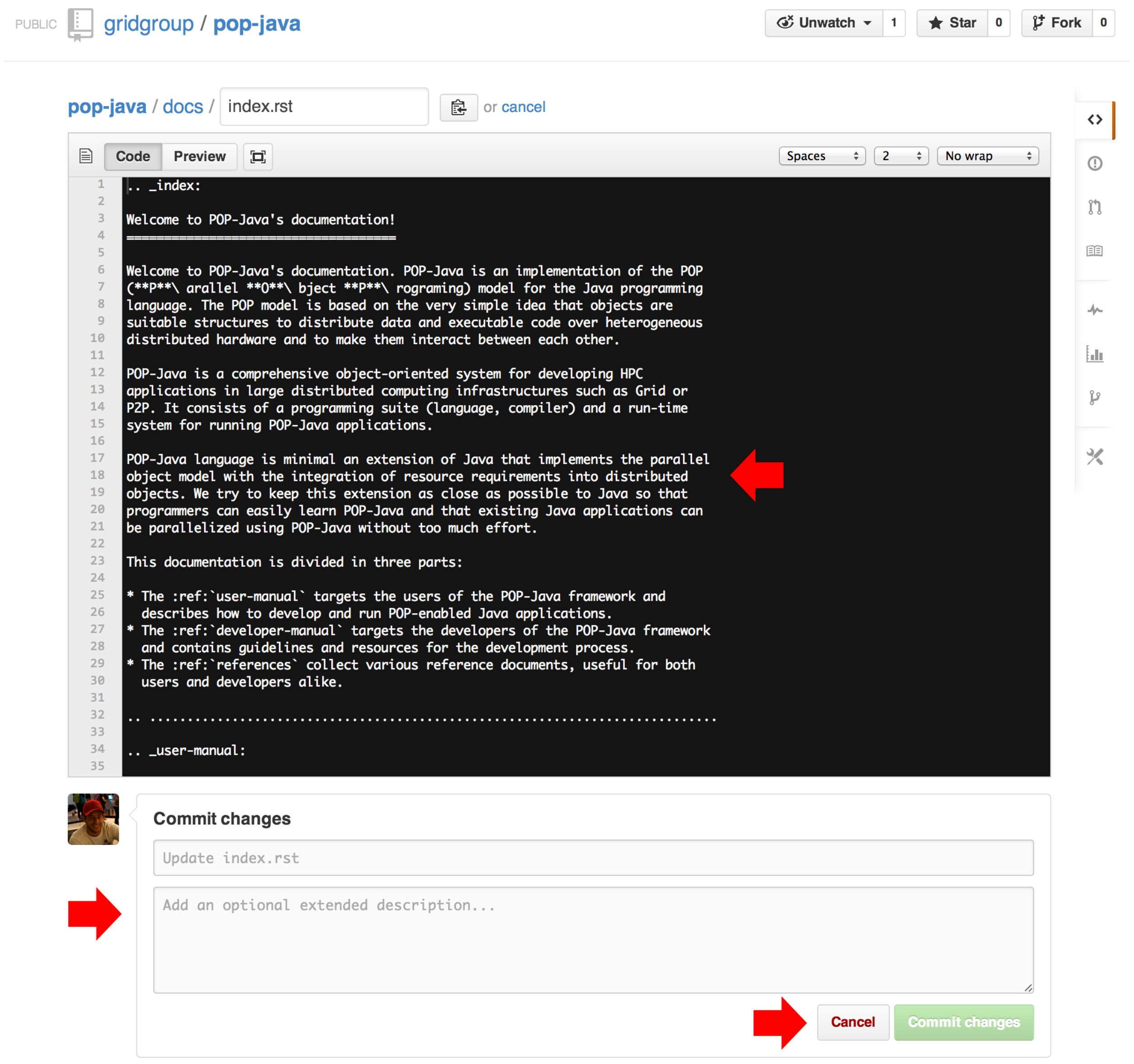
POP-Java is a comprehensive object-oriented system for developing HPC applications in large distributed computing infrastructures such as Grid or P2P. It consists of a programming suite (language, compiler) and a run-time system for running POP-Java applications.

POP-Java language is minimal an extension of Java that implements the parallel object model with the integration of resource requirements into distributed objects. We try to keep this extension as close as possible to Java so that programmers can easily learn POP-Java and that existing Java applications can be parallelized using POP-Java without too much effort.

- From there we can carry out the desired changes, commit them directly (if the account with which we are logged in to GitHub allows it; fork and open a pull request otherwise) and have the documentation hosted on RTD updated automatically.

14.4 Building and publishing the API reference

In the introduction to the present chapter we mentioned that the API Reference is currently being generated using Doxygen. In the following paragraphs we will describe how the documentation can be rebuilt and how the resulting artifact can be published on the GitHub pages service.



```
.. _index:

Welcome to POP-Java's documentation!
====================================

Welcome to POP-Java's documentation. POP-Java is an implementation of the POP
(**P**\ arallel **O**\ bject **P**\ rograming) model for the Java programming
language. The POP model is based on the very simple idea that objects are
suitable structures to distribute data and executable code over heterogeneous
distributed hardware and to make them interact between each other.

POP-Java is a comprehensive object-oriented system for developing HPC
applications in large distributed computing infrastructures such as Grid or
P2P. It consists of a programming suite (language, compiler) and a run-time
system for running POP-Java applications.

POP-Java language is minimal an extension of Java that implements the parallel
object model with the integration of resource requirements into distributed
objects. We try to keep this extension as close as possible to Java so that
programmers can easily learn POP-Java and that existing Java applications can
be parallelized using POP-Java without too much effort.

This documentation is divided in three parts:

* The :ref:`user-manual` targets the users of the POP-Java framework and
  describes how to develop and run POP-enabled Java applications.
* The :ref:`developer-manual` targets the developers of the POP-Java framework
  and contains guidelines and resources for the development process.
* The :ref:`references` collect various reference documents, useful for both
  users and developers alike.

.. ..........................................................................

.. _user-manual:
```

14.4.1 Building

To build the documentation you need a working installation of Doxygen on your system. Extensive documentation about the installation process is available directly from the Doxygen manual.

Once installed, the steps needed to build the documentation are very simple:

```
doxygen doxygen.conf
```

The resulting build will be available in the `doxygen/html` and `doxygen/latex` directories. The HTML version is ready to use, while the LaTeX version needs and additional build to get to the PDF version:

```
cd doxygen/latex
make
```

The resulting PDF will be available at `doxygen/latex/refman.pdf`.

14.4.2 Publishing on GitHub pages

GitHub supports a basic kind of static website hosting, based on GIT repositories. This service is called GitHub pages; you can find out more about it on http://pages.github.com/.

Currently, the latest build of the HTML version of the API reference is made available for browsing at http://gridgroup.github.io/pop-java/api/ and the PDF version at http://gridgroup.github.io/pop-java/api/POP-Java.pdf.

The steps involved in the update process of the `gh-pages` branch (the branch from which the static website is made available on GitHub) are summarized as follows:

```
git checkout gh-pages                    # Checkout the 'gh-pages'␣
↪branch
mv doxygen/html api                      # Move the built HTML␣
↪reference into place
git commit -am 'API reference update'    # Commit everything
git push origin gh-pages                 # Push to GitHub (publish)
git checkout <oldbranch>                 # Go back to the branch we␣
↪were working on
```

To make it easier to build and publish the documentation, a script to automate the process is made available at `scripts/api-reference` from the root of the repository. In order to build, commit, and publish a new version of the API reference, it suffices to execute it with no arguments:

```
./scripts/api-reference
```

CHAPTER 15

Architecture

POP-Java's general architecture is not fundamentally different from alternatives like RMI, see 1. Like in RMI, POP's Interface act like RMI's Stub, while POP's Broker act like RMI's Skeleton, but the similarities end there.

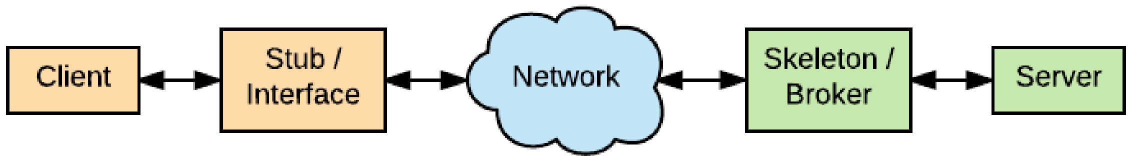


Fig. 1: POP-Java RMI similarity

The similarities end on the way object communicate with each other, since RMI require that object are registered on an RMI Server while every POP Object is independent and is a server on its own. We also need to write a single class with POP-Java while we need multiple for RMI.

15.1 Object Creation

The creation of a new POP Object always start in the same manner, via a call to `PopJava.newActive(...)`. This will call Javassist to wrap the `@POPClass` annotated class into a `POPObject`, creating a fictitious class used as a proxy to the actual instantiated remote object. See 2 to understand how the fictitious class is created.

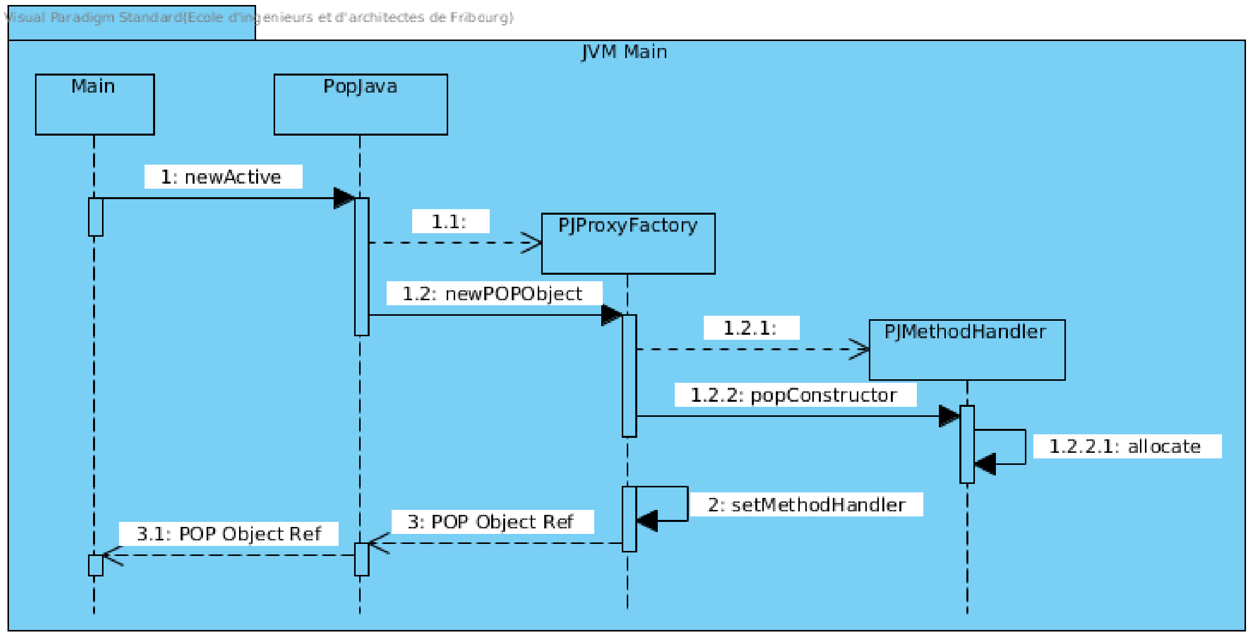


Fig. 2: Creation of Proxy class with Javaassist

The allocate method in 2 will handle the creation of a new JVM and the connection to the remote object. This happen transparently without the user intervention or knowledge.

3 shows a simplified version of how a new JVM is spawned and the connection between `PJMethodHandler` and `Broker` is established.

`bind`, at the end, connect `PJMethodHandler` and the `Broker` instance directly so they can communicate. `treatRequests` is a loop designed to handle all method calls toward an object.

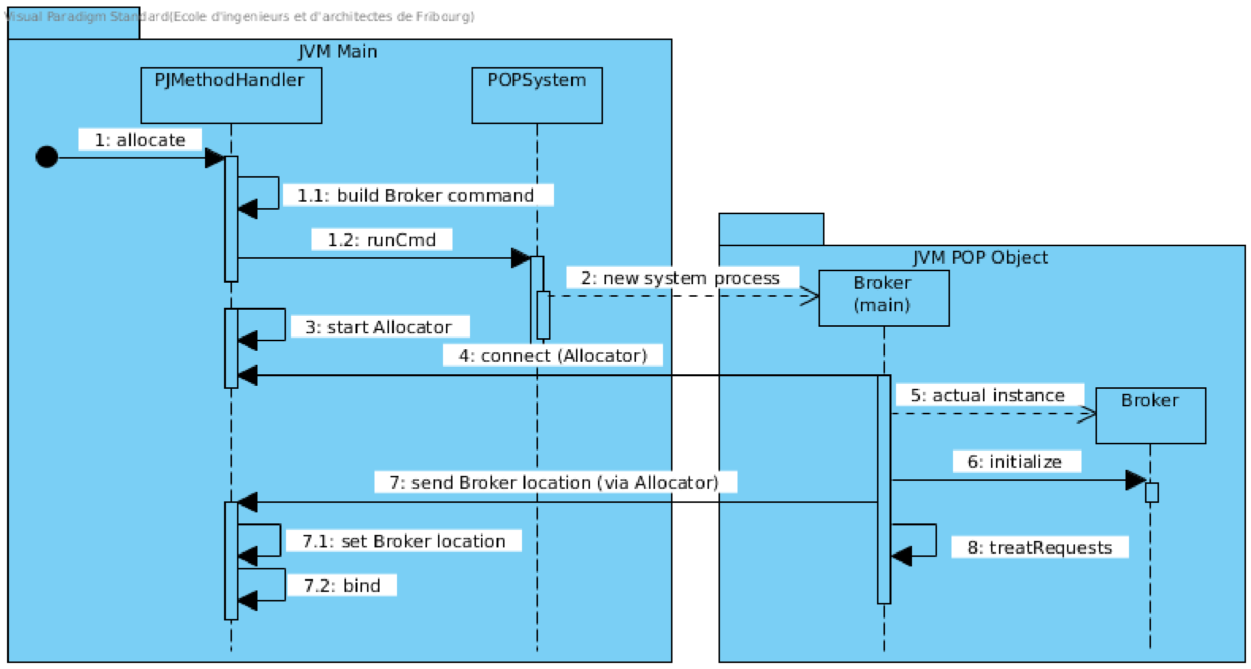


Fig. 3: PJMethodHandler (Interface) spawn a connect to a new JVM

CHAPTER 16

Broker & Interface

The Broker and Interface are the two fundamental blocks of POP-Java, the former represent the server instance of a POP Object while the latter is the client connecting to it.

16.1 Broker

The `Broker` class is a wrapper for an instance of a POP Object. Meaning that each instantiated POP Object will have its own Broker instance associated with it. It is also the entry point for newly created POP Objects.

16.1.1 General architecture

As we can see in 1 we have two distinct JVMs, one running the Main of the application while the second running the Broker's Main.

The Broker's Main will create a new `Broker` instance which will be the wrapper for the POP Object requested by the application Main. The instance will open one or more listening `ComboxServer` which will enable `Broker` to receive method calls from `PJMethodHandler` and send them to the wrapped object. See *Architecture* to understand how POP Objects are created.

Note: `PJMethodHandler` extends `Interface`, which is the base of communication with a `Broker`; without `PJMethodHandler` is what enable us to make remote calls to methods.

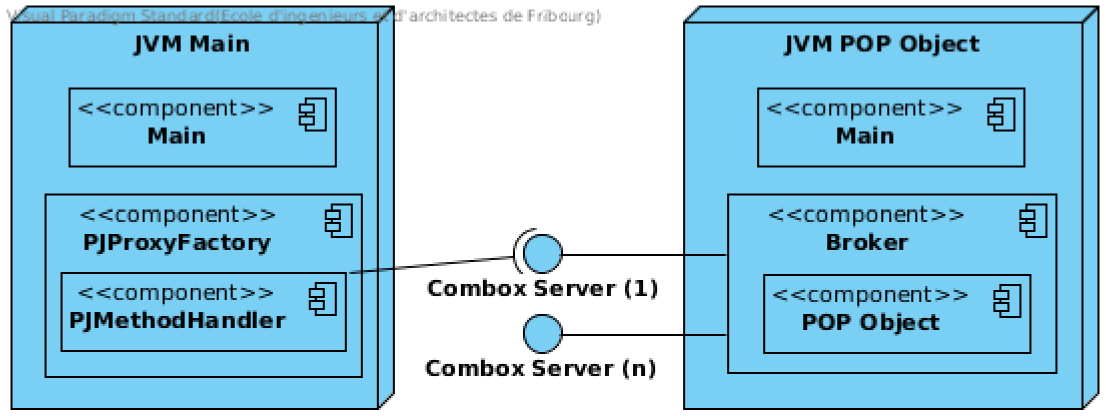


Fig. 1: Broker components

16.1.2 Entry Point

A `Broker` can be create in two ways: as the entry point (main) of a new JVM (1) or directly by `PJProxyFactory` if we decide *not* to create a new dedicated JVM.

Let's take a look at an example of the former case in a classic scenario, the arguments received by the `Broker` main are:

```
-object=MyPopObject
-codelocation=http://myapp.com/app.jar
-callback=socket://IP:PORT1
-appservice=socket://IP:PORT2 ssl://IP:PORT3
-socket_port=0
-ssl_port=0
```

- `object` is the name of the class of our POP Object.
- `codelocation` tell the Broker where to load `object` from.
- `callback` is an immediate location to report which `ComboxServer Broker` has opened.
- `appservice` the location where `AppService` can be accessed from.
- `<PROTOCOL>_port` open a `ComboxServer` of type `PROTOCOL`.

If the user is using a personal POP-Java configuration file and used `@POPObjectDescription(url = "localhost")` a extra parameter `configfile` is added with the path to the file which will be loaded by the `Broker`.

16.1.3 Initialization

In both *Entry Point‘s the method ‘‘public boolean initialize(java.util.List<java.lang.String>);‘* is called, the main objective of this method is to start `ComboxServer` in relation to how many `<PROTOCOL>_port` were supplied.

In the case no particular protocol was supplied, the default one specified in *DEFAULT_PROTOCOL* will be used.

Note: It’s possible to open multiple `ComboxServer` of the same type by supplying `<PROTOCOL>_port` multiple times.

16.2 Interface & PJMethodHandler

`Interface` is the class we use to allocate a new JVM and connect to a new POP Object; `PJMethodHandler`, instead, by extending it, add the ability to invoke the POP Object methods.

Generally `Interface` enable us only to connect to a POP Object, while `PJMethodHandler` allow us to use it.

Note: Currently `Interface` contains multiple static methods used to run a command on the local machine and lauch a new JVM. Those methods, name-wise and location-wise, are confusing and should be moved.

CHAPTER 17

Packages

POP-Java is divided in two parts, the first is a small public APIs for the user to make use of for its application in the simplest way possible, while the second is the core POP-Java's runtime.

In this chapter we will explore the two and their details.

17.1 Public packages and classes

This section explain the public available classes a user can use to make his POP application.

In POP the User never has to interact with the framework directly apart in very specific occasions. Which means that the public POP API are a simple set of annotations and some helper classes, as we can see below.

If we want to add new functionality visible to the User adding parameter in one annotation or adding a method `popjava.PopJava` should be the preferred way to go.

In case of very specific API, like `popjava.JobManagerConfig` used for configuring `POPJavaJobManager`, it is acceptable to add a new class to the public API.

`popjava.annotation.`
The minimum needed to use POP. This package contains all the `@POP` annotations for methods and classes.

Note: We have 6 different method annotations and, generally, if we want to add an option to one we add it to the six of them, or at least half of them.

popjava.**PopJava**

Needed for some POP specific tasks like getting the Access Point of a POP Object.

It contains all methods to initiate a new POP Object.

popjava.**JobManagerConfig**

Used to configure the JobManager on the local machine. It's a Proxy to methods of `POPJavaJobManager`.

popjava.util.**Configuration**

If the user need some more control on the behavior of POP-Java. Controls include timeouts and the defaults used in various situations.

popjava.baseobject.**ConnectionProtocol**

Used by `@POPObjectDescription(connection = ...)` to define the direct method of connection used.

Note: JobManagerConfig is a special class that enable the configuration of a peculiar POP service, for this reason it access to extra classes.

17.2 Internal packages and classes

This section explain the most important internal classes used by the POP Runtime.

Note: We are not going to describe every class in this section, only the most important which may need further explanation.

When modifying POP's internal classes be sure to run the JUnit tests before and after and ensure that no new error are generated from the modification. If what is added is designed to fail in some scenarios it is advised to add a new test to JUnit, to see how to do this see *Testing*.

popjava.**PJProxyFactory**

Use Javaassist to wrap a class into a POP Object. In `newPOPObject` it will create a new `PJMethodHandler` which will also create the new JVM for the POP Object.

popjava.**PJMethodFilter**

A helper class to knew which method are to be handled by `PJMethodHandler`, it also contains a *static* set of special POP methods which are to be handled internally.

popjava.**PJMethodHandler**

Extends `Interface` and add the ability of calling methods. The methods in the special set in *PJMethodFilter* are implemented here.

`popjava.interfacebase.`**`Interface`**
Handle the connection with a `Broker` instance and how to communicate with it.

`popjava.annotation.processors.`**`POPClassProcessor`**
‘‘ ‘‘

`popjava.base.`**`POPObject`**
‘‘ ‘‘

`popjava.base.`**`POPErrorCode`**
‘‘ ‘‘

`popjava.baseobject.`**`ObjectDescription`**
‘‘ ‘‘

`popjava.baseobject.`**`AccessPoint`**
‘‘ ‘‘

`popjava.baseobject.`**`POPAccessPoint`**
‘‘ ‘‘

`popjava.broker.`**`Broker`**
‘‘ ‘‘

`popjava.broker.`**`Request`**
‘‘ ‘‘

`popjava.buffer.`**`POPBuffer`**
‘‘ ‘‘

`popjava.buffer.`**`BufferXDR`**
‘‘ ‘‘

`popjava.buffer.`**`BufferRaw`**
‘‘ ‘‘

`abc`
‘‘ ‘‘

`def`
‘‘ ‘‘

Todo: Continue adding and write descriptions

CHAPTER 18

Communication

Communication in POP is abstract, multiple communication protocols can be used at the same time. Those protocols are abstracted by the `Combox` object and its companion, a 3 generic and 3 helper classes for a total of 6.

POP-Java plugin explains how to create a new user generated Combox while in this chapter we want to explain what's behind a standard Combox and what happen between a Client's Interface and the POP Object's Broker.

18.1 Combox architecture

18.2 Broker Interface architecture

18.3 POP Buffer

18.3.1 Standard Data Types

18.3.2 POP Data Types

18.3.3 Special Data Types

CHAPTER 19

Java Agent

???

CHAPTER 20

Configuration

POP-Java can be configured by placing a file in a specific location: `${POPJAVA_LOCATION}/etc/popjava.properties`.

There exists three level of configuration in POP-Java, the first level is POP Hardcoded values, the second is system level override in the location specified above, while the third level is user level override which enable the user to tweak POP even more locally. This can be seen in 1.

Note: For testing purpose the path `./etc/popjava.properties` is also valid.

20.1 Parameters available

`SYSTEM_JOBMANAGER_CONFIG : File`
`$POPJAVA_LOCATION/etc/jobmgr.conf` with `$POPJAVA_LOCATION` falling back to `./` if not set. The location where the Job Manager configuration file located.

`DEBUG : Booelan`
`false` print debug information to console.

`DEBUG_COMBOBOX : Booelan`
`false` print Combox debug information to console.

`RESERVE_TIMEOUT : Int`
`60000` milliseconds before the Job Manager free reserved registered resources.

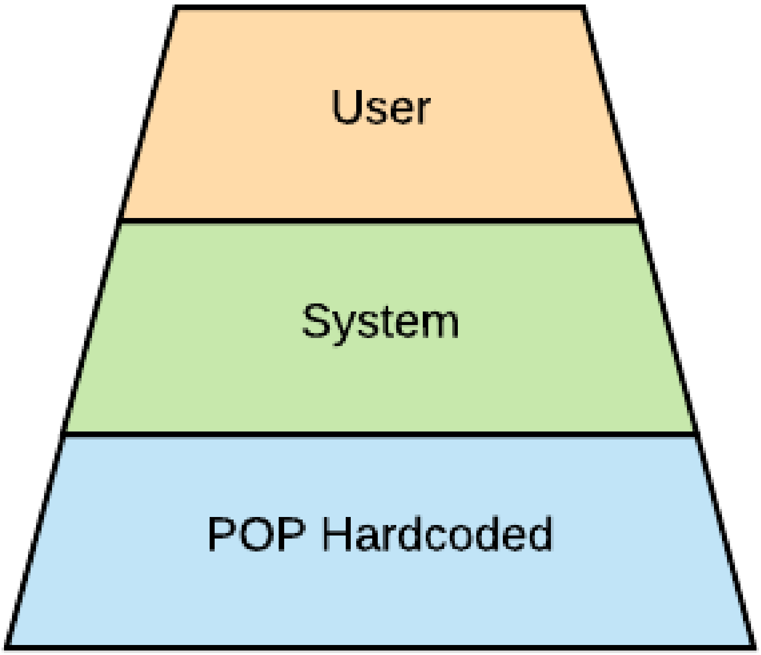


Fig. 1: POP-Java Configuration Layers

ALLOC_TIMEOUT : Int
`30000` milliseconds waiting for a connection to happen after a reservation.

CONNECTION_TIMEOUT : Int
`30000` milliseconds waiting before a connection exception is thrown.

JOBMANAGER_UPDATE_INTERVAL : Int
`10000` milliseconds waiting from interval to interval to check to free resources.

JOBMANAGER_SELF_REGISTER_INTERVAL : Int
`43200000` milliseconds (1/2 day) when the Job Manager register itself on its neighbors.

JOBMANAGER_DEFAULT_CONNECTOR : String
`jobmanager` which connector is used when none is specified.

JOBMANAGER_PROTOCOLS : String[]
`[ "socket" ]` protocols which are used for the Job Manager.

JOBMANAGER_PORTS : Int[]
`[ 2711 ]` ports which are used in combination with *JOBMANAGER_PROTOCOLS*.

JOBMANAGER_EXECUTION_BASE_DIRECTORY : File
`.` which directory should Job Manager use to start objects.

JOBMANAGER_EXECUTION_USER : String
`null` with which user should the Job Manager start objects as.

POP_JAVA_DEAMON_PORT : Int
`43424` the default port that the Java Daemon should use.

SEARCH_NODE_UNLOCK_TIMEOUT : Int
`10000` default time before unlocking the semaphore if no result was received.

SEARCH_NODE_SEARCH_TIMEOUT : Int
`0` default timeout for a Search Node research. `0` means that the first node responding will be used.

SEARCH_NODE_MAX_REQUESTS : Int
`Integer.MAX_VALUE` how many nodes should we visit before stopping. Unlimited by default.

SEARCH_NODE_EXPLORATION_QUEUE_SIZE : Int
`300` how many nodes should we remember before dropping them to save memory.

TFC_SEARCH_TIMEOUT : Int
`5000` minimum time to wait for TFC results are returned to the user. Similar to *SEARCH_NODE_SEARCH_TIMEOUT*.

DEFAULT_ENCODING : String
`xdr`

SELECTED_ENCODING : String
`raw`

DEFAULT_PROTOCOL : String
`socket` which protocol should we use when none is specified.

PROTOCOLS_WHITELIST : Set<String>
`[ ]` which protocols should be allowed to be used.

PROTOCOLS_BLACKLIST : Set<String>
`[ ]` which protocols should be blocked and not be used; also applied when using *PROTOCOLS_BLACKLIST*

ASYNC_CONSTRUCTOR : Booelan
`true`

ACTIVATE_JMX : Booelan
`false`

CONNECT_TO_POPCPP : Booelan
`false`

CONNECT_TO_JAVA_JOBMANAGER : Booelan
`true`

REDIRECT_OUTPUT_TO_ROOT : Booelan
`true`

USE_NATIVE_SSH_IF_POSSIBLE : Booelan
`true`

SSL_PROTOCOL_VERSION : String
`TLSv1.2`

SSL_KEY_STORE_FILE : File
`null` the file with the Key Store with the private key.

SSL_KEY_STORE_PASSWORD : String
`null` password for opening and checking the keystore.

SSL_KEY_STORE_PRIVATE_KEY_PASSWORD : String
`null` password to decrypt the private key in the keystore.

SSL_KEY_STORE_LOCAL_ALIAS : String
`null` alias of the private key and public certificate.

SSL_KEY_STORE_FORMAT : KeyStoreFormat.
`null`, format `JKS`, `PKCS12` (experimental).

20.2 New attribute

Adding a new attribute require the modification of the Configuration class, this is because we grant access to attributes via `get` and `set` methods. The process is done 4 steps.

1. Choose the name of the attribute and add it to the `Settable` enumerator.

```
private enum Settable {
    MY_NEW_ATTRIBUTE,
    ...
}
```

2. Add a class attribute which will be used to store the value.

```
private String myNewAttribute = "";
```

3. Create getter and setter methods.

```
public String getMyNewAttribute() {
    return myNewAttribute;
}
public void setMyNewAttribute(String value) {
    setUserProp(Settable.MY_NEW_ATTRIBUTE, value);
    myNewAttribute = value;
}
```

Note: Using `setUserProp` enable us to save only the changed information if the User call `store()`.

4. Add the parsing rules in `load`.

```
switch(keyEnum) {
    case MY_NEW_ATTRIBUTE: myNewAttribute = value; break;
    ...
}
```

20.3 Remarks

All Java version except Java 9, properties file are encoded with ISO-8859-1 which means that all character outside the first 256 byte will be encoded with its hexadecimal form `\uXXXX`. For this reason be on alert when using characters outside this charset manually. From Java 9 properties files are saved using UTF-8 so this problem shouldn't matter.

CHAPTER 21

Services

21.1 POP Java Job Manager

21.2 POP Java App Service

21.3 POP Java Daemon

CHAPTER 22

Annotations

22.1 Object Description

22.1.1 Add a new OD

CHAPTER 23

Testing

There are two types of tests, JUnit and environment dependant tests.

23.1 JUnit

This kind of tests are used to see if many expected behaviors don't changes over time.

This kind of tests can tricky because contrarily to Test Suite tests, all of the JUnit tests a

23.1.1 Create a new test

In the `junit` package in the POP-Java workspace look for an appropriate package or create a new one to host a new test.

Use the following template to start creating a test class. It's important that each unit test initialize and end the POP Environment, the methods marked with `@Before` and `@After` do exactly this. For further information in regards how JUnit works visit JUnit's documentation.

```
public class SomeTests {

    @Before
    public void initPOP() {
        POPSystem.initialize();
    }
```

(continues on next page)

(continued from previous page)

```
    @After
    public void endPOP() {
        POPSystem.end();
    }

    @Test
    public void myTest() {
        ...
        assertTrue(...)
    }
}
```

Note: As of now we are using JUnit4, when POP-Java will use Java 8 as a minimum platform we will probably upgrade.

After the test is written don't forget to add it to the Test Suite. For example

```
@Suite.SuiteClasses( { ..., SomeTests.class})
public class LocalTests
```

23.1.2 Peculiarities

The is one extra details we have to be on alert with writing JUnit tests, all POP Object apart from having the `@POPClass` annotation should also extends POPObject directly. Furthermore, all new POP Object create must use the `PopJava.newInstance` method since there is no Java Agent running in the JUnit tests.

```
@POPClass
class MyPOP extends POPObject {
    void MyPOP() { }
}

class MyTest {
    ... // before & after
    @Test
    public void test() {
        MyPOP my = PopJava.newInstance(MyPOP.class);
        ...
    }
}
```

23.2 Test Suite

The POP-Java Test Suite is a Shell Script with the objective of executing some small POP-Java program in a configured POP Environment.

Todo: continue

Bibliography

[Object Management Group01] *The Common Object Request Broker: Architecture and Specification - Version 2.6*. Object Management Group, Framingham, Massachusetts, December 2001.

[Grid and Ubiquitous Computing Group, EIA-FR10] *Parallel Object Programming C++, User and installation Manual*. Grid and Ubiquitous Computing Group, EIA-FR, Fribourg, Switzerland, 2010.

[ABB+02] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Meder, and S. Tuecke. *GridFTP Protocol Specification*. GGF GridFTP Working Group, September 2002.

[Cle10] Valentin Clément. Pop-java - technical report. Technical Report, EIA-FR, Fribourg, Switzerland, August 2010.

[CFK+88] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S.Tuecke. A resource management architecture for metacomputing systems. In *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, 62–82. 1988.

[FK98] I. Foster and N. Karonis. A grid-enabled mpi: message passing in heterogeneous distributed computing systems. In *Proc. 1998 SC Conference*. November 1998.

[FK97] I. Foster and C. Kesselman. Globus: a metacomputing infrastructure toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.

[FKNT02] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid services for distributed system integration. *Computer*, 2002.

[GFKH99] Andrew Grimshaw, Adam Ferrari, Fritz Knabe, and Marty Humphrey. Legion: an operating system for wide-area computing. *IEEE Computer*, 32(5):29–37, May 1999.

[KTIFoster03] N. Karonis, B. Toonen, and I.Foster. Mpich-g2: a grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 2003.

[KN07] P. Kuonen and T. A. Nguyen. Programming the grid with pop-c++. *Future Generation Computer Systems (FGCS)*, 23(1):23–30, January 2007.

[Ngu04] Thuan-Anh Nguyen. *An object-oriented model for adaptive high performance computing on the computational GRID*. PhD thesis, EPFL, Lausanne, Switzerland, 2004.

[RFG+00] A. Roy, I. Foster, W. Gropp, N. Karonis, V. Sander, and B. Toonen. Mpich-gq: quality-of-service for message passing programs. In *Proc. of the IEEE/ACM SC2000 Conference*. November 2000.

[SSA+01] H. Stockinger, A. Samar, B. Allcock, I. Foster, K. Holtman, and B. Tierney. File and object replication in data grids. In *10th IEEE Symposium on High Performance and Distributed Computing (HPDC2001)*. San Francisco, California, 2001.

[TDC03] Weiqin Tong, Jingbo Ding, and Lizhi Cai. A parallel programming environment on grid. In *International Conference on Computational Science 2003*, 225–234. 2003.

[WSF+03] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke. Security for grid services. In IEEE Press, editor, *Twelfth International Symposium on High Performance Distributed Computing (HPDC-12)*. 2003.

[WP00] Tiffani L. Williams and Rebecca J. Parsons. The heterogeneous bulk synchronous parallel model. Lecture Notes in Computer Science, 2000.

Index