Parallel Object Programming in C++
User and Installation Manual

The POP-C++ Team
Grid & Cloud Computing Group
http://gridgroup.hefr.ch
popcinfo@hefr.ch

Software Version : 2.5
Manual Version : 2.5-a

Hes·so // FRIBOURG FREIBURG
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences Western Switzerland

Parallel Object Programming in C++
User and Installation Manual
Manual version : 2.5-a, October 2012

This work was partially funded by the CoreGRID Network of Excellence, in the European Commission's 6th Framework Program and by the University of Applied Sciences of Western Switzerland of Fribourg.

The POP-C++ Team :

Pierre Kuonen
Tuan Anh Nguyen
Jean-François Roche
Valentin Clément
David Zanella
Marcelo Pasin
Laurent Winkler
Nicolas Brasey

Problems or questions concerning POP-C++ can be submitted on the POP-C++ web site or be sent by e-mail to:
`popcinfo@hefr.ch`

Table of content

CHAPTER

1 Overview

1 Introduction

Programming large heterogenous distributed computing environments such as GRID, P2P, Cloud or even large supercomputers is a challenging task. POP-C++ is an implementation, as an extension of the C++ programming language, of the POP (Parallel Object Programing) model first introduced by Dr. Tuan Anh Nguyen in his PhD thesis[1]. 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. to create parallel applications based on the MPMD (Multiple Programs Multiple Data) paradigm.

The POP-C++ programming language extends C++ by adding a new type of objects we call: **parallel object**. POP-C++ has been designed to ease, as much as possible, the development of efficient distributed applications using the powerful object-oriented programming paradigm and to allow existing C/C++ programs to be easily transformed in efficient parallel applications.

1.2 The POP Model

The POP model extends the traditional object oriented programming paradigm 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) to create the new concept of **parallel class**. By instantiating parallel classes we are able to create a new category of objects we will call **parallel objects** in this document.

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

- Distributed shareable objects

[1] Tuan-Anh Nguyen. An Object-oriented model for adaptive high performance computing on the computational Grid. PhD thesis, Swiss Federal Institute of Technology-Lausanne, 2004.

- Dynamic and transparent object allocation
- Various method invocation semantics

1.3 System Overview

Although the POP-C++ programming environment focuses on an object-oriented programming model, it also includes a runtime system which provides the necessary services to allow to run POP-C++ applications over distributed environments. An overview of the POP-C++ system architecture is illustrated in figure 1.1.

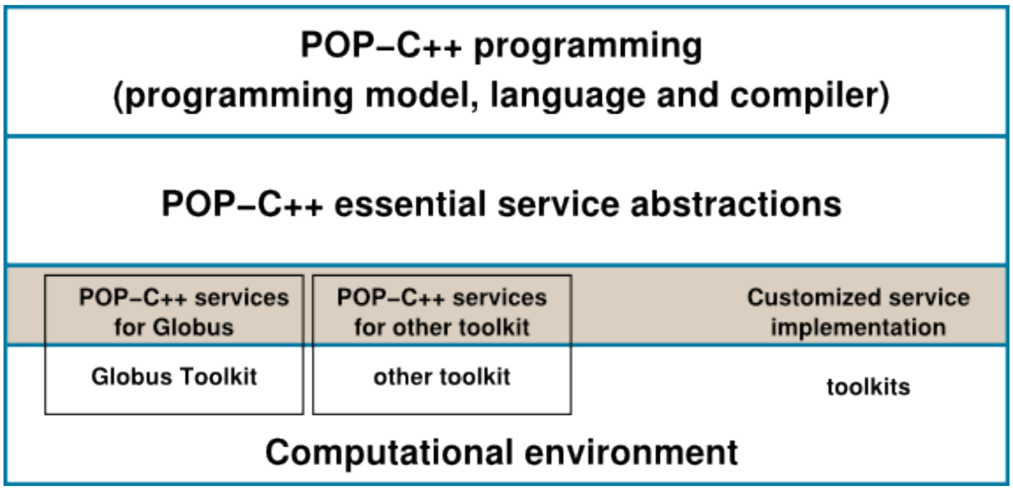

POP–C++ programming
(programming model, language and compiler)
POP–C++ essential service abstractions
POP–C++ services for Globus
POP–C++ services for other toolkit
Customized service implementation
Globus Toolkit
other toolkit
toolkits
Computational environment

Figure 1.1 - POP-C++ system architecture

The POP-C++ runtime system consists of three layers: the service layer, the 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.

1.4 Structure of this Manual

This manual has five chapters, including this introduction. The second chapter explains the POP programming model. The third chapter is the reference programming manual of POP-C++. It is the most important chapter. It describes the POP-C++ programming syntax and semantic. The fourth chapter explains how to compile and run POP-C++ applications. The fifth chapter shows how to compile and install the POP-C++ tool. Programmers interested in using POP-C++ should read first chapters 2, 3 and 4. System managers should read first chapter 5, and eventually chapters 2 and 4.

Beginners can also consult the **Getting started manual**, which is available on the POP-C++ website and show how to do a first simple installation and how to run a first simple POP-C++ program.

1.5 Additional Informations

More information can be found on the POP-C++ web site which contains :

- A quick tutorial to get started with POP-C++
- Solutions to commonly found problems
- Programming examples
- Latest sources available at :
  `http://gridgroup.hefr.ch/popc`

CHAPTER

2 The Parallel Object Model

2.1 Introduction

The POP programming model, is an general model which can be applied to many imperative object-oriented programming languages such as C++, Java, C#, Eiffel, etc. These programming languages provide high level abstractions for software engineering. 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 when executing an application. Nevertheless, two questions remain:

1. which objects should run remotely ?
2. where does each remote object lives ?

The answer, of course, depends 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 The Parallel Object Model

POP stands for **Parallel Object Programming**, and POP parallel objects are generalizations of traditional sequential objects. POP-C++ is an implementation of this model as an extension of C++. POP-C++ can instantiate parallel objects transparently and dynamically, assigning suitable resources to objects. POP-C++ 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 section 2.3.
- Syntactically, method invocations on parallel objects are identical to method invocations on traditional sequential objects. However, parallel objects support various method invocation semantics: **synchronous**, **asynchronous**, **sequential**, **mutex** and **concurrent**. These semantics are explained in section 2.4.

- 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 2.5.
- Each parallel object has the ability to dynamically **describe its resource requirement** when created. This feature is discussed in detail in section 2.6

As for traditional objects, parallel objects are active only when they execute a method (non active distributed object semantic). Communication between parallel objects are realized only through **remote methods invocations**.

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, figure 2.1 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` object.

To share the reference of a parallel object, POP-C++ allows parallel objects to be arbitrarily passed from one place to another as arguments of method invocations. See section 3.2.4 for more details on passing arguments.

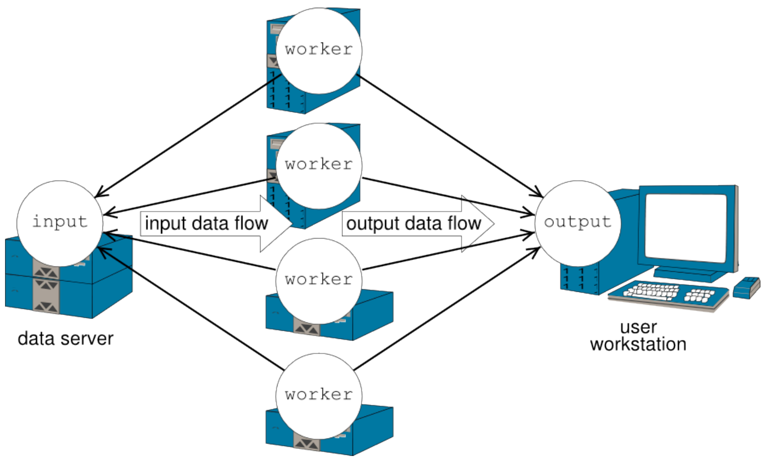


Figure 2.1 - A scenario using shared parallel objects

2.4 Methods Invocation 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.

- **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.
    - **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 figure 2.2). To learn how to pass this reference, see to section 3.2.4.

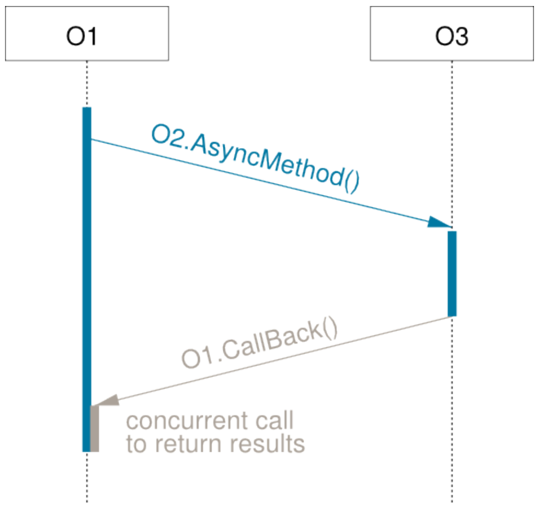


Figure 2.2 - Callback method returning value from an asynchronous call

- **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 on the object.
    - 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 later case he is executed after completion of all mutex calls previously arrived.

Figure 2.3 illustrates the 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 on-going execution of `Seq1()` (no order between concurrent and sequential invocations). When the mutex invocation `Mutex1()` arrives, it has to wait for previously called and running methods to finish. During this waiting, it also blocks other invocation requests arriving afterward (such as `Conc3()`) until the mutex invocation request completes its execution (atomicity and barrier).

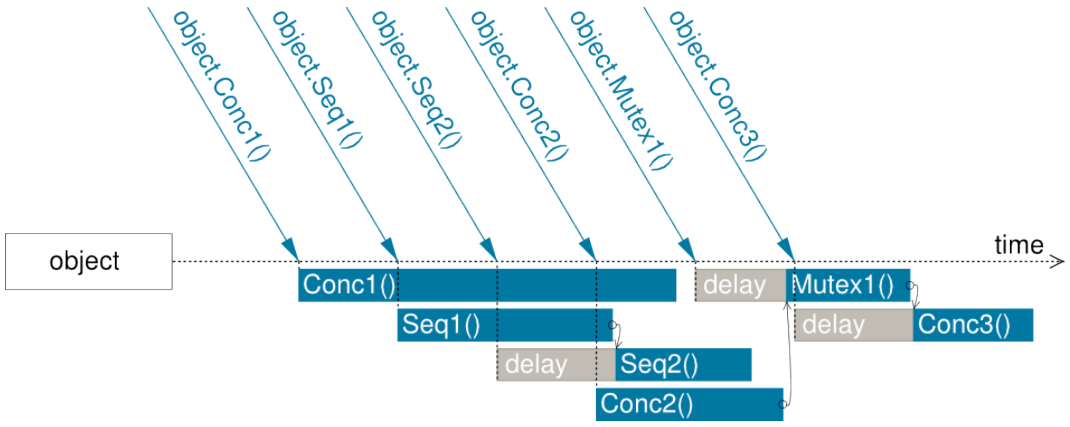


Figure 2.3 - Example of different invocation requests

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. This automatic features result in a dynamic usage of computational resources and gives to the applications the ability to adapt to changes in both the environment and the user behavior.

The creation of POP-C++ parallel objects is driven by high-level requirements on the resources where the object should lie (see section 2.6). If the programmer specifies these requirements they are taken into account by the runtime system for the transparent object allocation. The allocation process consists of three phases: first, using a **resource discovery** algorithm, the system finds a suitable resource where the object will lie, then the object code is transmitted and launched on that resource and, finally, the corresponding local interface is created to be connected to the remote object. All this process is fully transparent to the programmer.

2.6 Requirement-driven Parallel Objects

Parallel processing is increasingly being done using distributed systems, with a strong tendency towards web and global computing. Efficient extraction of high performance from highly heterogeneous and dynamic distributed environments is a challenging task. The POP model has been 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 enables objects to describe their

resource requirements.

The POP allows the programmer to integrate resource requirements into parallel objects under the form of high-level resource descriptions. Each parallel object can be associated with an **object description** that describe the characteristics of the resources needed to execute this object. Many different resources requirement can be expressed, such as:

- Resource name: we specify the exact name of the resource we want to use to run the object. In such a case there is no need to run the resource discovery algorithm.
- The minimum required computing power needed to run the object.
- The minimum amount of memory needed to run the object.
- The expected minimum communication bandwidth and maximum latency.
- ...

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

The evaluation of object descriptions are done at run time during the parallel object creation. The programmer can provide an object description with 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. The syntax of object descriptions is presented on section 3.2.9.

CHAPTER

3 User Programming Manual

3.1 Introduction

The POP programming model presented in the chapter 2 is a suitable programming model for large heterogenous 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, keeping the good properties of object oriented programming (data encapsulation, inheritance, polymorphism, ...) and add new properties.

The POP-C++ programming language is an extension of the C++ programming language implementing the POP model. Its syntax remains as close as possible to standard C++ so that C++ programmers can easily learn it and existing C/C++ programs and libraries can be parallelized without too much effort. Changing a sequential C/C++ application into a POP-C++ distributed application is rather straightforward on a syntactic point of view.

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 computing unit to execute the remote object, programmers can add object description information to each constructor of a parallel class (see sections 2.6 and 3.2.9). As presented in chapter 2, in order to create parallel execution, POP-C++ offers new semantics for method invocations. These new semantics are indicated thanks to five new keywords (see sections 3.2.3). Synchronizations between concurrent methods are sometimes necessary, as well as event handling. The standard POP-C++ library supplies some tools for that purpose (see section 3.4).

This chapter describes the syntax of the POP-C++ programming language and presents main tools available in the POP-C++ standard library.

A complete example of a POP-C++ program can be found in the **POP-C++ Getting Started** Manual available on the POP-C++ website.

3.2 Parallel Objects

Parallel objects are instances of parallel classes (see 3.2.1). Unless the term **sequential object** is explicitly specified, a parallel object is simply referred to as an **object** in the rest of this chapter.

3.2.1 Parallel Classes

As in C++ , developing POP-C++ programs mainly consists of designing and implementing classes. The declaration of a parallel class in POP-C++ is very similar to the declaration of standard (sequential) class in C++ but the declaration is introduced by the keyword **`parclass`** instead of the keyword `class`. Example:

```
parclass ExampleClass
{
     /* methods and attributes */
     ...
};
```

or

```
parclass ExampleClass: BaseClass1, BaseClass2
{
     /* methods and attributes */
     ...
};
```

As in the C++ language, multiple inheritance and polymorphism are supported in POP-C++ . A parallel class can be derived from other parallel classes (see section 3.2.6) and methods of a parallel class can be declared as overridable (`virtual` methods).

Parallel classes are very similar to standard C++ classes. Nevertheless, some restrictions applied:

- All data attributes must be `protected` or `private` (`public` is not authorized).
- Parallel objects must not access any global variable.
- Programmer-defined operators are not allowed.
- There are no methods that return memory address references (see section 3.2.5).
- A parallel class can only derived from other parallel classes.
- Class (`static`) attributes or methods are not allowed.

These restrictions are not major issues in the object-oriented programming paradigm. In some cases they can even improve the legibility and the clearness of programs. These restrictions can be mostly worked around using accessors (`get()` and `set()` methods) and by encapsulating global data and shared memory address variables into other parallel objects.

3.2.2 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. See section 3.4.3 to learn about the POP-C++ 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 more reference to this object. Nevertheless the programmer does not have to deal with this complex mechanism because the POP-C++ run-time automatically manages parallel objects' life time using an internal reference counter. A null counter value will provoke the object to be actually destroyed.

Syntactically, the creation and the destruction of a parallel object are identical to sequential object in C++. A parallel object can be implicitly created by declaring a variable of this parallel class or by using the standard C++ `new` operator. As for sequential objects in C++, parallel objects created using the `new` operator must be explicitly destroyed using the `delete` operator, when implicitly created parallel objects are automatically destroyed at the end of the block where the variable has been declared. Below is a example of declaration and creation of parallel objects:

```
parclass ExampleClass
{
  ExampleClass();  // default constructor;
  ...
};
...

ExampleClass x;     // create the parallel object x
ExampleClass y[5]; // create 5 parallel objects
ExampleClass* p;    // declare a pointer on a parallel object

p = new ExampleClass; // create the parallel object *p
```

3.2.3 Parallel Class Method Invocations

Like sequential classes, parallel classes contain methods and attributes. Method can be `public`, or `private` while attribute must be either `protected` or `private`. Public attributes are not allowed because it leads to share memory which is in contradiction with the distributed computing paradigm where memory is distributed (different memory address space). For each method, the programmer can define the invocation semantics by placing appropriated keywords, before methods declaration. These semantics, described in section 2.4, are specified using the following keywords:

- Synchronous/Asynchronous (interface semantic):
  - **`sync`**: Synchronous invocation. This is the default value.
  - **`async`**: Asynchronous invocation.
- Sequential/Concurrent/Mutex (object-side semantic):
  - **`seq`**: Sequential invocation. This is the default value.
  - **`mutex`**: Mutex invocation.
  - **`conc`**: Concurrent invocation.

The combination of both semantics defines the overall semantics of a method (six different combinations). For instance, the following declaration defines an synchronous concurrent method that returns an integer and belong to the parallel class `ExampleClass`:

```
parclass ExampleClass
{
  ExampleClass();  // default constructor;
  sync conc int myMethod(); // synchronous concurrent method
  ...
};
```

3.2.4 Passing arguments

***Note**: In this document the words '**argument**' and '**parameter**' have the same meaning. To distinguish between arguments (or parameters) which appear in the definition of the methods and arguments (or parameters) which are effectively used when calling methods, we use the terms **formal arguments** and **effective arguments** respectively.*

When calling methods of parallel objects (remote methods), the effective arguments of the call must be transferred to the object being called (the same happens for returned values). We have to distinguish between two different situations:

1) the argument is a parallel class
2) the argument is NOT a parallel class

These two situations are presented separately in the next two paragraphs.

The argument is not a parallel class

In such case the argument can be any basic or structured type of C++ or a sequential class. It has to be noted that **input arguments** are only transferred from the caller object to the callee object, when, in case of a synchronous method invocation, **output arguments** are only transferred back to the caller. **Input/output arguments** are, of course, transferred in both direction. Because the caller and the callee do not use the same memory address space, some rules applied for arguments of methods of parallel classes:

- if the method is asynchronous, arguments must be input-only.
  - arguments cannot be pointers (`*`) or passed by reference (`&`)
  - the returned type of the method must be `void`
- if the method is synchronous:
  - constant and passing-by-value arguments are input-only.
  - arguments passed by reference (`&`) are considered as both input and output.
  - the returned value (`return`) is output-only.
  - pointer arguments are normally not authorized (see section 3.2.5 for exception).

Programmers can optionally modify these rules to help the POP-C++ compiler to generate more efficient code by specifying which arguments have to be transferred and returned. This is done by using specific POP-C++ directives in the argument information block. The directives **`in`** (for input), **`out`** (for output), or both indicate if the argument is input-only, output-only or both respectively. The optional argument information block should appear between braces (`[` and `]`), right before each argument declaration (see figure 3.2).

When the caller and the callee run on different processor's architectures, data must be **encoded to a standard format** before being transferred and decoded at reception. The programmer can inform the compiler of the standard format to use for the encoding (see `od.encoding(...)` in section 3.2.9). By default data are not encoded, i.e. raw values are transferred.

Simple arguments are single arguments of basic C++ type such as `int`, `float`, `double`, `bool` etc. or structures that only contain basic types (no pointer, no array, no class). Simple arguments can be transferred without other precaution than a possible encoding. Figure 3.2 shows an example of methods having only simple arguments. In this example the `out` directive is used to indicate to the compiler that the second argument of the method `get` does not need to be sent to the method but only returned.

Argument of complex types such as arrays, pointers or sequential class argument must be serialized (or **marshaled**) prior to be sent (or returned) to (from) remote objects. How to marshal/demarshal complex argument is presented in the section 3.2.5.

```
parclass Toto
{
  struct data
  {
    int i;
    float f;
  }

  async seq void set(int i);
  sync  seq  int get([out] struct data &s);
  ...
};

/* main program */
int main(int argc, char* argv[])
{
  struct data a;
  Toto x;
  x.set(1);
  int res = x.get(a);
  printf("Value are %d, %d, %f\n", res, a.i, a.f);
  ...
}
```

Figure 3.2 - Simple arguments

The argument is a parallel class

Only **references** (using the `&` modifier) **to parallel classes can be passed as parameter** to any methods (method of parallel or sequential classes). This restriction directly derived from the semantic of parallel classes. Indeed, passing a parallel object by value means to create a copy of this parallel object. We want to avoid such a copy because, on one hand, it is usually not what the programmer wants and, on another hand, we should find a new computing resource for this new parallel object. Passing parallel objects by reference allows several objects (parallel or not) to share the reference of the passed parallel object. This is the way POP-C++ implements the concept of shareable parallel objects.

3.2.5 Marshalling complex arguments

When an argument of a method of a parallel class is not a simple argument (see section 3.2.4) and is not a parallel object it must be marshaled (or serialized) at the sender side and demarshaled (or deserialized) at the receiver side. The programmer must indicate to the POP-C++ run-time how to marshal and demarshal it. This must be done in two cases

- the argument is an instance of a sequential class;
- the argument is a pointer (this includes the case of arrays).

When the argument is an instance of sequential classes this **arguments must be an instance of a class which derive from the `POPBase` sequential class provided by the standard POP-C++ library**. The interface of the `POPBase` class is the following:

```
class POPBase
{
  public:
  virtual void Serialize(POPBuffer &buf, bool pack);
  ...
};
```

Programmer must implement, in the derived sequential class, the `Serialize` method which will be used by the POP-C++ run-time to marshal and demarshal effective parameters which are instances of this class.

As shown above, the method `Serialize` requires two arguments:

- the `buf` argument that stores the marshaled data object;
- the flag `pack` which specifies if the method `Serialize` is called to marshal or to demarshal the data into or from the buffer `buf`.

The `POPBuffer` class available in the POP-C++ standard library provides a set of `Pack`/`UnPack` methods for all basic C++ types (`char`, `bool`, `int`, `float`, ...). `Pack` is used to marshal the data into `buf` and `UnPack` is used to demarshal the data from `buf`.

Below is the declaration of the `POPBuffer` class:

```
class POPBuffer
{
  public:
    // Below, Type refers to any basic C++ type
    void Pack(const Type *data, int n);
    void UnPack(Type *data, int n);
};
```

For the method `Pack`, the parameter `data` contains the address of the data to marshal into `buf` when, in the case of the `UnPack` method, the same parameter (`data`) contains the address of the data into which the data received in `buf` must be demarshaled. `Pack/Unpack` methods offer the possibility to marshal/demarshal several data in one call. The `n` parameter contains the number of data of type `Type` to marshal/demarshal. This feature is especially useful when passing arrays of objects.

Figure 3.3 shows an example of marshalling/demarshalling of the `Speed` sequential class.

```
class Speed: public POPBase {
     public:
          Speed();
          virtual void Serialize(POPBuffer &buf, bool pack);
          float *val;
          int count;
};

void Speed::Serialize(POPBuffer &buf, bool pack) {
     if (pack) {
          buf.Pack(&count,1);
          buf.Pack(val, count);
     }
     else {
          if (val!=NULL) delete [] val;
          buf.UnPack(&count,1);
          if (count>0) {
               val=new float[count];
               buf.UnPack(val, count);
          }
     else val=NULL;
     }
}

parclass Engine {
     ...
     void accelerate(const Speed &data);
     ...
};
```

Figure 3.3 - Marshalling/demarshalling of sequential class argument

The best way to transfer complex information to a method of a parallel class is to use sequential classes. This is the most “object-oriented” approach. Nevertheless, in some cases for reasons of simplicity, arrays arguments (which are pointers) are used. In such a case it is not possible to use the `Serialize` method as explained in the previous section because the parameter does derive from the `POPBase` class.

Data pointers in C++ are ambiguous (pointer to data or starting address of an array ?). Therefore, programmers have to explicitly supply the number of elements pointed by that data pointer using the special POP-C++ directive `size` in the argument information block. Programmers can also indicate which specific function must be used for marshalling/ demarshalling the argument using the `proc` directive. The `size` and `proc` directives must be inserted in the argument information block together with the `in` or `out` directives (if any). Be aware that `void` pointers (`void *`) cannot be used as arguments of parallel object methods.

```
      parclass Table
      {
        ...
        void sort([in, out, size=n] int *data, int n);
      ...
      };
      /* main program */
      ...
      Table sales;
      int amount[10];
      sales.sort(amount, 10);
      ...
```

Figure 3.4 - Marshalling/demarshalling of an array argument

Figure 3.4 contains an example of a method `sort()` that has two arguments: an array of integer data (for input and output) and its size (the `n` parameter).

```
      struct Speed
      {
        float *val;
        int count;
      };

      void marsh(POPBuffer &buffer, Speed &data, int count,
                 int flag, POPMemSpool *tmpmem)
      {
        if (flag & FLAG_MARSHAL)
        {
          buffer.Pack(&data.count,1);
          buffer.Pack(data.val, data.count);
        }
        else
        {
          buffer.UnPack(&data.count,1);
          // performing temporary allocation before calling UnPack
          data.val=
              (float *)tmpmem->Alloc(data.count*sizeof(float));
          buffer.UnPack(data.val, data.count);
        }
      }

      ....

      parclass Engine
      {
        ...
        async seq void accelerate([proc=marsh] const Speed &data);
        ...
      };
```

Figure 3.5 - Marshalling/demarshalling of complex `struct` *argument*

Figure 3.5 illustrates a more complex example of usage of the `proc` directive. In this example the programmer needs to allocate temporary memory space before serving the invocation request. This memory space will be automatically freed by the system after the invocation finished if the method `Alloc` provided by the class `POPMemSpool` of the POP-C++ environment is used to do this temporary memory allocation. The declaration of the `Alloc` method of the `POPMemSpool` class is given below:

```
class POPMemSpool
{
  public:
    void* Alloc(int size);
};
```

3.2.6 Inheritance in POP-C++

Inheritance is fully supported in POP-C++. Nevertheless it is a little more tricky than in standard C++, because there are two different types of classes: the parallel and the sequential (C++ standard) classes. The following general rule applies in POP-C++:

Parallel classes can only inherit from parallel classes and sequential class can only inherit from sequential classes.

Inheritance between sequential classes is strictly identical to inheritance in pure C++. On a syntactic point of view, inheritance between parallel classes is identical to inheritance between sequential classes. Nevertheless, due to the way parallel classes behave, some specific knowledge is required to correctly use inheritance between parallel classes. This knowledge mainly concerns the usage of the `@pack` directive (see section 3.3.1) and the way the compilation of parallel classes with inheritance must be done (see section 4.3).

3.2.7 Usage of `this` in POP-C++

In C++ the identifier `this` provides a pointer to the current object. It can be used to access a method of the current object. Example:

```
class toto
{
  public:
    void aMethod();
    void anotherMethod();
  ...
}

void toto::anotherMethod()
{
  ...
  this->aMethod();  // call ‘aMethod’ on the current object
  ...
}
```

In this example the instruction:

```
this->aMethod();
```

is strictly equivalent to the instruction:

```
aMethod();
```

In both cases `aMethod` is called on the current object.

Another usage of `this` is to pass a pointer on the current object to another method (usually a method of another object). Example:

```
class titi;
class toto
{
  public:
   void setTiti(titi* t);
  ...
  private:
    titi* x;
  ...
}

class titi
{
  public:
    void aMethod();
  ...
}

void toto::setTiti(titi* t)
{
   x=t;
}
void titi::aMethod()
{
  toto t;
  t.setTiti(this);  // pass a pointer to the current object
}
```

In POP-C++ the usage of `this` is a little bit more tricky. We have to consider two cases:

- `this` is a pointer to an instance of a sequential class
  - In this case the behavior is exactly the same than in standard C++
- `this` is a pointer to an instance of a parallel class
  - In this case the instructions:
    ```
    this->aMethod();
    ```
    and:
    ```
    aMethod();
    ```
    are not anymore strictly equivalent.

Indeed, the instruction `this->aMethod();` calls the method `aMethod` taking into account the semantic of `aMethod()` (`sync`, `async`, `seq`, etc). In such a case you must take care because you can easily cause deadlocks. It is the case, for example, if the method which makes the call and the called method have both the `seq` semantic.

The instruction `aMethod();` (without the `this`), does **not** take into account the semantic of the method, because the call is considered as an internal call to the method and is handled as a normal C++ method call.

> ***Note:*** *Usage of* `this` *in parallel classes is complex because the detection of possible deadlocks can be very difficult. It is a good programming practice to limit, as much as possible, usage of* `this` *in parallel classes.*

When pointing toward a parallel class, `this` can be use as effective parameter for a method call. Nevertheless the POP-C++ limitation which imposes that pointers arguments are not authorized applies. In addition as parallel object can only be passed by reference (see section 3.2.4), only `&(*this)` can be really passed as an effective argument.

3.2.8 Standard output in POP-C++

To write on the standard output (`stdout`), POP-C++ programs can use the standard C function `printf` and the standard C++ `cout` operator.

The `printf` function behaves exactly as in traditional C++ programs. In particular, if a `printf` is executed in a remotely executed parallel object, the POP-C++ run-time routes the text to print on the computing resource which run the `main` program, i.e. the local computer. Please notes that the text is effectually routed to the local computer only when the end-of-line delimiter (‘`\n`’) is printed.

Usage of the C++ `cout` operator is also possible with POP-C++. Nevertheless, due to the fact that the text has to be routed to the local computer (that one which run the `main`), one must use a specific POP-C++ end-of-line delimiter instead of the standard C++ `endl` operator. Failing to do so will prevent the message to be printed. The specific delimiter to use is: `popcendl`.

Below is an example of usage of the `cout` operator in POP-C++:

```
cout << "A message to print by a parallel object" << popcendl
```

3.2.9 Object Descriptions

Object descriptions have been introduced in section 2.6 and are used to describe the resource requirements for the execution of a given object. Object descriptions are declared along with parallel object constructor statements. Each constructor of a parallel object can be associated with an object description that resides directly after the argument declaration and before the instruction terminator operator ‘`;`’. The syntax of an object descriptor the following:

```
@{expressions}
```

An object description contains a set of resource requirement expressions. All resource requirement expressions are separated by semicolons and can be any of the following:

```
od.resNum(exact);
od.resNum(desired, lbound);
od.resString(resource);
resNum := power | memory | network
resString := protocol | encoding | url
```

exact, desired and *lbound* terms are numeric expressions, and *resource* is a null-terminated string expression of type POPString (see section 3.4.1). The semantics of those expressions depend on the resource requirement specifier (the keyword corresponding to resNum or resString). The *lbound* term is only used in non-strict object descriptions, to specify the strict lower bound of the acceptable resource requirements when desired correspond to desired value. In the resource string, the different options must be separate with one blank. The priority of options is position dependent, the first option having the highest priority and the last one having the lowest.

The current implementation allows indicating resources requirement in terms of:

- Computing power (in Mflops), keyword power
- Memory size (in MB), keyword memory
- Bandwidth (in Mb/s), keyword network
- Location (host name or IP address), keyword url
- Protocol (socket or http), keyword protocol
- Data encoding (raw, xdr, raw-zlib or xdr-zlib), keyword encoding

An example of parallel class declaration is given in the figure 3.1. In this example, the constructor for the parallel object Bird requires a computing power of P Mflops, a desired memory space of 100MB (having 60MB is acceptable) and the communication protocol is socket or HTTP (socket having higher priority). There is an implicit “and” operator between the different od.

```
parclass Bird {
  public:
    Bird(float P) @{  od.power(P);
                       od.memory(100,60);
                       od.protocol("socket http"); };
    ...
};
```

Figure 3.1 - Object descriptor example

Object descriptors are used by the POP-C++ runtime system to find a suitable resource for the parallel object with the exception of the od.url object descriptor which directly specifies the name of the resource to use. If no suitable resource is found to execute the objet then an exception is raised (see section 3.4.3). If no object description is indicated, the POP-C++ run time will use an unspecified internal procedure to decide where to instantiate this object.

3.3 Object Layout

3.3.1 The @pack() directive

The POP-C++ compilation process generates several executable files (see section 4.1). One of them is the main program file which is used to start the application. Other executable files contain the implementations of the parallel classes for a specific platform. A given executable file can store the implementation of one or several parallel classes. Programmers must indicate to the POP-C++ compiler which parallel classes to store in which executable file. This is done by inserting the `@pack()` directive in the source file of the implementation of the parallel classes (the `.cc` file).

```
        Stack::Stack(...) {
                ...
        }
        Stack::push(...) {
                ...
        }
        Stack::pop(...) {
                ...
        }
        @pack(Stack, Queue, List);
```

Figure 3.4 - Packing objects into an executable file

Figure 3.4 shows a fragment of the implementation of a `Stack` parallel class. At the end of this file we have inserted a `@pack()` directive which indicates to the compiler that it must store the executable code of classes `Stack`, `Queue`, and `List` in the same executable file.

This does not prevent the programmer to put, if desired, the **source code** of the parallel classes `Stack`, `Queue` and `List` in separated source code files.

The rule to apply is:

For a given parallel class, among the source files passed to the POP-C++ compiler, exactly one source file must contain the `@pack()` *directive for this parallel class.*

Usually one put each parallel class in a separate executable file. As a consequence the source file of the implementation (the `.cc` file) of a parallel class is usually terminated by the `@pack(ParClassName);` directive.

There are some specific cases where it is required to put several parallel classes in the same executable files. For example when a parallel class inherits from other parallel classes. In such a situation, mother and child classes must be put in the same executable file. Failing to do so will lead to the following error at execution time:

```
        Out of resource (errno 10001)
```

which indicates that the POP-C++ run-time is not able to find the needed resources, namely the executable code of the parallel classes (usually the mother class), to execute the object.

Below is an example of parallel classes inheritance in POP-C++. In this example we have a parallel class `Child` which inherits from a parallel class `Mother`. The file `mother.ph`, `mother.cc`, `child.ph` and `child.cc` contain respectively the declaration and the implementation of the parallel class `Mother`, the declaration and the implementation of the parallel class `Child`.

File: `mother.ph` (fragment)

```
parclass Mother
{
  public:
    Mother();
    .....
  protected:
    ......;
};
```

File: `child.ph` (fragment)

```
parclass Child : public Mother  // Inherits from Mother
{
  public:
    Child();
    ......
};
```

File `child .cc` (fragment)

```
Child::Child() {....;}
.....
@pack(Child, Mother);  // Classes Mother and Child will
                       // be in the same executable file
```

The file `mother.cc` must not contain a `pack` directive as a `pack` directive already exists for this class in the file `child.cc`.

3.3.2 Class Unique Identifier

For a given program, the C++ compiler assigns a unique class identifier to each class of the program. As the POP-C++ compiler generates several C++ programs from a unique POP-C++ program there is a little risk that the same identifier is assigned to several different classes residing in different C++ executables but belonging to the same POP-C++ program. This will cause a program crash because the POP-C++ run-time use class unique identifiers to identify classes in side a program. To avoid this problem POP-C++ provides to programmers the possibility to manually assign unique class identifiers to parallel classes.

This is done using the `classuid` function as shown below:

```
parclass ExampleClass {
     ...
     public:
         classuid(1001);
         ...
}
```

We recommend to use values for `classuid` greater that 1000.

Note: *Even is the problem mentioned above can, theoretically, occur, the POP-C++ team never reported it. However it is a good programming practice to indicate* `classuid`. *The POP-C++ compiler shows a warning message when no* `classuid` *is specified for a parallel class.*

3.4 POP-C++ standard library

The POP-C++ standard library offers classes and functions which can be useful or even necessary to write complex POP-C++ programs. This library in automatically include therefore no `#include` directive is required. This library are described in this section.

3.4.1 The `POPString` class

The class `string` is an often used class in C++ programs. Used "as is" the `string` class cannot be marshaled/demarshaled because it does not derived from `POPBase`. To overcome this difficulty the POP-C++ library provides the `POPString` class. As for basic C++ types, instances of `POPString` class are automatically marshaled/demarshaled by POP-C++. This class can be used to pass string argument to methods of parallel classes. It is designed to ease as much as possible conversion from `string` or `char*` to `POPString` and the reverse. Methods of the `POPString` class are shown in the figure 3.5.

```
class POPString
{
  public:
    // Constructors
    POPString();
    POPString(const char *x);
    POPString(const char *x, int n);
    POPString(std::string x);
    POPString(const POPString &x);

    // Destructor
    ~POPString();

    // Casting
    operator const char *() const;
    operator std::string () const;

    // Extracts a substring
    void substring(int start, int end, POPString &sub);

    // Get length of POPString
    int Length() const;

    // Returns a pointer to the (char*) data
    char *GetString();
};
```

Figure 3.5 - The POPString class

3.4.2 Synchronization and locks

POP-C++ provides the concurrent semantic (`conc`) for method invocations (see section 2.4). As a consequence we can have several methods which are concurrently executed inside the same parallel object. If these methods try to access the same data (attribute of the object) this can lead to race conditions and can require a way to synchronize the access to the share data. This is a standard problem in concurrent systems. POP-C++ provides a standard solution to this problem thanks to the `POPSynchronizer` class.

Figure 3.6 shows the declaration of the `POPSynchronizer` class

```
class POPSynchronizer
{
  public:
    POPSynchronizer();
    lock();
    unlock();
    raise();
    wait();
};
```

Figure 3.6 - The POPSynchronizer class

A synchronizer is an object used for general synchronization of concurrent execution inside a parallel object. Every synchronizer can handle a **lock** and an **event**. Locks and events can be used independently of each other or not.

Calls to `lock()` close the lock and calls to `unlock()` open the lock. A call to `lock()` returns immediately if the lock is not closed by any other method. Otherwise, it will pause the execution of the calling method until another method releases the lock. Calls to `unlock()` will reactivate one (and just one) paused call to `lock()`. The reactivated method will then succeed closing the lock and the call to `lock()` will eventually return. When creating a synchronizer, by default the lock is open. A special constructor is provided to create synchronizer with the lock already closed.

Figure 3.7 shows an example of usage of locks with the `POPSynchronizer` class.

```
parclass Example1
{
  private:
    POPSynchronizer syn;
    int counter;

  public:
    sync conc int getNext()
    {
      syn.lock();
      int r = ++ counter;
      syn.unlock;
      return r;
    }
};
```

Figure 3.7 - Using locks with the `POPSynchronizer` class

```
parclass Example2
{
  private:
    int cakeCount;
    boolean proceed;
    Synchronizer syn;
  public:
    void produce(int count)
    {
      cakeCount = count;
      syn.lock();
      proceed = true;
      syn.raise();
      syn.unlock();
    }
    void consume()
    {
      syn.lock();
      if (!proceed) syn.wait();
      syn.unlock();
     /* can use cakeCount from now on... */
    }
};
```

Figure 3.8 - Using event with the `POPSynchronizer` class

Events can be waited and raised. Calls to `wait()` cause the calling thread to pause its execution until another method triggers the event by calling `raise()`. If the waiting method possesses the lock, it will automatically release the lock before waiting for the event. When the even occurs (is raised), the waiting method will try to re-acquire the lock that it has previously released before returning control to the caller.

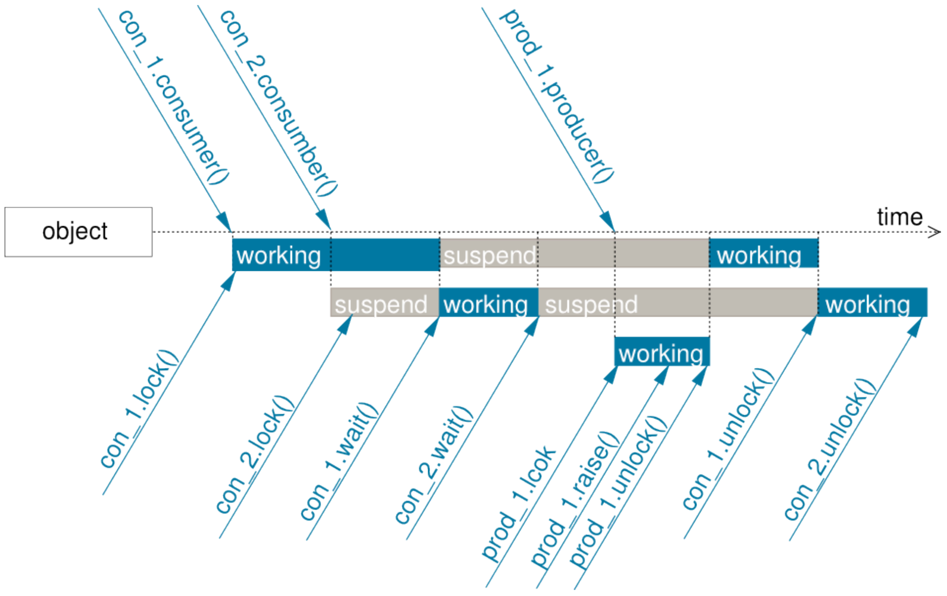


Figure 3.9 - Example with one producer and two consumers using the parallel class of figure 3.8

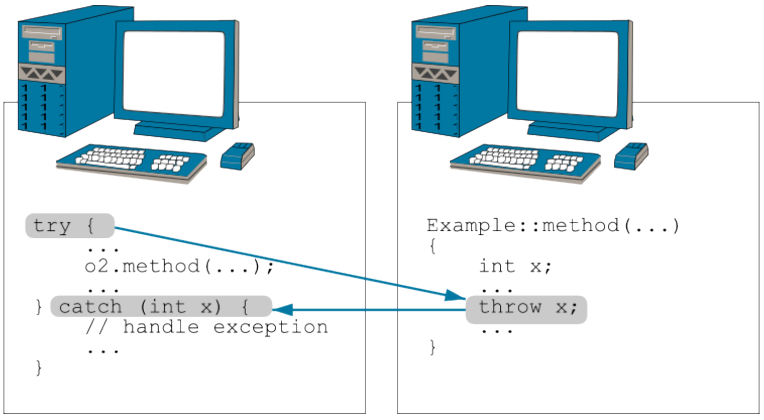
Many methods can wait for the same event. When a method calls `raise()`, all waiting-for-event methods are reactivated at once. If the lock was closed when the `wait()` was called, the reactivated methods will close the lock again before returning from the `wait()` call. If other methods calls `wait()` with the lock closed, all will wait the lock to be re-open before they are actually reactivated.

The typical use of locks is to implement critical sections when several methods can modify, at the same time, a shared attribute. One of the most typical usage of events is to synchronize the producer-consumer situation. Figure 3.8 presents an example of usage of event with the `POPSynchronizer` class and figure 3.9 shows an example of usage of this example.

3.4.3 Exceptions

Exceptions are a powerful way provided by C++ to handle errors. Exceptions allow the programmer to filter errors trough several calling stacks. When an error is detected inside a method, an exception can be thrown and can be caught somewhere else in the calling stack.

The implementation of exceptions in non-distributed applications, where all components run within the same memory address space is rather straightforward. In distributed environments where each component is executed in a separate memory address space (and data could be represented differently due to heterogeneity), the propagation of exceptions back to a remote caller is much more complex. In addition as POP-C++ supports asynchronous calls when a exception is thrown in an asynchronous method, the caller can be out of the context where it can catch this exception. For all these reasons, exceptions handling in POP-C++ is slightly different than in pure C++ programs.



```
try {
    ...
    o2.method(...);
    ...
} catch (int x) {
    // handle exception
    ...
}
```

```
Example::method(...)
{
    int x;
    ...
    throw x;
    ...
}
```

Figure 3.10 - Exception handling example

POP-C++ supports transparent exceptions propagation. Exceptions thrown in a parallel object will be automatically propagated back to the remote caller only when the exception is thrown in a **synchronous** method. In addition, the current POP-C++ version allows the following types of exceptions:

- Scalar data (`int`, `float`, etc.)
- Parallel objects
- Objects of class `POPException` (provided by the standard POP-C++ library)

All other C++ exception types (`struct`, `class`, ...) will be converted to `POPException` with the `UNKNOWN` exception code.

If the exception is thrown in an **asynchronous** method, the exception is not transferred to the caller but directly to the POP-C++ run-time which will cleanly abort the program. The drawback of this approach is that the programmer cannot catch exceptions thrown in asynchronous methods. This is especially penalizing when an exception derived from the `std::exception` C++ class has been defined by the programmer and a message has been associated with this exception. As the exception cannot be caught by the caller and, as the exception is transformed to `POPException` type, this message is lost and will never be displayed. To overcome this problem, when a exception derived from `std::exception` is thrown in a remote method (in both cases, asynchronous and synchronous methods), the POP-C++ run-time behaves in the following way:

Before giving back the control to the remote caller or to the POP-C++ run-time, the following message is displayed on `stdout`:

```
POP-C++ Warning: Exception 'TexteOfException' raised in
method 'NameOfMethod' of class 'NameOfClass'
```

Besides the exceptions defined by programmers, POP-C++ uses exceptions of type `POPException` to notify the user about the following system failure:

- Parallel object creation fails. It can happen due to the unavailability of suitable resources, an internal error on POP-C++ services, or the failures on executing the corresponding object code.
- Parallel object method invocation fails. This can be due to the network failure, the remote resource down, or any other causes.

The interface of `POPException` is the following:

```
class POPException
{
  public:
    const POPString Extra()const;
    int Code()const;
    void Print()const;
};
```

The `Code()` method returns the corresponding error code of the exception. The `Extra()` method returns a `POPString` associated with the exception.Finally the `Print()` method prints a text describing the exception.

All exceptions that are instances of parallel objects are propagated by reference. Other exceptions are transmitted to the caller by value.

3.5 Limitations

There are several limitations to the current implementation of POP-C++. Some of these restrictions are expected to disappear in the future while others are simply due to the nature of parallel programming and the impossibility for parallel objects to share a common memory. Some of these limitations have already been presented but a summary of these limitations for the current version (v2.5), is listed below:

- A parallel class cannot contain public attributes.
- A parallel class cannot contain class attributes or methods (`static`).
- A parallel class cannot be template.
- A parallel class cannot contain programmer-defined operators.
- An asynchronous method cannot return a value and cannot have reference or output parameters.
- Parallel objects can only be passed by reference.
- Global variables exist only in the scope of parallel objects (`@pack()` scope).
- A parallel object method cannot return a memory address (exception see fig. 3.4).
- Sequential classes used as parameter must derived from `POPBase` and the programmer must implement the `Serialize` method.
- Only scalar, parallel object and `POPException` exception types are propagated "as is". All other exceptions are converted to `POPException` with the unknown code.
- Exceptions raised in an asynchronous method are not propagated to caller. They abort (cleanly) the application.
- The specific POP-C++ end-of-line delimiter `popcendl` must be used with the `cout` operator (instead of the standard `endl` delimiter)

CHAPTER

4 Compiling and Running

4.1 The POP-C++ compilation process

The POP-C++ compiling process generates a main executable file and several object executables files. The main executable file provides a starting point for launching the application and object executables files are loaded and started by the POP-C++ runtime system whenever a parallel object is created.

The compilation process is illustrated in figure 4.1.

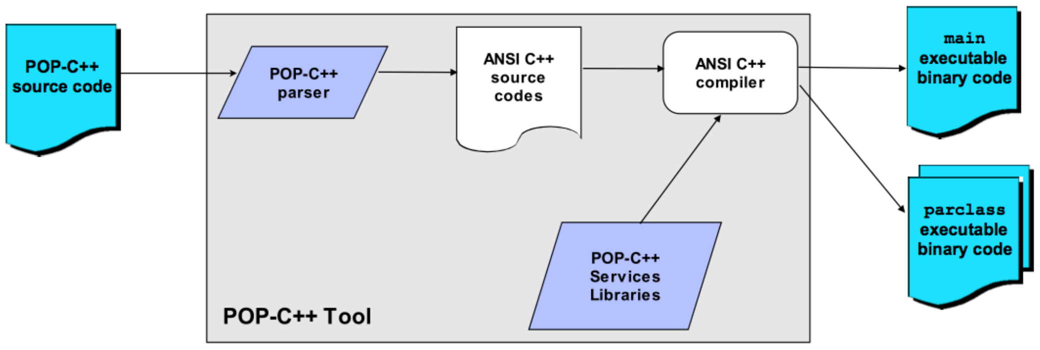

POP-C++ source code
POP-C++ parser
ANSI C++ source codes
ANSI C++ compiler
main executable binary code
parclass executable binary code
POP-C++ Services Libraries
POP-C++ Tool

Figure 4.1 - POP-C++ compilation process

The POP-C++ compiler contains a parser which translates the POP-C++ source code files into pure ANSI C++ source code files. Service libraries provide APIs that manages communication, resource discovery, object allocation, etc. At the end of the compiling, an ANSI C++ compiler generates binary executables files.

4.2 Compiling a simple POP-C++ program

In this section we illustrate the POP-C++ compiling process by describing how to compile a simple POP-C++ program. We assume that our simple POP-C++ program is constituted of three source files:

- `main.cc`: contains the main program

- `myclass.ph` : contains the declaration (the header) of the parallel class `myClass`
- `myclass.cc` : contains the body (the implementation) of the parallel class `myClass`

It has to be noted that, by convention, header files of parallel classes have the `.ph` extension instead of `.h` as for usual classes in C++.

4.2.1 Compiling

We have to generate several executables: one for the main program (`main`) and one for each parallel class declared in the program which, by convention, have the extension `.obj`.

POP-C++ provides the command `popcc` to compile POP-C++ source code. To compile the main program we use the following command:

```
popcc -o main myclass.ph myclass.cc main.cc
```

Note that we have to explicitly compile the declaration of the parallel class (the file `myclass.ph`) when in C++ we usually do not compile the header files (`.h` files). This is a specificity of the POP-C++ compiling process. The `-o` option has the same meaning than for standard C++ compilers (introduce the name of the executable file). It has to be noted that all options available with the used C++ compiler are also available with the `popcc` compiler as these options are directly transmitted to the C++ compiler which will generate the final executable files (see section 4.1).

4.2.2 Compiling the parallel classes

The compilation of the parallel classes are done using `-object` option of the POP-C++ compiler:

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

Again we have to explicitly compile the declaration of the parallel class. This command will generate the `myclass.obj` file which contains the executable code of the parallel class `myClass`.

The compiling process is now terminated and we produced two executable files:

- `main`: contains the executable code for the `main` program. This executable will be launched and executed on the local machine when running the program
- `myclass.obj`: contains the executable code for parallel objects which are instance the parallel class `myClass`. This executable will be remotely launched on the processing unit that will run the parallel object.

4.2.3 Running POP-C++ programs

To execute a POP-C++ application we need to generate the **object map file** which contains the list of all compiled parallel classes used by the application. Indeed, as POP-C++ allows to run distributed application in heterogenous environments, we have to compile parallel class for each potential computing unit architectures where we want instances of this parallel class be able to run. Therefore the object map file contains for each parallel class for which architecture the compilation has been done and the location of the corresponding executable file (`.obj` file).

With POP-C++ it is possible to get this information by executing the object executable file with the option `-listlong`.

Example for the `myClass` parallel class, if we type:

```
./myclass.obj -listlong
```

the following message will be displayed:

```
myClass i686-pc-Linux /home/myloging/popc/test/myclass.obj
```

which indicates that the file `myclass.obj` contains the executable code of the parallel class `myClass` compiled for `i686-pc-linux` architecture and that the name (and path) of the file is `/home/myloging/popc/test/myclass.obj`

To generate the object map file we simply redirect the output to the object map file:

```
./myclass.obj -listlong > obj.map
```

The object map file must contain all mappings between object names, platforms and the executable files locations.

We can now run the program using the command `popcrun`:

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

4.3 Compiling a complex POP-C++ program

The compilation is a little bit more difficult for more complex applications using several different parallel classes. This is the case, for example, when the main program calls methods from objects of different parallel classes or when there is a chain of dependencies between the main program and several parallel classes as illustrated on figure 4.2.

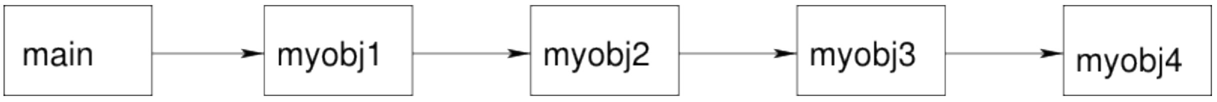


Figure 4.2 - Parallel classes with dependencies

Since each class contains some internal POP-C++ classes (which are transparent to the programmer) such as the **interface** or the **broker** classes, the compilation must avoid to create multiple definitions of these classes. An easy way to avoid this is to begin the compilation with the last class of the chain (the class `myobj4` on the example of the figure 4.2) and then to compile each parallel class in reverse order. To compile any class in the chain we needs the parallel class which is directly after the one we are compiling in the chain of dependency. To do so we need to generate relocatable code files (`.o` files) that can be linked using a C++ compiler. As in the normal C++ compiling process, this is done using the `-c` option (compile only) along with the `popcc` command.

When compiling a parallel class without generating the executable code (using the `gcc` option `-c`), the POP-C++ compiler generates a relocatable object file called `className.stub.o`. In addition the POP-C++ compiler has an option called `-parclass-nobroker` which allows to generate relocatable code without internal

POP-C++ classes.

The way to compile a POP-C++ application with dependencies as illustrated on figure 4.2 is shown in figure 4.3.

```
    popcc -object -o myobj4.obj myobj4.ph myobj4.cc
    popcc -c -parclass-nobroker myobj4.ph
    popcc -object -o myobj3.obj myobj3.ph myobj3.cc myobj4.stub.o
    popcc -c -parclass-nobroker myobj3.ph
    popcc -object -o myobj2.obj myobj2.ph myobj2.cc myobj3.stub.o
    popcc -c -parclass-nobroker myobj2.ph
    popcc -object -o myobj1.obj myobj1.ph myobj1.cc myobj2.stub.o
    popcc -o main main.cc myobj1.ph myobj1.cc myobj2.stub.o
```

Figure 4.3 - How to compile applications with dependencies

The commands:

```
    popcc -object -o myobjX.obj ...
```

creates the complete executable file `myobjX.obj` when the commands:

```
    popcc -c -parclass-nobroker myobjX.ph ...
```

creates a firl called `myobjX.stub.o containing the` relocatable binary code which will be used to be linked with other parallel classes or with the main program.

The `-parclass-nobroker` option of the POP-C++ compiler indicates that the produced relocatable binary code must not contain internal POP-C++ classes.

Compiler options specific to the POP-C++ compiler are presented in the appendix A of this manual.

More examples of POP-C++ source files with corresponding `Makefile` can be found in the `test` directory of the POP-C++ distribution and on the POP-C++ web site:
`http://gridgroup.hefr.ch/popc`.

CHAPTER 5 Installation Instructions

5.1 Introduction

POP-C++ has been designed to run applications in large heterogenous distributed computing environments. Nevertheless, when starting using POP-C++ or for testing purposes it is sometime very useful to be able to run POP-C++ programs on a very simple infrastructure. This is why POP-C++ can be installed in two different ways:

- **Standalone installation** to install POP-C++ on a single machine. With this installation each parallel object will run in separated processes on the same computer.
- **POP-Community installation** to run POP-C++ in a complex distributed infrastructure.

Standalone installation is the easiest way to install POP-C++ on a single computer. It is useful to get familiar with POP-C++ programming or to test POP-C++ programs before running them on a complex hardware infrastructure. Of course, using POP-C++ in this way does not allow you to increase the computing power you have access to.

If you are beginner with POP-C++ we strongly recommend you to start with the standalone installation. This installation can at any be upgraded to a POP-Community installation by re-running the set-up of POP-C++ (see section 5.3.3).

5.2 Before installing

POP-C++ has is built on top of several widely known software packages and, therefore, has some prerequisites which are described in the next sub-sections.

5.2.1 Prerequisites

Before installing and running POP-C++ the following packages must be installed:

- A standard C++ compiler (g++)

- the `zlib-devel` package[2]

Optional packages are :

- the GNU Bison[3]
- the Globus Toolkit[4]

5.2.2 Location of the Files

Before installing POP-C++ you have to decide about files locations:

- *In which directory the source files will be downloaded ?*
  - This directory will be the root of the directory tree where you will download all the source files of POP-C++.
  - It will also contains the compiled files during the installation.
  - This directory should hold roughly 250 MB (all sources and compiled files)
  - It can be erased after the installation
  - We call this directory `<source dir>` in the present document
  - There is no default value for the name of this directory
- *In which directory the POP-C++ runtime will be installed ?*
  - It should hold less than 100 MB
  - This directory is necessary on every computer where POP-C++ is installed.
  - The default name is `/usr/local/popc`.
  - We call this directory `<install dir>` in the present document
- *In which directory will be stored temporary files ?*
  - This directory contains the temporary files produced by POP-C++ when running.
  - The default name for this directory is `/tmp`
  - This directory will be asked during the installation process
  - We call this directory `<temp dir>` in the present document.

5.2.3 Downloading the POP-C++ Distribution

POP-C++ is distributed in form of a `tar` file with the following naming convention :

`popc-<version.subversion>.tgz`

You can download the tar file either

- Directly from the website (recommended) of the GridGroup who develops and maintains the tool POP-C++:
  http://gridgroup.hefr.ch/popc/doku.php/download
- From sourceforge: `http://sourceforge.net/projects/popcpp/`

[2] The name of the package is distribution dependent. For example the `zlib-devel` package is named :
on Fedora (red hat based) : `zlib-devel`
on Ubuntu (debian based) : `zlib1g-dev`

[3] This package is only necessary for those who wants to modify the POP-C++ tool itself

[4] This package is only necessary if you want to install POP-C++ over Globus. This installation is not described in the present document and is not supported since version 2.5 of POP-C++

- From freash meat: http://freshmeat.net/projects/pop-c

POP-C++ is available for Linux and MacOS. Download the version corresponding to the OS which run on the computer you want to install POP-C++.

POP-C++ is currently not available for Windows.

5.3 Installation

This section describe how to install POP-C++ without any special option. It's also possible to customize the installation. Customized installation is explained in more detail in the document: Advanced POP-C++ User Manual.

5.3.1 Preparing compilation

After downloading the tar file, you must decompress it :

```
tar -C <source dir> -zxf popc_<version.subversion>.tgz
```

Note : *You must check that you have R/W access to the* `<source dir>`

Then go to `<source dir>`

```
cd <source dir>
```

If you want have a POP-C++ with all default options, you can now configure the compilation files by entering the following command on a **Linux** operating system:

```
./configure
```

and the following command on a **MacOS** operating system
(*only for POP-C++ version older that 2.5*)

```
./configure CPPFLAGS=-DARCH_MAC
```

Default options are :

- `<install dir>` = `/usr/local/popc`
  (assuming you did note defined another directory in `POPC_LOCATION` !)

Remember that you must have write access to this directory. If you want to install POP-C++ in another directory use use the parameter **--prefix**=<`install dir`>.

```
./configure --prefix=<install dir>
```

or, if the variable `POPC_LOCATION` has been defined :

```
./configure --prefix=$POPC_LOCATION
```

Several other parameters are available with the `configure` command but they are usually only useful for complex POP-C++ installation. More details about the available options can be found by typing the command :

```
./configure --help
```

5.3.2 Compiling POP-C++ tools

When configured, you can compile POP-C++ tool by entering the command :

```
make
```

Note: *Do not forget that you need to have write access to the current directory.*

This command can take several minutes.

5.3.3 POP-C++ Setup

If the compilation completes successfully you can install POP-C++ in the `<install dir>` by entering the command :

```
make install
```

Note : *you must have read/write access to the* `<install dir>`

Shortly before the installation finished, the system will automatically launch the `popc_setup` script. This script can be re-launched at any time after the installation to modify the POP-C++ set-up. The set-up is divided in two parts, the first one configures your POP-C++ installation and the second one creates startup scripts which are used by POP-C++ applications.

Standalone installation

At this time, you will be able to choice if you want to do **Standalone** installation or **POP-Community** installation. The following question is displayed:

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

By answering **y**, you make an standalone installation of POP-C++. No more question will be asked and all default values will be used. This is the simplest way to install POP-C++ for the first time. If you need to change the set-up, you can at any time launch the POP-C++ set-up again by typing `make install` and by answering `n` to this question.

Note : *instead of re-launching* `make install`, *you can also launch the script* `popc_setup`. *This will be explained later in this document.*

When the installation is terminated the following message is displayed:

```
IMPORTANT : Do not forget to add these lines to your .bashrc
file or equivalent :
---------
    POPC_LOCATION=<install dir>
    export POPC_LOCATION=<install dir>
    PATH=$PATH:$POPC_LOCATION/bin:$POPC_LOCATION/sbin
```

Add these lines in the corresponding files to allow POP-C++ to access needed tools or refer to section 5.5 to learn about an easy alternative to do it.

POP-Community installation

By answering **n**, you can make a POP-Community installation. POP-C++ allows for different ways to create a POP-Community. In this document we will only present the most basic and most used way. A typical POP-Community is a set of computers sharing disks space and users accounts using, for example, NFS (Network File system). In addition each node of the community can start an SSH session on any others node of the POP-Community but each node does not need to know the addresses of all others nodes of the POP-Community. The only constraint to fulfill is that the graph created by linking node A to node B, if A knows the address of B is a bidirectional connected graph. At any time a new node can join the POP-community by contacting a node already belonging to the POP-Community and by exchanging public SSH keys and IP addresses. Figure 5.1 shows an example of POP-Community.

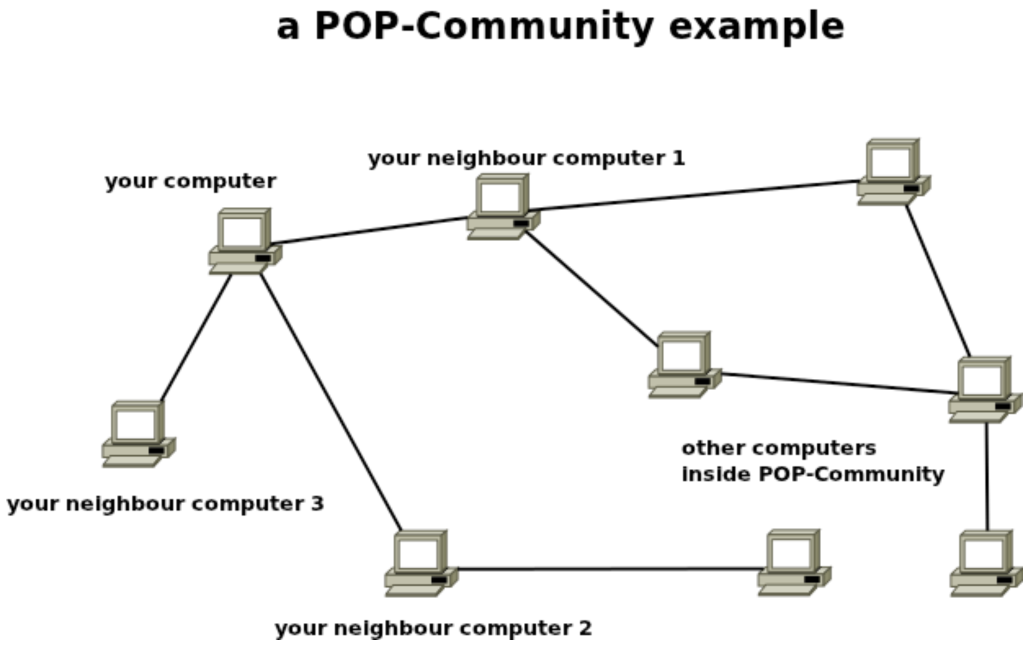

a POP-Community example
your computer
your neighbour computer 1
your neighbour computer 3
other computers inside POP-Community
your neighbour computer 2

Figure 5.1 Example of a POP-Community

To allow POP-C++ to create the POP-Community you will have to answer, during the installation, to several questions which are detailed below.

> **Note** *: if you answer by typing the* `<return>` *key to all questions, the effect will be exactly the same as making a standalone installation.*

As mentioned above, in a POP-Community, the computing resource topology is a connected graph and any computing resource can join the environment by register itself to other node(s) in the community (its **neighbor nodes**(s)). To make your local computer joining a community, you have to select one or several nodes belong to this community and to make them your neighbors. You must at least enter one neighbor node by answer to the following question:

```
Enter the full qualified neighbour host name or IP address :
```

You can give the IP address or the name the of machines which are in the POP-community you want to join.

If you are installing POP-C++ on **the first machine of a POP-Community** you still can enter addresses of machines where you intend to install POP-C++ and that will be part of the same POP-Community. You can also leave neighbor list empty but in such a case you will have to re-run the POP-C++ set-up later in order to make your machine joining the POP-Community.

When you have entered all the names of your neighbor nodes just type `<return>` to terminate the process.

> **Note** *: do not define your computer as your "neighbor node" ! POP-C++ will hang if you do this. The script will inform you about this but will allow you to do it.*

POP-C++ needs to know how many processor are available on your system.

```
Enter number of processors available (default:1):
```

This is mainly used when installing POP-C++ on the front end of a cluster. In such a case indicate the number of computing node can the front end will manage. Be aware that a multi-core processor is considered as a single processor.

The next question is the number of parallel objects that can simultaneously run on the local machine:

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

> **Note :** *The default value is 100 because in standalone mode all parallel objects run on the same machine. Be aware that each parallel object is a full Unix/Linux process.*

The next question concern the amount of memory you want to allocate to POP-C++ jobs. You may not want to give all memory available. So indicate in MB the maximum amount of memory you agree to allocate to POP-C++ jobs:

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

As already mentioned, each POP-C++ object is Unix/Linux process. Therefore, each process is owned by a specific user. You can force the user under which the POP-C++ jobs must run. Of course, you must have the rights to run processes under this username. By pressing just <return> the user you are actually logged in will be used.

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

The last questions are very advanced features of POP-C++. They allow to configure POP-C++ for very specific situations of for testing purposes. If you are note a great specialist of POP-C++ just type `<return>` to keep default values.

You can specify the name of the script which will submit jobs (parallel objects) on the local machine. The name of the script can be given by answering to the question :

```
Enter the script to submit jobs to the local system:
```

If a particular protocol is required you can give it by entering the communication pattern :

```
Communication pattern:
```

> **Note:** *Communication pattern is a text string defining the protocol priority on binding the interface to the object server. It can contain "*" (matching non or all) and "?" (matching any) wildcards.*

For example: given communication pattern `socket://160.98.* http://* :`

- If the remote object access point is :
  `socket://128.178.87.180:32427 http://128.178.87.180:8080/MyObj`
  the protocol to be used will be “http”.
- If the remote object access point is :
  `socket://160.98.20.54:33478 http://160.98.20.54:8080/MyObj`
  the protocol to be used will be “socket”.

If special runtime environment variables are required, you can give them now. Give the name of the variable then the system will ask the value. After giving the value, the system will again ask for a variable name until you press `<return>` without giving a name.

If no special environment variable are required by the POP-C++ application, just enter `<return>` to the question :

```
Enter variable name:
```

In other case, give the name and the procedure will ask the value :

```
Enter variable value:
```

The procedure ask for a new variable as long as you define any. If you do not need to define more variable just press `<return>`.

Because POP-C++ uses the network, it will use a communication port. The default port is 2711. If for some reason you want to change it, you can do it here. Note that in case a firewall is installed between the nodes the port must be open on this firewall.

```
Enter the service port[2711]:
```

The domain name is also asked by POP-C++. Thus this is not mandatory.

```
Enter the domain name:
```

POP-C++ uses `<temp dir>` to store informations during execution. Default directory is `/tmp` but you are can change it here.

Note: You must have read/write access to this directory !

```
Enter the temporary directory for intermediate results:
```

As for standalone installations, when the installation is terminated the following message is displayed:

```
IMPORTANT : Do not forget to add these lines to your .bashrc
file or equivalent :
---------
    POPC_LOCATION= <install dir>
    export POPC_LOCATION= <install dir>
    PATH=$PATH:$POPC_LOCATION/bin:$POPC_LOCATION/sbin
```

Add these lines in the corresponding files to allow POP-C++ to access needed tools or refer to section 5.5 to learn about an easy alternative to do it.

5.4 Testing Installation

Several test programs allowing to test your installation are provided with POP-C++ distributions. These programs are located at:

```
<source dir>/test
```

This directory is copied in the `<instal dir>` during installation, thus it is also available at:

```
<install dir>/test
```

Be aware that to run test programs, you need to have "execute (`X`)" permission on the directory you are using to run test. By default it is not the case for the `<install dir>`.

Before running tests you have to start the POP-C++ deamon (we call the **Job Manager**). The Job Manager is used by POP-C++ to find and allocate computing resources during executions of POP-C++ programs. Refer to section 5.5 to learn how to launch the Job Manager.

Then run the tests by typing:

```
./runtests_short
```

You also can run each test individually.

Type:

```
./runtests --help
```

for more information.

5.5 The Job Manager

As mentioned at the end of section 5.3.3, it is required that you define paths by adding lines to your login file (files like `.profile`, `.bashrc`, `.cshrc`, etc.). To ease this task, the POP-C++ installation provides you a script which makes this work for you. You just have to call this script at the end of your login file. This script is provided for C-shells and Bourne shell in the following corresponding files:

```
<install dir>/etc/popc-user-env.csh
```

and

```
<install dir>/etc/popc-user-env.sh
```

If you do note use these shells (or a shell compatible with these shells) you have to add all the necessary lines in your login file (refer to section 5.3.3).

When this is done launch the Job Manager by typing the following command:

```
SXXpopc start
```

Note: *the* `SXXpopc` *script is located at:* `<install dir>/sbin/SXXpopc`

Again, be aware that to launch the Job Manager you need to have "execute (`X`)" permission on the corresponding directory.

Note: *You can at any time* `stop`, `restart` *or* `kill` *the Job manager by using the corresponding keyword. After any new installation or configuration of POP-C++, kill the Job Manager and start it again:*

```
SXXpopc Kill

SXXpopc start
```

Before executing any POP-C++ application, the runtime system (Job Manager) needs to be started. It must be launched on every node of the POP-Community by entering the command:

```
SXXpopc start
```

Note: *In some specific cases POP-C++ does not require that the Job Manager is running, nevertheless, we strongly recommend to always launch the Job Manager to avoid POP-C++ programs execution failures.*

Usually when a POP-C++ program cannot reach the job manager, for example because it is not running, one of the following errors is displayed:

```
Cannot create object via POP-C++ Job Manager
```

or

```
Fail to bind to the remote object broker
```

Appendix A Command Line Syntax

A.1 Compiling an application

In this appendix, only options specific to POP-C++ are presented. However all options of the C++ compiler your are using (by default g++) are also available.

```
popcc [-cxxmain] [-object[=type]] [-cpp=<C++ preprocessor>] [-cxx=<compiler>] ]
[-popcld=linker] [-popcdir=<path>] [-popcpp=<POP-C++ parser>] [-verbose] [-
noclean] [other C++ options] sources...

   -cxxmain:               Use standard C++ main (ignore POP-C++ initialization)
   -popc-static:           Link with standard POP-C++ libraries statically
   -popc-nolib:            Avoid standard POP-C++ libraries from linking
   -parclass-nointerface: Do not generate POP-C++ interface codes
                           for parallel objects
   -parclass-nobroker:     Do not generate POP-C++ broker codes
                           for parallel objects
   -object[=type]:         Generate parallel object executable (linking only)
                           (type: std (default) or mpi)
   -popcpp:                POP-C++ parser
   -cpp=<preprocessor>:    C++ preprocessor command
   -cxx=<compiler>:        C++ compiler
   -popcld=<linker>:       C++ linker (default: same as C++ compiler)
   -popcdir:               POP-C++ installed directory
   -noclean:               Do not clean temporary files
   -verbose:               Print out additional information
   -nopipe:                Do not use pipe during compilation phases
                           create _paroc2_ files)
   -version:               Display the installed POP-C++ version

   Environment variables change the default values used by POP-C++:
      POPC_LOCATION:   Directory where POP-C++ has been installed
      POPC_CXX:        The C++ compiler used to generate object code
      POPC_CPP:        The C++ preprocessor
      POPC_LD:         The C++ linker used to generate binary code
      POPC_PP:         The POP-C++  parser
```

A.2 Running an application

To run aPOP-C++ application type the following command where *popc_options* is one or several of the options described below.

```
popcrun objects.config [popc_options] prog.main args...
```

```
  -drun:             Print launching command only.
  -runlocal:         Force to create all objects locally : do not use JobMgr
  -debug:            Give some debugging informations
  -log=<filename>:   Put all message in the <filename> file
  -version:          Display the installed POP-C++ version
```

Appendix

B Environment variables

The following environment variables affect the default behaviors of the POP-C++ runtime. To ensure proper execution of parallel objects these variables are set during the installation process (`make install`).

Manipulation of these variables can lead to an unstable POP-C++ environment and should be done only if your are an expert of the inside of the POP-C++ tool.

These variables can be manipulated directly or using the scripts `popc-runtime-env.*` located in the directory `<install dir>/etc`.

If you expect problems with the values of these variables, re-run the installation process (see section 5.3)

POPC_LOCATION	Location of installed POP-C++ directory.
POPC_PLUGIN_LOCATION	Location where additional communication and data encoding plugins can be found.
POPC_JOBSERVICE	The access point of the POP-C++ job manager service. If the POP-C++ job manager does not run on the local machine where the user start the application, the user must explicitly specify this information. Default value: `socket://localhost:2711`.
POPC_HOST	Full qualified host name of local node. This host name will be interpreted
POPC_IP	IP of local node. Only used if POPC_HOST is not defined
POPC_IFACE	If `POPC_HOST` and `POPC_IP` are not set, use this interface to determine the node IP. If not set, the default gateway interface is used.
POPC_PLATFORM	The platform name of the local host. By default, the following format is used: `<cpu id>-<os vendor>-<os name>`.
POPC_MPIRUN	The `mpirun` command to start POP-C++ MPI objects (not documented in this manual).
POPC_JOB_EXEC	Script used by the job manager to submit a job to local system.
POPC_DEBUG	Print all debug information.