`C++//` on top of PM^2 via Schooner

Françoise Baude - Nathalie Furmento - Denis Caromel
Projet SLOOP
INRIA Sophia-Antipolis - I3S URA 1376 - UNSA
930 Route des Colles, B.P. 145
06903 Sophia-Antipolis Cedex
{baude,furmento,caromel}@unice.fr

Raymond Namyst - Jean-Marc Geib - Jean-François Méhaut
Projet ESPACE
LIFL Lille - CNRS URA 369 -USTL
Bâtiment M3 - Cité scientifique
59655 Villeneuve d'Ascq Cedex
{namyst,geib,mehaut}@lifl.fr

Résumé

Dans cet article, nous présentons une implantation d'un support d'exécution d'une extension parallèle de C++, `C++//`. Cette implantation se base sur SCHOONER qui est une librairie de communications orientée objet, elle-même mise en œuvre grâce à l'utilisation d'un support d'exécution parallèle multi-tâche, PM^2.

1 Introduction

The *SLOOP* (Simulation, and Parallel Object-Oriented Languages) project, is a recently-formed research team with approximately 15 members at I3S-CNRS, the University of Nice-Sophia Antipolis, and INRIA Sophia Antipolis. This team is investigating three research areas: parallel and distributed discrete event simulations, parallel object-oriented languages, and communications and interconnection networks. These three domains are not independent: each level needs and uses the primitives and capabilities of the layer beneath it.

One of the research objectives of the project is the development of `C++//`, a parallel extension of C++ language. `C++//` is built on top of SCHOONER, an object-oriented support, itself based on PM^2, an environment developed by the Espace project, which provides a computing model based on lightweight process. The purpose of this article is to present this structuring.

The rest of this paper is organized as follows. Section 2 gives a quick overview of the `C++//` language. In Section 3, we present the SCHOONER library. Then, Section 4 focuses on how `C++//` is built upon SCHOONER. Sections 5 and 6 respectively present the PM^2 environment and how it is used by SCHOONER.

2 The `C++//` language

The `C++//` language (pronounced "C++ parallel") has been designed with the aim of importing reusability into parallel and concurrent programming in the framework of a MIMD model. `C++//` defines a comprehensive and versatile set of libraries, using a small set of simple primitives, *without extending the syntax of C++*. The libraries are themselves extensible by end-users, making `C++//` an open system.

We begin by describing the basic features of the programming model, which is a distributed memory MIMD model, on top of which other models could be built [2].

2.1 Programming model

This section describes four important characteristics: parallel processes, communication between them, synchronization, and data sharing.

2.1.1 Processes

All objects which are instance of a class which publicly inherits from the `Process` class are processes (**active objects = processes**). Passive objects (i.e. objects which are not active) belong at run-time to a process object, thus giving a distributed structure of active objects encapsulating passive objects. One possible way to create a process is as follows:

```
class Parallel_A : public A, public Process {
   ...
};
         ⋮
Parallel_A* p;
p = new Parallel_A ( ...  );
```

Because our system is oriented towards reuse and software engineering of parallel systems rather than operating systems programming, we choose to make processes sequential. Thus the model does not allow the user to program multi-threaded processes, but this does not prevent multi-threading at the operating system (OS) level. As we will see in Section 4, several sequential processes can be implemented with one multi-threaded operating system process for the sake of light-weightness.

2.1.2 Communication

Since a process is an object, it has member functions. When an object owns a reference to an active object, it is able to communicate with it by calling one of its public members. All communications towards active objects appear syntactically as member function calls and are asynchronous by default.

```
p→f( parameters );
```

Synchronous function calls are possible in `C++//`, but must be specified explicitely in either the function call or the process definition. Function calls towards passive objects are synchronous as happens in standard sequential C++.

More precisely, each call issued to a process member function is deposited into the list of pending calls, denoted requests, of the process. The user is then free to explicitely program the way the process serves them (i.e. extracts requests from the list of requests). By default the service policy (i.e. the programming control of the `Process` class defined by the `Live()` routine) is FIFO.

Regarding the communication semantics, there are in fact three different protocols thanks to which a request is delivered to an active object, either by an active or a passive object. Note that each active object follows exactly one of the three protocols at a time, and that the sender object issuing a call automatically get in conformity with the communication protocol of the receiver object.

1. *reactive rendezvous protocol*: the sender is blocked until: (1) a rendezvous is made with the receiver which is interrupted in order to enter the rendezvous, (2) the request is transmitted into the receiver context (i.e. into its list of requests).

2. *reactive protocol*: the communication is non blocking for the sender, and the receiver is also interrupted when a new request arrives (no rendezvous).

3. *proactive protocol*: the communication is non blocking for the sender, and the receiver is not interrupted; some explicit receiving action has to be undertaken by the receiver in order to get the newly arrived request(s).

The design of `C++//` pursues the work done with the design of Eiffel// [1] in which only the reactive rendezvous protocol was provided.

2.1.3 Synchronization

Asynchronous communication can be difficult for programmers to manage. For example, since function calls to active objects are asynchronous, one needs to explicitely synchronize before using result values to make sure they have been returned by the active object. We use a simple rule, called *wait-by-necessity*, to address this problem.

In `C++//`, an object is automatically blocked when it attempts to use the result of a parallel member function call that has not yet been returned. Thus, a caller does not wait for the result of an asynchronous function call until that value is explicitely used in some computation. Should a value not have been returned at that point, the caller is automatically blocked until the value becomes available. This mechanism is a form of future [6].

```
v = p→f( parameters );
...
v→foo();                    // Automatically triggers a wait
                            // if v is awaited
```

2.1.4 Sharing

The semantics of communications towards processes is a copy semantics: all parameters which are passive objects are automatically transmitted by copy. Of course, active objects are subject to a reference semantics: all processes are transmitted by reference. As a consequence, shared objects (i.e. passive objects which are shared between active objects) are not allowed as such. The implication of this is that any passive object which is to be shared between several active objects must be made an instance of a `Process` heir.

2.2 Implementation

`C++//` is based on a Meta-Object Protocol (MOP). This permits (1) to program the `C++//` programming model without any syntactical extention, through a library of classes, and (2) to allow other programming models to be defined as well. We give an overview of this MOP which is proposed by the *Europa* working group as a standard for parallel programming in C++ (see [3] for more details).

The first principle of the MOP is embodied in a special class, called `Reflect`. All classes inheriting publicly from `Reflect`, are called *reified classes*. All calls issued to instances of reified classes are reified, i.e. trigger the execution of the member function `reify()` with the appropriate object of type `Call` as a parameter. Figure 2.1 illustrates the reification mechanism. This technique is important for reusability as it permits users to take a normal class, and then globally modify its behaviour, in order to transform it into a process for example.

```
class Reflect {
protected:
   virtual void reify(Call* c) {   // A call reification
      c→execute();
   }
public:
   void* operator new(...)  {
      ⋮
   }
};
```

```
class Call {
public:
   virtual void execute();
   List<Any>* eff_params;     // effective parameters
   member_id m;               // member to be called
   Any object;                // target object
   Any result_place;          // result address
};
```

The creation of an instance of a reified class returns a meta-object (called proxy in our terminology) for the type being passed in as a parameter. Proxies classes are generated by a pre-processor.

The `Process` class simply redefines the `new` operator in such a way that additionally to the launching of the - possibly remote - active object, a proxy object is created. The proxy then records the OS process identifier of the active object.

Moreover, a `Request` class is defined as an heir of the `Call` class and `Process::reify(Request *r)` triggers the sending of the request. More precisely, the request contains a reference to an instance of the `Future` class especially created in order to subsequently store the result of the member function call. This result will be conveyed by an object of type `Reply`, which is defined and managed very like an object of type `Request`. Thus the `Request` class has an additional member which is a copy of the caller proxy, subsequently used to send the reply (through a member function `Process::reify(Reply *r)`).

Needless to say that prior to the sending of a request (or reply), it has to be marshalled if the destination active object is not running in the same OS context.

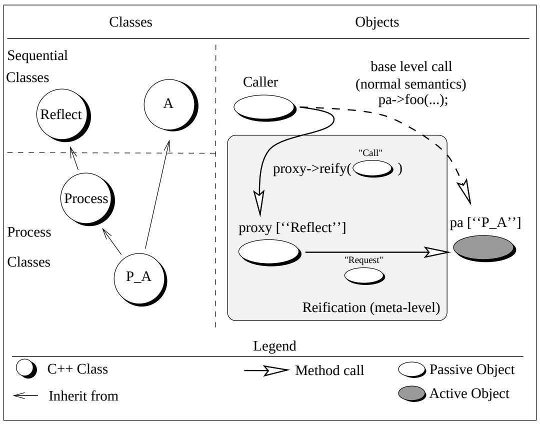

Figure 2.1 : Reification of Calls

One important additional effort has been to identify which concepts for the implementation of `C++//` are in fact OS and architecture dependant, as for instance heavy or light-weightness of OS processes to support active objects, active objects interruption, effective communications towards active objects. As one important concern of `C++//` is to be an open system, in order to have access to for example the most efficient communication libraries, we have grouped the definition of those concepts outside this level of the `C++//` system - level which can thus be qualified as the *proxy level*. All the target dependant concepts needed to implement the proxy level are thus provided by an underlying level, called SCHOONER. As we will see in the next section, SCHOONER itself has as a main concern to be open so is essentially a set of bindings which relate to - preferably standard - communications and multi-threading libraries.

3 SCHOONER

3.1 Introduction

SCHOONER [5] (which stands for **S**loop **C**ommunication Library for **H**igh-level, **O**bject-**O**riented, a**N**d **E**fficient **R**emote Exchanges) is a library of classes used for writing parallel and distributed applications in C++. This library consists in a object-oriented run-time support. Its main purposes are:

- to be independent of the used underlying communication - and possibly multi-threading - library,
- to give the user a structured and abstract view of its application (farm of available computers, set of computation nodes, . . .),
- to allow a straightforward redefinition of its behaviours thanks to the object-oriented technology. For example, it's straightforward to add a new message type or to change the application scalability.

The SCHOONER library can be split into three layers (see Figure 3.1):

1. the first layer is an interface to any communication package;
2. the second layer allows the user to create computation nodes also called clusters on chosen computers; these clusters can interact thanks to active messages (see Section 3.2);
3. a third layer supports communicating objects which can also interact via messages (see Section 3.3). This layer thus provides the user with an asynchronous message-passing programming model. An important point, not directly connected to this paper concerns, is that this third layer could also be replaced by an asynchronous RPC programming model (like for instance the one provided by PM^2 (see Section 5) or Nexus [4]).

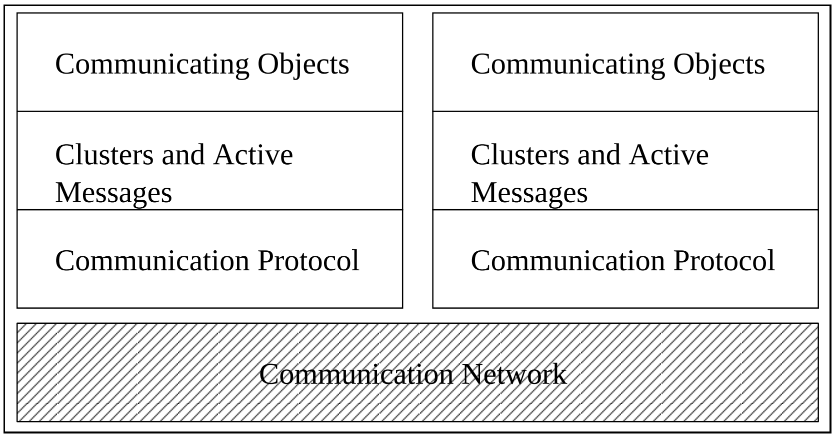

Figure 3.1 : The different layers of SCHOONER

3.2 Clusters and active messages

The first two layers of SCHOONER aim to gather all common basic concepts needful to support - either leightweight or heavyweight - concurrent distributed applications development and execution on either homogeneous or heterogeneous virtual parallel computers. The definition of this set of concepts arises from the following observations.

At run-time, computation entities need at least some CPU and memory resources which are always provided by an heavyweight OS process, called cluster in SCHOONER. These processes form the core of the parallel virtual machine, so one has to be able to define their locations. These processes usually do not share memory, so whatever the provided computation model be (RPC or traditional message-passing, at the third level of SCHOONER), at least a basic asynchronous send/receive mechanism is needed to exchange data - implemented by an active message mechanism. Because remote data exchanges have to pass the OS processes frontiers, data have to be marshalled and latter unmarshalled, and an encoding scheme must be applied whenever the implied computers do not code primitive data in the same format. Again, notice that thread related concepts do not belong to this basic set.

We will now detail the SCHOONER classes a user (for example, the `C++//` run-time developer) has to manipulate.

3.2.1 Definition of clusters

A cluster can be thought of as an entity which designs a context (memory, CPU, ...) on a computer. We can also find this notion in other libraries: environment process of PT-PVM [7], context in Nexus, module in PM^2.

A cluster is defined by the `Cluster` class (see Figure 3.2). An identifier of cluster (`Cluster_Id` class instance) has a global meaning in the entire application.

```
class Cluster {
   protected:
      Cluster_Id *id;                              // cluster identifier
   public:
      static Cluster *any();                       // any cluster
      static Cluster *best();                      // best cluster
      static Cluster *current();                   // current cluster

      virtual bool destroy();                      // destroy cluster
      void *operator new(size_t, Computer *);      // allocator
      Cluster();                                   // constructor
      virtual ~Cluster();                          // destructor
}
```

Figure 3.2 : Partial definition of the `Cluster` class

The member function `destroy()` is called to destroy the effective cluster entity (which can be remote), whereas the function `~Cluster()` destroys the C++ object (whose aim was to "trigger" the cluster creation, as for examples `cl1, cl2, cl3` in Figure 3.4).

3.2.2 Creation of clusters

The location for a new `Cluster` object is given thanks to an instance of the `Computer` class (see Figure 3.3). This class supports various selection functions such as: a specific computer, the "best" computer, ... An instance of this class is defined by three informations: a symbolic name, the real name of the computer (network name, IP address) and its architecture.

Many computers with different symbolic names can be associated to the same physical machine. The use of symbolic name allows to define a virtual architecture which can be mapped on different configurations. We can also have a configuration file which will automatically be used at run-time to instantiate computers.

Figure 3.4 shows examples of `Cluster` objects creations. There are two ways to give the location

```
class Computer {
  protected:
    String symbolic_name;                  // symbolic name
    String real_name;                      // real name
    Computer_Arch arch;                    // computer architecture
  public:
    Computer(String&, String&);            // constructor

    static Computer *any();                // any computer
    static Computer *best();               // the "best" computer
    static Computer *current();            // current computer
    static Computer *specific(String &);   // the computer with the
                                           // given symbolic name
}
```

Figure 3.3: Partial definition of the `Computer` class

of a new cluster:

1. The programmer can specify to the allocator the target computer either by giving an instance of the `Computer` class, or by giving the symbolic name of an existing `Computer`.
2. By default, the cluster will be mapped on the "best" computer which is selected at run-time according to a load balancing scheme.

```
Computer *cp = new Computer(``mach'', ``lorien.m_earth.fr'');
Cluster *cl1, *cl2, *cl3;
cl1 = new(Computer::current()) Cluster();          // on current computer
cl2 = new(``mach'') Cluster();                     // on specific computer
cl3 = new Cluster();                               // on default computer
```

Figure 3.4: Creation of clusters

3.2.3 The "best" clusters and computers

The policies used to choose the "best" computer or the "best" cluster are normally the ones provided by the underlying used communication package (PVM, PM^2, ...). They could be redefined as follows: the "best" computer could be the less-loaded one (memory, CPU, ...), and the "best" cluster could be the one which contains the fewest communicating objects. We can enhance this policy by adding parameters to the two functions `Cluster::best()` and `Computer::best()`. In this way, SCHOONER enables to easily adapt the selection criteria (memory load, computation power, ...) depending on the target application.

3.2.4 Active messages

The model for inter-cluster communications is based on an active message mechanism. Active messages [12] allows to extend the *message-passing model* to a *single-sided operation*. Once a message is received by a cluster, a processing function dependent of the received message type is automatically triggered. Furthermore, the call to the reception function being implicit, the destination object does not need to call the reception function in order to get the newly available messages.

This active message mechanism implements the basic asynchronous send/receive mechanism described in Section 3.2. This mechanism will also be especially useful to support the implementation of the communication protocols of the `C++//` language.

An active message object is an instance of a class which publicly inherits (directly or indirectly) from the `Message` class (see Figure 3.5). This class and its heirs define the following member functions:

1. `flat(...)` which is used to flatten an object `Message` into an object `Buffer` in order to send it to another cluster,
2. `rebuild(...)` which is used to build an object `Message` from an object `Buffer`,
3. `process(...)` which specifies the processing to perform when an object of class `Message` is received by a cluster.

```
class Message {
  public:
    virtual Buffer *flat(Buffer *buff=0)=0;
    static Message *rebuild(Buffer *buff, Message *msg=0);
    virtual void process()=0;
}
```

Figure 3.5 : Partial definition of the `Message` class

The definition of a new class of active messages is quite easy: the user must only (1) define a new class which inherits from the `Message` class, with the associated data, (2) write the two marshalling functions (`flat()` and `rebuild()`), and (3) write the `process()` function.

3.3 Communicating objects

This constitutes a third possible layer of SCHOONER, offering a communicating objects programming model. These objects could be passive objects, and by a simple derivation of their class, these objects can have one thread of control (mono-active objects) or many threads of control (multi-active objects). The design of this level has been conducted with the aim to support the `C++//` run-time (mainly to support active objects of `C++//`). So, we will concentrate the discussion of Section 4 on a version of this third layer offering mono-active communicating objects. Of course, the user of this third level must be free to have 1 or n communicating objects mapped on a cluster object.

3.3.1 Definition and creation of communicating objects

Creation of a communicating object is done through the creation of a proxy object. A proxy is a local representant of the communicating object which is on a local or remote cluster. The proxy allows to send data to the communicating object.

A communicating object is defined by two classes: the `Proxy_Comm_Object` class (see Figure 3.6) and the `Comm_Object` class (see Figure 3.7). When a cluster wants to create a communicating object, in fact, it has to create an instance of the `Proxy_Comm_Object` class. An instance of the `Comm_Object` class is then automatically created on the cluster object given in parameter to the allocator of the `Proxy_Comm_Object` class.

```
class Proxy_Comm_Object {
  protected:
    Comm_Object_Id *id;                        // object identifier
    Cluster *cl;                               // object location
  public:
    virtual void send(Data_Comm_Object *);     // send function

    void *operator new(size_t, Cluster *);     // allocator
    Proxy_Comm_Object();                       // constructor
    virtual ~Proxy_Comm_Object();              // destructor
}
```

Figure 3.6: Partial definition of the `Proxy_Comm_Object` class

```
class Comm_Object {
  protected:
    Comm_Object_Id *id;                        // object identifier
  public:
    virtual void receive(Data_Comm_Object *);  // entry point

    Comm_Object();                             // constructor
    virtual ~Comm_Object();                    // destructor
}
```

Figure 3.7: Partial definition of the `Comm_Object` class

As for the creation of a cluster object, there are two ways to give the location of a new communicating object: either by specifying to the allocator the target cluster, or by default, the new object will be mapped on the "best" cluster (see Figure 3.8).

3.3.2 Communication

Sending data to a communicating object is done via its proxy (see Figure 3.8). Nothing special is required concerning these data. They just have to be embedded in an instance of a class which

```
Proxy_Comm_Object *p1, *p2;
p1 = new(cl1) Proxy_Comm_Object();   // on specific cluster
p2 = new Proxy_Comm_Object();        // on default cluster

Data_Comm_Object_Int *d = ...
p1→send(d);                          // Data are transparently embedded in an active
                                     // message and sent to cluster cl1
```

Figure 3.8 : Creation of communicating objects

inherits from the Data_Comm_Object class (see Figure 3.9).

```
class Data_Comm_Object {
   public:
      virtual Buffer *flat(Buffer *buff)=0;
      static Message *rebuild(Buffer *buff, Data_Comm_Object *d=0);
}
```

Figure 3.9 : Partial definition of the Data_Comm_Object class

These data are automatically sent by an active message, whose `process()` function triggers the call of the recipient `Comm_Object::receive(Data_Comm_Object *)` function. Thus, the user never has to call this last function.

4 SCHOONER as a run-time support for C++//

A C++// application obliviously needs to be supported by a set of clusters running on choosen computers. At initialisation time, it suffices to create SCHOONER clusters and computers. Moreover, C++// active objects will be easily implemented by SCHOONER mono-active communicating objects.

Currently, active objects of C++// are mapped at their creation according to a C++// `Mapping` object which essentially gives either the symbolic name of a computer, or tells with which already existing C++// active object to map the new active object. More sophisticated C++// mapping strategies could use or redefine the `best()` functions defined on the SCHOONER `Cluster` and `Computer` classes. For example, the `Cluster::best()` function could return the cluster which contains the fewest communicating objects.

C++// active objets are implemented by SCHOONER communicating objects as follows:

- The constructor of an heir of the `Comm_Object` class has to launch a thread, whose associated function is the `Live()` routine of the `Process` class the active object belongs to.
- The `Proxy_Comm_Object` class can be derived to implement the concept of C++// active object proxy. One instance of this class stands as a reference to an active object.
- C++// requests and replies are instances of `Data_Comm_Object` sub-classes.

- The three different communication modes of `C++//` active objects are programmed in the `send()` function of an heir of the `Proxy_Comm_Object` class and in the `receive()` function of an heir of the `Comm_Object` class. In case of the proactive mode, the receive function will only put the message in a queue. The active object will extract them on demand and put them in its request or reply list. For the two reactive modes,we have to trigger this extraction operation without the knowledge of the active object[1]. We thus need a mechanism to interrupt the current activity of this object.

As SCHOONER does not by itself include communication and multi-threading facilities - it is a set of bindings to those types of packages - we need to interface one communication and multi-threading library to eventually support a `C++//` run-time. All the SCHOONER binding points to such a package have been grouped in the SCHOONER `System` and `Thread` classes, so the use of one or another package will only have incidence on those two classes. A first prototype has been quickly obtained using PM^2. The next section briefly describes PM^2 and Section 6 shows how the `System` and `Thread` classes implementation is achieved.

5 The PM^2 environment

PM^2 [10] (Parallel Multithreaded Machine) is an environment for distributed architectures providing a computing model based on lightweight processes. It's a portable platform that allows the exploitation of heterogeneous configurations. PM^2 was designed on top of a lightweight process package (called MARCEL) and a communication library (PVM), as illustrated in Figure 5.1.

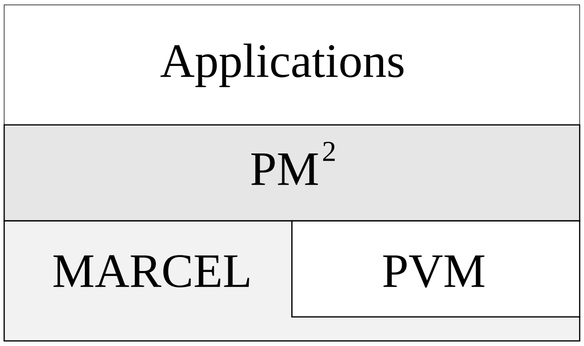

Figure 5.1 : PM^2 was designed on top of Marcel and PVM

MARCEL is the user-level thread package designed and implemented at LIFL. It provides a large subset of the POSIX standard thread interface primitives with some additional features. The MARCEL scheduling policy is preemptive, which means that a clock forces context switches between threads at some (customizable) frequency. Although this policy has some drawbacks (code reentrancy problems, context switching overhead), it has the main advantage to allow the setting of a fair priorities management (without any risk of starvation).

[1]The queue length is thus never greater than 1.

The PVM communication package was only used for implementation purposes, and thus its interface is not part of the PM^2 interface.

The PM^2 programming model is mainly based on the LRPC mechanism. A LRPC (Lightweight Remote Procedure Call) consists in creating a thread on a remote processor to execute a specified *service*. This mechanism was chosen because of its suitability to irregular applications support.

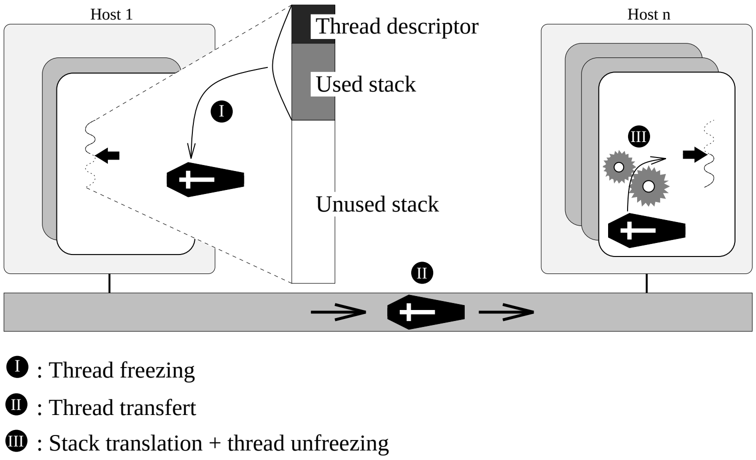

Figure 5.2: The PM^2 migration mechanism

PM^2 also provides a thread migration mechanism which allows to move threads between processors during their execution. This mechanism, triggered by a call to the `pm2_migrate()` function, is done in three steps (Figure 5.2):

1. The thread is frozen, and both its descriptor and the useful part of its stack are packed in a buffer;

2. Then, the buffer is sent to the remote processor;

3. Last, the thread's descriptor and stack are unpacked on the remote processor, the stack is relocated in a new address space, and the thread is unfrozen.

The PM^2 environment and programming interface are available on the following systems: ALPHA/OSF-1, Sparc/Solaris, Sparc/SunOs, PowerPC/Aix, PC-Linux. The whole package can be downloaded from http://www.lifl.fr/~namyst/pm2.html. Several French laboratories are using and experimenting PM^2 (LIP-Lyon [11], PR*i*SM-Versailles [8]). In this paper, we are focussing on the usage of PM^2 as an execution support for SCHOONER and indirectly `C++//`.

6 SCHOONER on top of PM^2

6.1 `System` class

A cluster is implemented by a PM^2 module (use of the `pm2_start_modules(..., host, ...)` primitive). Because no facility to add hosts to the parallel virtual machine is provided by PM^2, we simply use the PVM `pvm_addhost()` primitive, in order to create SCHOONER `Computer` instances.

Active messages for inter-cluster communications, and indirectly supporting inter-communicating object communications, are all implemented by a `QUICK_ASYNC_LRPC` service bind to a function called `RECV_MESSAGE()`. In order to send a message to a cluster, the sender cluster invokes this service. The PM^2 run-time support for LRPCs automatically conveys the data using PVM for instance, and triggers the `RECV_MESSAGE()` service on the destination cluster whose aim is: create an instance of the corresponding `Message` sub-class the data belong to, then launch the `rebuild()` and `process()` functions defined on this sub-class.

As `Data_Comm_Object` instances are conveyed by instances of an heir of the `Message` class, inter-communicating object communications are supported in this same way.

6.2 `Thread` class

To create an instance of a mono-active communicating object (which supports a `C++//` active object), we use the Marcel `pthread_create()` primitive in the constructor .

To deviate the activity of a communicating object, in order to implement the two reactive interprocess communication modes of `C++//`, we use the Marcel `pthread_deviate()` primitive whose handler is the "extraction operation" described in Section 4. Care must be taken when interrupting such an activity: the object can be currently engaged in any C++ library function call; so if this library is not re-entrant, the object does not correctly resume its execution. We have been unable to solve this point in general. Except using a re-entrant version of the C++ library[2], an object can temporarily desactivate the preemptive mode of the Marcel scheduler each time it enters a critical code section. But this dramatically precludes reusability of C++ programs in the context of `C++//`.

Finally, the PM^2 capability of migrating threads will be on interest for moving active objects between clusters. This could be useful for `C++//` applications such as the PROSIT [9] distributed discrete-event simulations environment developped in *SLOOP*. We have not yet experimented with this functionality.

7 Conclusion

We have presented in this paper how a SCHOONER implementation on top of the PM^2 environment can easily be used as a `C++//` run-time. This work has mainly enabled the validation of the design of

[2]Not provided on SUNOS for example.

this C++// run-time.

Bibliography

[1] D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[2] D. Caromel, F. Belloncle, and Y. Roudier. *Parallel Programming Using C++*, chapter The C++// System. G. Wilson ed., MIT Press, 1996. To appear.

[3] D. Caromel, P. Dzwig, R. Kauffman, H. Liddell, A. McEwan, P. Mussi, J. Poole, M. Rigg, and R. Winder. EC++ – EUROPA Parallel C++ : A Draft Definition. In *High-Performance Computing and Networking (HPCN'96)*. Springer, LNCS, 1996. 848-857, Brussels, Belgium, April.

[4] Ian Foster, Carl Kesselman, and Steven Tuecke. The Nexus Task-parallel Runtime System. In *Proceedings of 1st International Workshop on Parallel Processing*, 1994.

[5] Nathalie Furmento and Françoise Baude. Schooner: An Oriented-Object Runtime Support for Distributed Applications. Technical Report 95-50, I3S, 1995.

[6] R. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, October 1985.

[7] Oliver Krone, Marc Aguilar, and Béat Hirsbrunner. Pt-pvm : Using PVM in a multi-threaded environment. In Dongarra and al., editors, *Proceedings of Euro-PVM'95*, volume 5 of *Parallélisme, Réseaux et Répartition*, pages 83–88. HERMES, September 1995.

[8] Bertrand LeCun, Mohamed Benaichouche, Van-Dat Cung, and Catherine Roucairol. Towards an unified or and ai exact methods library. In *Proceedings of Parallel Optimization Colloquium 96*, pages 37–46, Versailles France, 1996.

[9] Philippe Mussi and Günther Siegel. The Prosit Sequential Simulator: A Test-bed for Object Oriented Discrete Event Simulation. In *European Simulation Symposium*, pages 297–301, Erlangen, Germany, October 1995.

[10] Raymond Namyst and Jean-François Méhaut. PM2: Parallel Multithreaded Machine. A computing environment for distributed architectures. In *ParCo'95*, Gent, Belgium, September 1995.

[11] Christian Perez. Utilisation de processus légers pour l'exécution de programmes à parallélisme de données : étude expérimentale. Technical Report 96-09, LIP, 1996.

[12] Thorsten von Eicken, David E. Culler, Seth C. Goldstein, and Klaus E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architectures*, Gold Coast, Australia, May 1992. ACM Press.