Programming with Object Groups in PHOENIX*

Pascal Felber Rachid Guerraoui
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
CH-1015 Lausanne, Switzerland
felber@lse.epfl.ch *rachid@lse.epfl.ch*

Abstract

PHOENIX is a toolkit for distributed programming with groups in large-scale distributed systems. The PHOENIX programming interface is object-oriented. It consists in an extensible class library of group management and group communication abstractions, designed with a particular concern for modularity and reusability. By supporting groups of abstract objects rather than groups of operating system processes, PHOENIX offers a higher abstraction level than existing comparable toolkits. In this paper we describe the PHOENIX programming interface and we present a small example to illustrate its use.

1 Introduction

1.1 Programming with Groups

Many applications require an explicit group notion to gather entities and to provide one-to-many communication structures, i.e. multicasts. Among these applications are, for example, replication and cooperative editing. Replication is very useful to tolerate failures in a distributed system. A file is more likely to tolerate failures if it is replicated on different nodes of a network. The set of the file replicas can be viewed as a group maintaining the file's state and reliable atomic multicasts can be used to update the replicas. The aim of a cooperative editing application is to facilitate the development of a document by a set of participants. Hence groups and multicast communications are useful for information dissemination. Each participant works on its local part and multicasts the modifications to the group of participants.

*This work has been supported in part by the Commission of European Communities under ESPRIT Programme Basic Research Project 6360 (BROADCAST).

1.2 Related Work

The V system was the earliest system to offer an explicit notion of group and multicast communication [Cheriton 85]. Its design influenced most existing group-based systems.

The Isis system extended the group model of the V system by providing support facilities for fault-tolerance such as process group membership, reliable totally ordered multicast, reliable causally ordered multicast, etc. [Birman 91, Birman 93]. The Isis group membership service ensures that every non-faulty process, member of a group G, receives periodically a *view* of G describing G's current members. The Isis model, called *virtual synchrony*, ensures that all members of a group receive the same sequence of *views* and guarantees that messages are totally ordered with respect to view changes. Communications are said to be *view synchronous.* The Amoeba system [Kaashoek 91] also offers reliable multicast and totally ordered multicast but does not provide the full range of fault-tolerance possibilities provided in Isis, e.g. delivery of views.

The weakness of both Amoeba and Isis is that they do not provide a structured way of modeling applications. Their programming interface consists in flat sets of heavy-weight[1] process group management and communication primitives. More recently, the Transis [Amir 92] and Horus [Robert 92] toolkits followed the Isis approach to fault-tolerance. They provide in addition a light-weight[2] group concept. However, no structuring facility is implemented.

1.3 Towards an Object Oriented Approach

PHOENIX also follows the Isis approach by providing a wide range of group-oriented fault-tolerance supports [Malloth 94]. However, while designing PHOENIX, we concentrated on defining a structured application interface with a high abstraction level.

Our main motivation was to build a modular and reusable system. To achieve this goal, we have adopted an object-oriented approach (in the sense of Wegner [Wegner 87]). The set of application services offered by PHOENIX consists in an extensible class library. In addition, we have provided a higher abstraction level than the one found in comparable existing systems (such as Isis, Horus and Transis) by grouping passive and active objects no matter how they are implemented, i.e. whether they are light-weight threads or heavy-weight processes. Finally, by distinguishing the different roles of group members, PHOENIX goes further towards modularity by easying the way of structuring applications and addressing efficiently large-scale distributed systems [Babaoglu 94]. The current prototype of PHOENIX is implemented in C++, on top of a network of Unix Sun workstations. It can be used in a stand alone way, or as an underlying support of a programming environment such as GARF [Garbinato 94].

In this paper we focus on the object-oriented programming interface of PHOENIX. Other aspects such as group membership and view synchronous communication are described in [Malloth 94]. The rest of the paper is organized as follows. Next Section briefly presents the main concepts of the model and the architecture of PHOENIX. Section 3 describes the PHOENIX programming interface. Section 4 presents a small example of application and Section 5 discusses some implementation issues. Section 6 concludes by recalling the main aspects developped in this paper.

[1]Processes in Isis and Amoeba are typically Unix processes.
[2]Processes in Horus for example can be light-weight threads.

2 Overview of PHOENIX

2.1 The Model

PHOENIX can be viewed as a toolkit providing group management and group communication primitives for writing distributed fault-tolerant applications in large scale systems.

Whereas traditional group-based systems define a single type of membership [Amir 92, Birman 93, Cheriton 85, Kaashoek 91], i.e. a process is either member of a group or not, PHOENIX distinguishes three different types of members based on their role. As we will see in Section 4, this distinction contributes to application modularity. The three roles are sketched below and described in more details in section 3.

(1) *Core members* — shortly called *members* — manage shared state and have the strongest reliability guarantees with respect to message delivery and membership changes [Guerraoui 94]. (2) *Clients* interact with members in order to direct requests to them more efficiently. An interaction between a client and a member is more efficient than one between two members since the former offers weaker reliability guarantees than the latter. Finally, (3) *sinks* only receive diffused information regarding the shared state maintained by the core. As suggested by their name, sinks can not perform requests and only receive messages from the members.

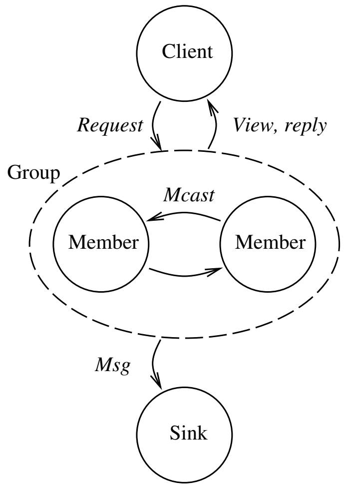

Figure 1: Members, clients and sinks

Figure 1 illustrates the main messages exchanged by members, clients and sinks. Members basically communicate within the same group through reliable multicasts. Current group membership, transmitted by *view-change messages*, is known at each instant by members and clients. Sinks only receive messages from the group they have joined[3].

While multicasts between members offer reliable communication, messages exchanged with clients and sinks are *best-effort* communication. With respect to various costs, members can be seen as "heavy-weight" objects whereas clients and sinks are rather "light-weight" ones. In Section 4 we illustrate these characteristics on a simple example.

2.2 The Architecture

PHOENIX has been developped following a layered architecture, as shown in figure 2. Reliable communication is performed by the bottom layer (layer 1). View-synchronous communication and ordering primitives like *total-order delivery* and *uniform delivery* are handled by layer 2. Core members rely on the strong view-synchronous semantics for internal group communication and request/reply interactions with clients.

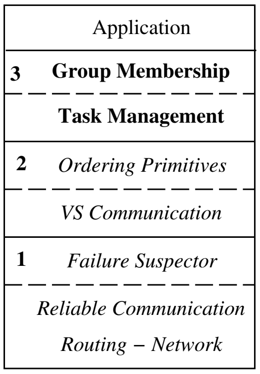

Figure 2: Architecture

In the following we describe layer 3 which constitutes the PHOENIX object-oriented programming interface. This layer provides a built-in library of classes called *application services*, that deals with *group membership* (i.e. members, clients and sinks) and *tasks* (i.e. thread management).

[3]To be more explicite, members and clients can send messages to members, clients and sinks. View-changes are received by members and clients. Only members can send and receive multicasts.

3 PHOENIX Library

The PHOENIX programming interface is a class library. The main classes offered to the end user are: `Sink`, `Client`, `Member` and `Task`. In our current prototype, these classes are implemented in C++ and use Unix inter-processes communication primitives (see Section 5). Instances of `Sink`, `Client`, `Member` or one of their subclasses can be gathered inside groups and can perform remote communications.

3.1 Sink Objects

Instances of the class `Sink` (or one of its subclasses) are called sink objects. After having successfully joined a group[4], a sink object will eventually receive messages from the group. Since its information concerning the group is not necessarily up-to-date, the sink does not receive any view-change from the group. It can become a sink member of one or more groups. The following class interface represents the main operations that enable a sink to join or leave a group, and to receive information from a group.

```
class Sink: public {
  // ...
public:
  Sink();
  Sink(GroupID group);
  ~Sink();

  void SinkJoin(GroupID group);
  void SinkLeave(GroupID group);
  void Receive(Message msg);
  // ...
}
```

[4]When talking about sink objects, *join* means *to become sink member*.

3.2 Clients Objects

Instances of the class `Client` (or one of its subclasses) are called client objects. After having successfully joined a group[5], a client object will send requests and receive view-changes from the group. It can become a client member of one or more groups, and can also be the sink of any group. The following class interface represents the main operations that enable a client to join or leave a group, to send messages and receive view-changes from a group.

```
class Client : public Sink {
  // ...
public:
  Client();
  Client(GroupID group);
  ~Client();

  void ClientJoin(GroupID group);
  void ClientLeave(GroupID group);

  void Send(IDList dest,
            Message msg);
  void Request(PObjID dest,
               Message msg);
  void Request(GroupID group,
               Message msg);

  void ViewChange(Group grp);
 // ...
}
```

3.3 Core Member Objects

Instances of the class `Member` (or of one of its subclasses) are called core member objects. Communication between the core members (or simply members) is performed by view synchronous multicasts, i.e changes to the group composition have ordering guarantees with respect to message delivery. Members receive all the view-changes from the group to which they belong, just like clients do. One can't be a core member of more than one group, but a member can be the client or the sink of many groups. The following class interface represents the main operations that enable a member to join or leave a group, and to send multicasts.

```
class Member : public Client {
  // ...
public:
  Member();
  Member(GroupID group);
  ~Member();

  void Join(GroupID group);
  void Leave();
  void MCast(Message msg);
  // ...
}
```

Sinks are the most general objects, with the strongest restrictions; clients have a few more properties than sinks; finally, members are the most specific objects. The inheritance hierarchy of the corresponding classes is illustrated by figure 3.

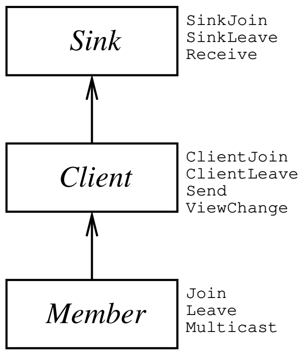

Figure 3: Members, clients and sinks inheritance hierarchy

3.4 Tasks

In PHOENIX, a task is an instance of the `Task` class or of one of its subclasses. It has a specific operation `Body` performed during all the task object's life time. The interface of the `Task` class is outlined below. In the current PHOENIX prototype, tasks are implemented with POSIX "light-weight" threads (see Section 5).

[5]When talking about client objects *join* means *to become client member*.

```
class Task {
  // ...
public:
  Task();
  ~Task();

  virtual void *Body() = 0;
  void Start();

  // Task management
  int Waitfor(void **status);
  int Detach();
  int Kill(int signal);
  int Cancel();
  // ...
}
```

A frequent use of members, clients and sinks is to create derived classes which also inherit (through the *multiple inheritance* mechanism) from the `Task` class[6]. This creates "active" objects which can perform the background operation `Body`. The latter can be customized for each subclass.

4 Application Example

We illustrate the use of our application library by applying it to the implementation of a bank service. Money can be deposited or withdrawn on a particular account from almost any bank. The information about the accounts is replicated on many servers to ensure its availability. If an error occurs or if the servers are partitionned, the information might not be the same in all the replicas. In that case, one could even withdraw all the money from an account more than once in bank offices belonging to different partitions. To avoid such undesirable[7] behavior, operations that change the state of the accounts must have strong delivery guarantees. Consulting an account doesn't require to have the latest information available and can allow weaker garantee. If a withdrawal is just being performed on an account, consulting a local replica that has not been already updated does not lead to an inconsistent state between servers.

[6] Actually through C++ multiple inheritance.

[7] At least for the bank.

In the PHOENIX model, the servers will build a group — let's call it *G*. Depositing or withdrawing money requires operation consistency within the whole group. To perform such operations, one needs to join *G* as a client member. Agreement is performed among the members of *G* before validating a deposit or a withdrawal. If the operation succeeds, PHOENIX ensures that every member of *G* has either handled the request or has left the group. Consultations are made on local databases which are regularly updated by the members of *G*. These databases are declared as sink members of *G*. They only receive stable and consistent information, but there is no guarantee concerning delivery — we do provide best-effort communication outside groups. In our example, consultation of local databases takes place in local consultation points (*LCP*). Databases could also be accessible through data communication services. The bank system is illustrated by figure 4.

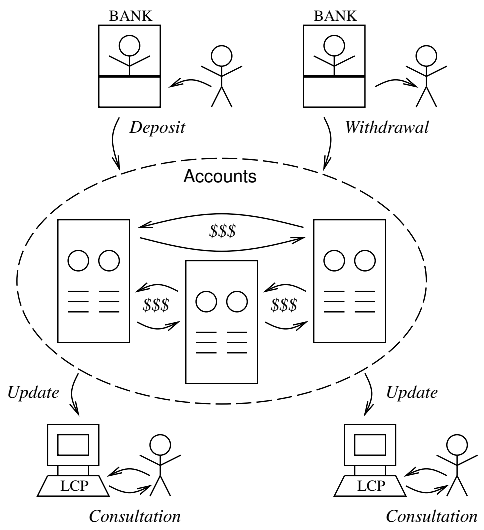

Figure 4: Bank system

The structure of local consultation points is described by the following class interface:

```
Class LCP : public Sink,
            public Task {

public:
  LCP();
  ~LCP();
```

```
  // Overridden functions
  void Receive(Message msg);
  void Body();
  // ...
}
```

Since a specific task is needed to allow interaction with the user, the `LCP` class inherits from the `Task` class (see figure 5). The main task to be executed is the `Body` operation. In this operation, a *customer* first becomes a sink member of each bank group he wants to consult and then starts the account consultation.

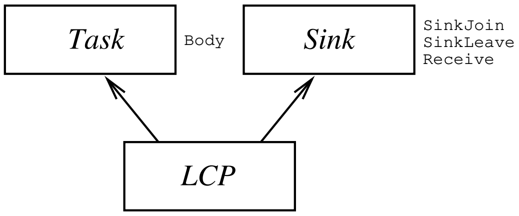

Figure 5: LCP Class Tree

Each time an object of the `LCP` class receives a new message, the `Receive` operation is invoked (by PHOENIX). This operation treats incoming messages and stores information relative to the accounts.

The interface of the class required for deposits and withdrawals is the following:

```
Class BankAgent  : public Client,
                   public Task {

public:
  BankAgent();
  ~BankAgent();

  // Overridden functions
  void Receive(Message msg);
  void ViewChange(View newView);
  void Body();
  // ...
}
```

The main part of the `Body` function consists in joining a group as a client, sending requests, waiting for the answers and finally leaving the joined group. The `Receive` operation analyses incoming messages and possibly finds out answers to specific requests. The `ViewChange` operation is invoked (by PHOENIX) whenever a change in the membership of the group occurs. This operation can be used to perform some action according to the new composition of the group.

The interface of the core members class, maintaining the global state of all the accounts is the following:

```
Class BankDataBase : public Member {

public:
  BankDataBase(GroupID group);
  ~BankDataBase();

  // Overridden functions
  void Receive(Message msg);
  void ViewChange(View newView);
  // ...
}
```

This class does not inherit from the `Task` class since it does not perform any background operation. The `ViewChange` method can be used in members to start a new server each time one crashes or disappears from the group.

We believe that the main classes of the PHOENIX programming interface (`Sink`, `Client`, `Member` and `Task`) provide a convenient way to describe the simple banking application in a modular way. Such modularity can be very helpful (if not necessary) in more complex fault-tolerant distributed applications.

The class library can be extended (through inheritance) to offer additional functionalities. For example, one can define new types of members which would be represented as new classes within the inheritance hierarchy.

5 Implementation

5.1 General Architecture

In PHOENIX, the low level layers (1 and 2 in figure 2) and the application interface layer (3) are implemented by separated processes. The process implementing layers 1 and 2 is called PHOENIX *daemon*. There is one daemon on each participating site (i.e computer) in the PHOENIX system. The daemon is responsible for the site state: if the daemon fails, the site is considered as having failed. Every message coming from and addressed to an application is handled by the daemons. This approach has several advantages.

- Applications are smaller.
- Speed can be improved by using only site-to-site communication and not overloading the network with direct application messages.
- The same application will run with new versions of the daemon without recompilation.

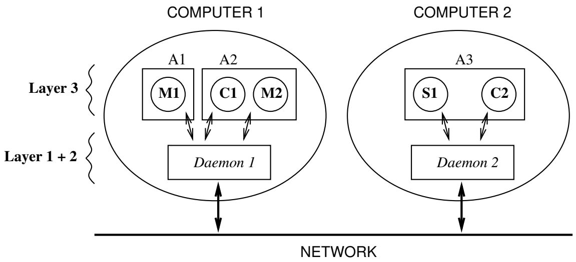

Figure 6: Tasks, processes and sites

Figure 6 describes the interactions between different components in the PHOENIX prototype. A1, A2 and A3 represent three Unix processes, called PHOENIX application processes. Each application process holds a set of members, clients or sinks (noted M1, C1, S1, etc.) and a set of tasks. There is one PHOENIX daemon on each site, i.e. on each computer. In the following, we bring to the fore some implementation features of layer 3.

5.2 Sinks, Clients and Members

Subclasses from `Member`, `Client` and `Sink` will generally have to override the `Receive` and `ViewChange` methods which are called by PHOENIX. On creation members, clients and sinks can optionaly perform an implicit join to a group by using one of the provided constructors. They keep an intern trace of each group they have joined for each membership type and implicitely leave these groups on destruction. Each class also provides specific operations like, for example, the `Request` method of the `Client` class which sends a request[8] to a group or a group member and waits for the reply. A default behaviour is assumed for most operations so that the user only overrides the relevant functions.

Instances of `Member`, `Client` and `Sink`, or of one of their subclasses, are uniquely designated with identifiers of the class `PObjID`. These identifiers are used to access distant objects with the PHOENIX primitives. The `Group` class is an abstraction for real groups. It contains the list of all the members of the group, its identifier and other information. Group identifiers are objects of the `GroupID` class. Resolving a group name into an identifier requires communicating with a dedicated nameserver. Nevertheless, the class `GroupID` provides a constructor which performs automatic conversion of group names into identifiers. Views are univocally identified in the system. They are represented by objects of the `ViewID` class.

Since we had to deal with lists of identifiers — for instance when sending a message to a list of objects — we have introduced a class `IDList` which provides standard list-handling functions such as insertion, removal and iteration. These lists store identifiers of the abstract `ID` class, which is the base class of `PObjID`, `GroupID` and `ViewID` (see figure 7). This offers a convenient way to work with sets of identifiers whatever is their type.

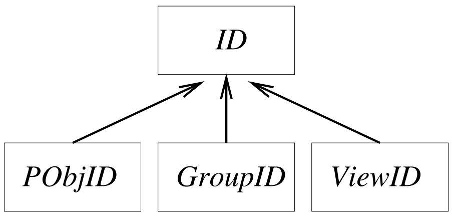

Figure 7: Identifiers hierarchy

[8] A request is a simple message issued by the primitive `Send`.

5.3 Tasks

In our system, tasks are built on the top of a library implementation of POSIX threads [Mueller 93, POSIX1003.1c 94] which provides pre-emptive threads, convenient synchronization mechanisms, thread-level signal handling, priority scheduling, thread specific data and some other functionalities. One task is associated to one single flow of execution which is created at the same time as the task object. All tasks in a process have the same addressing space and data protection is only based on the mechanisms provided by C++. The main function of a task is placed in the `Body` method of the `Task` class, which is declared as pure virtual so that subclasses must override it. After creation, the flow of execution associated to the `Task` object is in a *blocked* state. It then requires then an explicit call to unblock it[9]. This special function — called `Start` — is invoqued before any other call to the operations of the task object and leads to the execution of `Body`.

6 Summary

PHOENIX is a toolkit for distributed programming with groups in large-scale distributed systems. It provides fault-tolerance services for group management and group communication and offers various reliability guarantees.

To provide modularity and reusability, we designed the PHOENIX programming interface as a class library of group management and group communication services. The main abstractions provided by the library correspond to different types of members: sinks, clients and core members. This membership distinction is a specific characteristic of PHOENIX and has been designed to help the programmer specifying clearly, and in a modular way, the functionalities and the needs of its application.

Every object in PHOENIX can hold a specific thread which is executed during all the object life-time. This behavior is inherited from a built-in class representing tasks. As a consequence, sinks, clients and members can either be passive objects or active objects.

[9] This is due to implementation matters with C++. Some of these problems are evoqued in [Buhr 92].

Acknowledgments

The PHOENIX architecture has been designed by C. Malloth, A. Schiper and U. Wilhelm. Discussions with B. Garbinato and K. Mazouni have been helpful in designing the class library of group management and group communication services.

References

[Amir 92] Y. Amir, D. Dolev, S. Kramer, and D. Malki - Transis: A communication subsystem for high availability - Proc of the International Symposium on Fault-Tolerant Computing - pp 76.84 - 1992.

[Babaoglu 94] O. Babaoglu and A. Schiper - On Group Communication in Large Scale Distributed Systems - ACM Proc of the European SIGOPS Workshop - pp 17.23 - 1994.

[Birman 91] K. Birman, A. Schiper, and P. Stephenson - Lightweight causal and atomic group multicast - ACM Transactions on Computer Systems - Vol 9, Num 3, pp 272.314 - 1991.

[Birman 93] K. Birman and R. van Renesse - Reliable Distributed Computing with the Isis Toolkit - IEEE publisher, K. Birman and R. van Renesse editors - 1993.

[Buhr 92] P. Buhr and G. Ditchfield - Adding Concurrency to a Programming Language - Proc of the C++ Usenix International Conference - pp 207.223 - 1992.

[Cheriton 85] D. Cheriton and Willy Zwaenepoel - Distributed process groups in the V kernel - ACM Transactions on Computer Systems - Vol 3, Num 2, pp 77.107 - 1985.

[Garbinato 94] B. Garbinato, R. Guerraoui, and K. Mazouni. *Distributed Programming In GARF*. In Object-Based Distributed Programming. Springer Verlag (LNCS 791) pubisher, R. Guerraoui, O. Nierstrasz and M. Riveill editors - pp 225.240 - 1994.

[Guerraoui 94] R. Guerraoui and A. Schiper - Transaction model vs. virtual synchrony model: bridging the gap - Technical Report No 94/62 - LSE/DI/EPFL - 1994.

[Kaashoek 91] F. Kaashoek and A. Tanenbaum - Group Communication in the Amoeba Distributed Operating System - IEEE Proc of the International Conference on Distributed Computing Systems - pp 222.230 - 1991.

[Malloth 94] C. Malloth and A. Schiper - View Synchronous Communication in the Internet - Technical Report 94/84 - LSE/DI/EPFL - 1994.

[Mueller 93] F. Mueller - A Library Implementation of POSIX Threads under UNIX - Proceedings of the USENIX Conference - pp 29.42 - 1993.

[POSIX1003.1c 94] IEEE - Threads Entension (P1003.1c, Draft 9) - 1994.

[Robert 92] R. van Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson - The Horus System - In Reliable Distributed Computing with the Isis Toolkit - IEEE publisher, K. Birman and R. van Renesse editors - pp 133.144 - 1992.

[Wegner 87] P. Wegner - Dimensions of Object-based Language Design - ACM Proceedings of the International Conference on Object-Oriented Programming Systems, Languages and Applications - pp 168.182 - 1987.