BOAR: A Library of Fragmented Object Types for Distributed Abstractions

Mesaac Makpangou, Yvon Gourhant, Marc Shapiro

BOAR: A Library of Fragmented Object Types for Distributed Abstractions

Mesaac Makpangou
Yvon Gourhant
Marc Shapiro

INRIA, B.P. 105, 78153 Rocquencourt Cédex, France

Abstract

We present BOAR, a forest-structured library of predefined Fragmented Objects (FOs) encapsulating commonly used distributed abstractions. We identify three kinds of interaction FO types (channel, sharing, and synchronization), which can be beneficially included in BOAR. A channel FO type encapsulates a remote or group invocation protocol. A sharing FO type encapsulates a sharing mechanisms such as replication or partition. Finally, a synchronization FO type encapsulates a synchronization mechanism. A FO programmer picks-up from BOAR, the FO types implementing the mechanisms which fits best its needs.

1 Introduction

The object-oriented programming methodology is chiefly about re-use: one can design high level object types that are then included in libraries for future re-use [2, 3]. Despite recent progress in extending the object programming methodology in a distributed environment, most existing object-oriented libraries lack support of distributed abstractions such as remote invocation, multicasting, memory sharing, and distributed locking. Consequently, a programmer of a distributed application has to implement on his own, the distributed mechanisms that he needs. This makes designing a distributed application tedious and complex. Furthermore, this solution is error prone and does not encourage the re-use of existing and/or efficient code. To facilitate designing distributed applications, we propose the *fragmented object* model, along with its programming tools amongst which, BOAR, a library of object types encapsulating commonly used distributed abstractions.

Fragmented Objects (FOs) extend the object programming methodology in a distributed environment [4]. A FO provides two levels of abstraction. Externally, for its clients, a FO is a shared object. Internally, for its designer, it is a set of fragments, communicating through lower-level shared FOs called *connective objects*.

Typically, a connective object encapsulates a communication protocol (e.g., RPC or parallel-RPC), a specific sharing abstraction (e.g., replication, migration and partition), or a specific synchronization mechanism (e.g., locking, semaphore, rendez-vous and token passing).

BOAR provides to a FO designer, connective FO types encapsulating various implementations of commonly used abstractions necessary for structuring distributed applications. A FO designer can address critical distribution issues such as load-balancing, protection and fault-tolerance, just by picking-up from BOAR the connective FO types implementing the abstractions that he needs. In general, all connective FO types implementing the same abstraction have the same public interface. This allows a FO designer to switch easily from one implementation to another. Furthermore, one goal is to facilitate the design of application FO types by reusing centralized components as such.

The rest of the paper is organized as follows: Section 2 presents briefly the FO model. Section 3 reviews distributed abstractions commonly used in structuring distributed applications. It identifies the core set of distributed mechanisms that are implemented by generic FO types included in BOAR. Finally, Section 4 concludes the paper.

2 Fragmented Object Model

As is common in object-oriented approach, a FO can be viewed at two different levels of abstraction, corresponding respectively to a client's (abstract) view, and to the designer's (concrete) view.

Abstractly (for its clients), a FO is a single shared object. It is shared by several client objects, localized in different address spaces, possibly on several sites. It is accessed via a strongly-typed interface. A FO can offer distinct interfaces to different clients.

Concretely (for the designer), a FO encapsulates a set of cooperating *fragments*. Each fragment is an elementary object (i.e., with a centralized representation).

We distinguish three fragment interfaces: the private interface, the public interface, and the group interface.

The *private* interface of a fragment is composed of internal methods, accessible only from within the fragment.

The *public* interface contains methods accessible by clients. The abstract interface of a FO is provided to some client via the public interface of a local fragment, that can be invoked locally. A client sees no difference between a fragment implementing the interface exported by the FO to this client, and the FO itself. The public interface of the fragment offers transparency of the distribution to a client. A method of the public interface can be entirely implemented by the fragment itself, or it can trigger invocations to other fragments.

The *group* interface comprises those methods which are internal to the FO as a whole (i.e., which can be invoked remotely from other fragments). The group interface concept extends type-checking to remote communications.

The fragments cooperate using *connective objects*. A connective object is just another FO at a lower level of abstraction. The most primitive connective objects are the communication objects implemented by the system (for instance, communication protocols).

The types of parameters in the group interface of a FO may be totally unrelated to the interface of its connective object. A compiler generates automatically marshalling/unmarshalling code for methods of the group interface.

The mains benefits of the FO approach are:

- Separation of interface from implementation. Since, a FO is a distributed object, it is easy to switch between different *policies* for the same *mechanism*.
- Strongly type-checked remote communications.
- Reusability of programmer-defined distributed shared objects.
- Support for different levels of transparency.
- High-level communication abstractions.
- Support for concept of layered protocols. Marshalling/unmarshalling conversions between these multiple layers are handled automatically and in a type-safe manner.

The FO model suits well for the object structuring of distributed abstractions. However, for application programmers to really benefit from this model, it is necessary to provide them with specific tools such as a library of predefined FO types implementing the commonly used distributed abstractions. BOAR is such a library.

3 Predefined Connective FO Types

BOAR, (french acronym for Library of FO Types for Distributed Abstractions), is a forest-structured library of connective FO types. We distinguish mainly three classes of connective FO types: *channel*, *sharing* and *synchronization* connective FO types. Examples of FO types of these classes are successively presented in Sections 3.1, 3.2 and 3.3.

3.1 Channels

A *channel* permits fragments of a same higher-level FO using it as a connective FO, to communicate according to the client/server model. A fragment of the higher-level FO can plays the *client* role (i.e. it can invoke a method of the group interface implemented by another fragment), the *server* role (i.e. it implements a method of the group interface and can therefore be the target of a remote invocation from another fragment), or both roles.

Each fragment of the higher-level FO communicating through a channel is bound with a fragment (or endpoint) of this channel. The channel endpoint bound with a client fragment offers it an interface to invoke its server counterparts; we will refer to this interface as the channel interface. The endpoint bound with a server fragment is essentially in charge of relaying invocations.

There are two types of channel interface: a RPC-like interface for point-to-point channels, and a parallel-RPC-like interface for multipoint channels.

Channels implementing the same interface are organized hierarchically. The following two sections describe these two hierarchies of channels.

3.1.1 Point-to-point Channels

A point-to-point channel is characterized by its semantics. Typically, a channel can enforce the *maybe, at-least-once, at-most-once, or exactly-once* semantics [5].

A point-to-point channel is also characterized by the type of the synchrony that it enforces. We distinguish three kinds of synchrony: request/reply (upon a call, the client is blocked until the receipt of the reply); synchronous (upon a call, the client is blocked until the invocation message is delivered to the server); and asynchronous where a client doesn't block at all.

Finally, a point-to-point channel is characterized by the ordering in which invocations and replies are delivered to their destinations. We distinguish two ordering policies: datagram and FIFO.

No one of the cited semantics, synchrony or ordering policy can satisfy all distributed applications. Hence, in order to accommodate different applications, we provide an hierarchy of point-to-point channels implementing various combinations of these characteristics. However, to allow application programmers to switch from one implementation to another, all point-to-point channels have the same public interface.

The root point-to-point channel type is rpcChannel. A rpcChannel has two kinds of fragment types: channel and channelStub, associated respectively with a client and a server. A FO of type rpcChannel implements both blocking and non-blocking remote procedure call. It enforces the exactly-once semantics. It is datagram oriented (i.e. all calls and replies handed to this channel are eventually delivered to their destination, not necessary in their order sent).

Another is the a fifoChannel channel. It differs from the basic one in that, invocations are delivered to the server in FIFO order.

3.1.2 Multi-point Channels

A multipoint channel implements a specific multicast protocol between fragments of a same FO. To simplify the addressing issue, all server fragments constitute a group. The group paradigm permits a client fragment to address all its server counterparts as a whole.

A multipoint channel is characterized by the quality of the service provided by its underlying multicast protocol (e.g., atomic, causal, fifo, or datagram ordering). It is also characterized by the management and the monitoring of its server group. Like for the point-to-point channel, we have an hierarchy of multipoint channel, each implementing a specific multicast protocol, a specific group management and monitoring policy.

The root multi-point channel is multicastChannel. A FO of type multicastChannel has two kinds of fragments: multiChannel and a multiChannelStub associated respectively with a client and a server fragment. The class multiChannel (resp. multiChannelStub) inherits from channel (resp. channelStub).

A multicastChannel supports both 1-to-N ("parallel") and 1-to-1-out-of-N ("functional") invocation. It also allows to address any subset of the server group. It relies on a reliable datagram multicast protocol (i.e., messages eventually reach their destinations, but no ordering is guaranteed).

Other available multi-point channels are fifoMulticastChannel and atomicMulticastChannel channels. They have the same client interface as the multicastChannel type. The fifoMulticastChannel guarantees FIFO delivery. The atomicMulticastChannel guarantees that invocations are delivered to all destinations in the same order.

3.2 Sharing FO Types

A *sharing* FO type implements a specific sharing abstraction (e.g., memory sharing, replication, migration, partition, stream). It provides to its clients (fragments of a higher-level FO using it as a connective FO) a memory-like interface: some of them (called producers) write information that others (called consumers) will later explicitly read.

Sharing FO types are structured in several hierarchies. For each data structure (e.g., simple types[1] and records, un-typed raw memory access, lists, trees and graphs), there is an hierarchy of sharing FO types.

For each data structure, the root of its associated hierarchy of sharing FO types defines the generic interface that all other sharing FO types will inherit. Other sharing FO types of this hierarchy implement different policies of sharing this data structure. Furthermore, different sharing FO types implementing a same sharing abstraction (e.g., replication, partition) form an hierarchy too (i.e. a sub hierarchy of the global sharing hierarchy).

For instance, the interface of the root sharing FO type for simple data types is read (DataObject)/write (DataObject), where DataObject is composed of an operation code, un-interpreted data, and the size of data.

[1] Simple types are char, int, ... There is only one hierarchy for all these types.

The operation code is produced by a compiler, during the typed-checking phase of higher-level FOs, and is dynamically checked by the generated marshalling/unmarshalling procedures.

Programmer-defined shared FOs can be easily implemented by using a sharing FO as a connective FO. For instance, a distributed directory can use a specific sharing FO type of type "list", implementing a specific sharing mechanism (e.g., list partitioning).

In the following sections, we present examples of sharing FO types. Section 3.2.1 presents the most primitive FO type. It defines access to physical shared memory. Sections 3.2.2, 3.2.3, 3.2.4 and 3.2.5 present generic sharing FO types for replication, partition, migration and stream. Finally, Section 3.4 presents a higher-level sharing FO type which hides the exact sharing mechanism to the programmer. It can then switch from one mechanism to another according to statistics, load-balancing or fault-tolerance considerations.

3.2.1 Physical Memory Sharing

Objects on a same machine can communicate, through a physical shared memory. Access to a physical shared memory is implemented by a FO type, Shared-Memory FO, which hides the internal representation of the memory (e.g. linear memory space, list of buffers, or array of structures). It ensures type-checked access and performs concurrency control.

3.2.2 Replication

A replicated object is characterized by the consistency which is enforced between its replicas. Well known ones are causal consistency, strong consistency, weak consistency and release consistency. All these consistency semantics suit well for certain classes of application. A replicated object is also characterized by the protocol used to update (or synchronize) replicas (e.g., update, invalidate, replicate operations). Each protocol is appropriate for certain classes of data structure. For instance, for a large data structure, it is more efficient to replicate operations than to use the update protocol.

The designer of a replicated data structure will want a sharing FO type implementing the replication mechanism. Furthermore, he will want this sharing FO to guarantee the consistency of replicas of his data, and to enforce an efficient synchronization protocol.

A replication FO type offer a generic interface of accessing a replicated data structure. Each replication FO implements a specific consistency and/or replication protocol. As we already mentioned earlier, there is one hierarchy of replication FO types per data structure.

A replication FO type uses two connective FO types: a channel for sending data between replica and a synchronization FO type for synchronizing access to replicas.

3.2.3 Partition

Partition FO types provide generic interface for fragmenting a data structure in several partitions. Such a fragmentation allows to support parallel non-conflicting accesses to the fragmented tree.

A partition FO allows to access transparently to all partitions of the data structure that its encapsulates.

Each FO type implements a specific fragmentation protocol. Consider for instance, a tree, fragmented into multiple partitions, localized on several sites. A possible fragmentation can be similar to the one of V-system[1], in which each part is a vertical slice, starting at the root of the tree. Each partition is made of nodes and leaves associated with un-typed data. The set of partitions constitute the partition FO tree. The main role of such a partition FO type is to manage the distribution of the tree and to ensure its consistency.

3.2.4 Migration

A Migration FO type implements a migration policy. This is a elementary object that moves on the site where it is accessed. Several Migration FO types ensure different synchronization policies.

3.2.5 Stream

A Stream FO carries unidirectional information between producers and consumers. It provides two different interfaces: one for producer clients and one for consumer clients. The ordering (FIFO or LIFO) and the buffering policy differs between Stream types, which however all export the same interface. A Stream uses a Channel as a connective FO.

3.3 Synchronization and Concurrency Control

Distributed synchronization and concurrency control can be structured as FOs.

For instance, a Token FO allows its clients to acquire/release a token. At one time, a single fragment holds the token. Each fragment of the Token knows

the token owner. The acquire operation sends a request to the owner and blocks. The release operation (by the owner) broadcasts the list of requesters to all the fragments. Each fragment determines the next owner using a deterministic algorithm. The Token FO has a connective FO of type channel.

Other Synchronization types are semaphore, rendez-vous, barriers, monitors. Each of them will have a connective FO of type channel.

3.4 Examples of Application FO Types Exploiting BOAR FO Types

Application FO types are higher-level distributed shared objects, designed for specific classes of application.

One can remark that an application FO type can also be re-used by several applications. For instance, a FO type implementing a replicated file abstraction can be used by many distributed file systems.

3.4.1 SOS Partitioned Naming Tree

The first example of application FO type is the SosTree, implementing a distributed tree structure, used by the SOS Name Service [4].

The SOS Name Service is implemented by server objects instantiated at boot time in separate "naming contexts", cooperating to process the clients' requests and to ensure the consistency of the distributed Name Space. This cooperation is through a connective object of type SosTree.

SosTree manages a distributed Name Space and uses itself a Tree Partition FO as a connective object. The former FO manages symbolic names and references whilst the latter manages names and DataObjects (section 3.2).

The fragmentation policy of SosTree is similar to the one of V-system[1]. This policy is implemented by the Partition FO type. This choice is specified at binding time; in this case, at run-time for flexibility purpose. Actually, the fragmentation policy to solve conflicts, when adding a name to a partionned node, is simple. In the future, we investigate to define another policy based on load balancing. That will not affect the SosTree.

3.4.2 Replicated Mailbox

A distributed mail application, which needs to replicate mailbox objects for availability purpose, can benefit of the BOAR replicated FO types. In this case, a weak consistency is enough. The designer has just to implement local management of data; he relies on a replication FO type enforcing a weak replication consistency.

4 Conclusion

We presented the Fragmented Object Model and one of its tools, the BOAR library.

Fragmented Objects extend the object programming methodology in a distributed environment. One benefit of this model is that it enforces a clear separation between distribution mechanisms and policies.

BOAR provides FO programmers with predefined FO types implementing commonly used distributed mechanisms. These FO types are organized in several hierarchies, one per distributed abstraction. Connective FO types implementing a same abstraction have the same public interface. This permits a FO programmer to interchange connective FO types implementing a same distributed abstraction. Ideally, a FO programmer can address critical distribution issues such as load-balancing, protection and fault-tolerance just by picking-up from BOAR the connective FO types implementing the mechanisms that he needs. BOAR is highly extensible: new connective FO types can be designed and added into BOAR for future re-use. Finally, BOAR permit a layering structuring of distributed abstractions (e.g., replication uses the channel and synchronization abstractions; hence these abstractions are hidden to application programmer using a replication abstraction).

References

[1] David R. Cheriton and Timothy P. Mann. Decentralizing a global naming service for improved performance and fault tolerance. *ACM Transactions on Computer Systems*, 7(2):147–183, May 1989.

[2] Keith E. Gorlen. Object-oriented program support. OOPS Version 1, Reference Manual, May 1986.

[3] Douglas Lea. Libg++, the GNU C++ library. In *Proc. C++ Conference*, pages 243–255, Berkeley, CA (USA), October 1988. USENIX.

[4] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Structuring distributed applications as fragmented objects. Rapport de recherche 1404, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), January 1991.

[5] Alfred Z. Spector. Performing remote operations efficiently on a local computer network. *ACM*, 25(4):246–260, April 1982.