An Object-Oriented Approach for Replication Management

Yvon Gourhant*
Distributed System Group, Trinity College, Dublin 2, Ireland
e-mail: ygourhant@dsq.cs.tcd.ie

Abstract

One important bottleneck in research on replication management is the time needed for implementing algorithms in order to validate them. Our approach, based on the fragmented object model, encourages re-usability of distributed abstractions for many replication management algorithms. In this paper, we present high-level building blocks for various replication protocols, ensuring different policies of consistency, replicated data management and failure handling.

1 Introduction

One of the main goals of the object-oriented approach is software reuse. Research on replication management brings forward many algorithms that often need reinventing the wheel to progress. Our approach consists of providing high-level building blocks for various replication protocols, each one paying only for the mechanisms it uses. Our goal is to encourage re-usability of distributed abstractions in replication management.

In this paper, we present a library of replicated objects, based on the fragmented object model [11].

Fragmented objects extend the object programming paradigm to a distributed environment. A fragmented object can be viewed from two perspectives. Abstractly (for its clients), it is a single distributed shared object, providing to each client a strongly-typed interface and consequently distribution transparency. Concretely (for the designer), it encapsulates a group of cooperating fragments (i.e., objects with a centralized representation).

These fragments cooperate by invoking the abstract interface of lower-level fragmented objects, called connective objects. For instance, replicated objects use connective objects encapsulating communication protocols (e.g., RPC, diffusion) and synchronization abstractions (e.g., locks, semaphores, rendez-vous, token passing).

These primitive connective objects make up the core of the BOAR library [12], for the moment. The replicated objects, presented in this paper, will extend the library.

Replicated objects can themselves be used as connective objects by higher-level fragmented objects, and recursively until application specific objects. One main idea is that each protocol layer uses only the necessary mechanisms of the lower-level layer and only pays for the cost of them.

Each replicated object implements a particular protocol ensuring specific policies of consistency, replicated data management and failure handling. A number of benefits ensues from providing these building blocks. Firstly, a high-level of reuse between different replicated object types simplifies their implementation and encourages to implement new ones. Secondly, the designers of higher-level fragmented objects can address critical replication issues such as fault-tolerance and availability, just by picking-up from the library the types implementing the protocols that best suit theirs needs.

2 Related work

Several projects provide tools for replicated data management, but the abstractions offered to the programmer are too low level.

For instance, ISIS [3] provides group management and failure detection support that satisfies different needs of different applications, but it doesn’t provide reusable components for replicated data consistency or for recovery scheme.

*The preliminary of this work was done at INRIA. This work is currently implemented by the SOR project at INRIA, 78153 Rocquencourt, France.

1 The cost of the mechanism for handling connective objects is light: two procedure calls plus parameter marshalling/unmarshalling. This is automatically handled by the FOG compiler [5].
On the other hand, z-kernel [1] provides a library of individual protocols encapsulating a single function and easily composed each other. They can be used for various levels of replicated services, each level using only the mechanisms it needs.

Our approach is similar to z-kernel, except that our abstractions are themselves distributed. Distributed components can be re-used for several replicated objects and also interchanged for quickly implementing various replication algorithms.

3 Structure of replicated objects

Internally, a replicated object is composed of replicas, replicating channels, and possibly a logging channel and a storage object (as shown on figure 1). For each component, there are several class hierarchies, with a high degree of reuse. We present these components, sequentially, in this section.

3.1 Replicas

Each replica is itself composed of two objects, which the classes inherit from a common class defining a common interface. This common class may be provided by a centralized library for traditional data structures such as lists, trees, collections.

The first object, called data object, contains a local copy of the data. The second, called interface object, provides the interface of the replicated object to clients on this node. A client can not distinguish between the interface object and the whole replicated object.

The object interface triggers client invocations to the data object or to the replicating channel and possibly to the logging channel. It handles full or partial replication depending on the granularity and structure of the data. So the replicating channel may be used either for sending data (e.g., for updating) or for replicating operations (i.e. processing the same operation on each replicas). This latter is essential for replicating efficiently some data structures. Consider, for instance, adding a new host in a replicated host table structured as a linked-list. Moreover, such a data structure benefits of a semantic-based synchronization [15]. The synchronization objects presented in the next section extend this model to a distributed environment.

3.2 Replicating channels

Replicated channel types offer a generic interface for replicating data and operations, and for enforcing consistency between replicas. Well known consistency semantics are strong consistency, causal consistency [10], weak consistency and release consistency [4]. All these consistency semantics are well suited to certain classes of application. Each replicated channel implements also a particular protocol used to synchronize replicas (e.g., update, invalidate). The class of data structure in use influences the form of protocol that is appropriate. For instance, for a small data structure, it is more efficient to update than to use an invalidation protocol.

Internally, a replicating channel uses two connective objects, to multicast data and to maintain consistency between replicas.

First, a multicast channel provides multicast communications. Several multicast channels offer the same interface but provide different qualities of service such as reliability and ordering (fifo, causal, atomic, global [3]) at different costs.

Second, most used synchronization objects implement distributed locking and token passing. We provide both implicit and explicit synchronization. The implicit case is attractive because the synchronization objects are encapsulated by replicated objects, and their methods are automatically invoked when the enclosing replicated object is invoked. The explicit case is potentially more efficient. For instance, it allows processing of several invocations locally and to update other replicas once at release time [4]. Implicit synchronization is best suited to application-level objects while explicit synchronization is more suitable to connective objects.

This approach provides a high degree of flexibility. Different protocols may implement different buffering policies. The class of a synchronization protocol is reused by the classes of protocols ensuring particular consistency semantics, as well. A replicated object just uses the abstract interface, common to different interchangeable protocols. The choice of a particular protocol is made at creation time. It’s easy to choose and replace replication protocols and consistency policies.

The multicast channels and the synchronization objects of the same enclosing replicated object use themselves a common lower-level connective object for sending and receiving messages. Thus, messages are multiplexed on the same transport protocol, which is tightly coupled with the operating system.
3.3 Logging channels

Some replicated objects use a logging channel for registering updates, for synchronization or fault-tolerance purposes ("redo" past actions, correct possible faults, errors or loss of consistency). A logging channel is a connective object, storing data at the end of a physical medium and able to recover data for registering updates, for synchronization or fault-tolerance purposes. Each application to define its own (arbitrary) state.

Logging is the basic mechanism for both optimistic and pessimistic concurrency control mechanisms. Our approach does not enforce a specific policy but allows each application to define its own. In particular, a logging channel can be used for replaying operations, with an optimistic concurrency control algorithm [9].

A logging channel is characterized by its buffering policy, by multiplexing/de-multiplexing mechanisms, and by the management of physical media. Each characteristic is embodied in a class hierarchy. A logging channel is built using object composition, in the same way than in Lipto [5]. Moreover, a programmer can customize its own logging channel by selecting and stacking appropriate objects offering the same interface, as same as Ficus [13].

A logging channel is itself a replicated object at a lower-level. The number of replicas and the storage media (primary or secondary storage) can be parametrized at creation time. The number of replicas can be totally unrelated to the number of replicas of the enclosing object, depending on failure assumptions. New replicas may of course be created in case of failure.

3.4 Reliable storage objects

Just as logging channels register updates, storage objects offer a generic interface for check-pointing data. Internally, a reliable storage object uses a replicated object for storing objects on several nodes. Different storage policies may be chosen (e.g., full or partial replication).

On the other hand, the network partitioning problem may be solved by an approach based on object semantics: e.g., knowledge of the number of the partners and possibly of partners themselves (a special tree may provide particular semantics in case of partitioning [8]).

Moreover, the layering of fragmented objects allows global decisions being taken in case of failure. For instance, relations between different replicated objects contained in an enclosing object could be defined for providing consistency without exchanging messages in case of network partitions [2].

Finally, one can notice that many protocols implementing reliable storage are variations of common protocols such as: write all/read 1 protocol or Gifford's quorum protocol [6]. It is of primary interest to reuse code between different implementations (for example, quorum management, weight allocation, mutual decision).

4 Conclusion

We have presented a library of replicated objects structured as fragmented objects, in order to facilitate quick prototyping of algorithms. One benefit of the fragmented object model is that it enforces a clear separation between mechanisms and policies.

Currently, BOAR contains mainly primitive fragmented objects, such as communication channels, synchronization objects and logging channels. Different replicated objects and reliable storage objects are being implemented. We strongly believe that this library is of primary interest for encouraging programmers to reuse abstractions between distributed applications. It will progressively accumulate objects needed by a large number of applications.

Our experience shows that this approach increases the problem of choosing the classes implementing the particular characteristics associated with each problem. But research on this topic is also progressing.

Acknowledgements

The work presented here has been developed by the SOIR project at INRIA. The author is grateful to Messa Makpangou and Michel Ruffin for their participation in the design. I would like also to thank Mark Sheppard and Hervé Soulard for their assistance with this paper.

References


Figure 1: Macroscopic structuring