An Object-Oriented Approach for Replication Management*

Yvon Gourhant†
Distributed System Group, Trinity College, Dublin 2, Ireland
e-mail: ygourhnt@dsg.cs.tcd.ie

April 22, 1992

1 Introduction

One of the main goals of the object-oriented approach is software reuse. Research in 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 reusability of distributed abstractions in replication management.

In this paper, we present BOAR, a library of replicated objects, based on the fragmented object model [10]. 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). Replicated objects are themselves used as connective objects by higher-level fragmented objects, and recursively until application specific objects. Each protocol layer uses only the necessary mechanisms of the lower-level layer and only pays for the cost of them[1].

Each replicated object in BOAR 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 Structure of replicated objects

Internally, a replicated object is composed of *replicas*, *replicating channel*s, 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, one per one, in this section.

2.1 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

*Submitted to WMRD-II (Monterey, CA, November 12-13, 1992)

†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 [6].

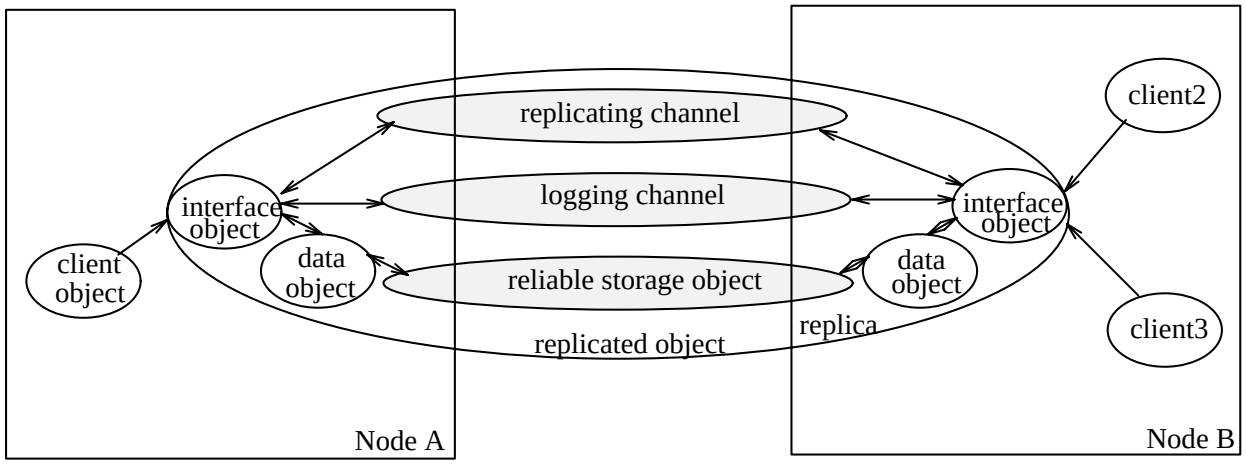

Figure 1: Macroscopic structuring

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 replica). 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[2]. Moreover, such a data structure benefits of a semantic-based synchronization [12]. The synchronization objects presented in the next section extend this model to a distributed environment.

2.2 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 [9], weak consistency and release consistency [3]. 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[2]) 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 theirs methods are automatically invoked when the enclosing replicated object is invoked. The explicit case is potentially more efficient. For instance, it allows to process several invocations locally and to update other replicas once at release time [3]. 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

[2] A list is also a typical example of a class picking-up from a centralized library.

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. This allows to choose and replace easily 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.

2.3 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 from an arbitrary state[3] [11].

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 [8].

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 [4]. A programmer can customize its own logging channel by selecting and stacking appropriate objects.

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.

2.4 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[4]).

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 [7]).

Moreover, the layering of fragmented objects allows global decisions to be 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 [1].

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 [5]. It is of primary interest to reuse code between different implementations (for example, quorum management, weight allocation, mutual decision).

3 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

[3] Object interfaces are omitted from the extended abstract due to space constraints.

[4] The data of a replicated object may be fragmented in several distinct partitions. Each partition may be stored independently on one or several media, for availability and space-economy purposes.

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 SOR project at INRIA. The author is grateful to Mesaac 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

[1] Daniel Barbará and Hector Garcia-Molina. The case for controlled inconsistency in replicated data. In *IEEE Computer Society Technical Committee on Operating Systems and Application Environments Newsletter*, volume 4, pages 8–11. IEEE Computer Society, 1990.

[2] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 123–138, Austin TX (USA), November 1987. ACM.

[3] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, volume 25 of *Operating Systems Review*, pages 152–164, Pacific - Grove CA (USA), October 1991.

[4] Peter Druschel, Larry L. Peterson, and Norman C. Hutchinson. Lipto: A dynamically configurable object-oriented kernel. In *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems and Application Environments*, volume 5, pages 11–16. IEEE, 1991.

[5] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pages 150–162, December 1979.

[6] Yvon Gourhant and Marc Shapiro. FOG/C++: a fragmented-object generator. In *C++ Conference*, pages 63–74, San Francisco, CA (USA), April 1990. Usenix.

[7] Andy Hisgen, Andrew Birrell, Jerian Chuck, Timothy Mann, Michael Schroeder, and Garret Swart. Granularity and semantic level of replication in the Echo distributed system. In *IEEE Computer Society Technical Committee on Operating Systems and Application Environments Newsletter*, volume 4, pages 30–32. IEEE, IEEE Computer Society, 1990.

[8] D. B. Johnson and W. Zwaenepoel. Output-driven distributed optimistic message logging and checkpointing. Technical Report TR90-118, Dept. of Comp. Sc., Rice University, Houston, Texas (USA), May 1990.

[9] Rivka Ladin, Barbara Liskov, and Liuba Shrira. Lazy replication: Exploiting the semantics of distributed services. In *IEEE Computer Society Technical Committee on Operating Systems and Application Environments*, volume 4, pages 4–7. IEEE, IEEE Computer Society, 1990.

[10] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Fragmented objects for distributed abstractions. In T. L. Casavant and M. Singhal, editors, *Advances in Distributed Computing: Concepts and Design*. IEEE Computer Society Press, 1992. To appear.

[11] Michel Ruffin. Kitlog: a generic logging service. In *11th Symp. on Reliable Dist. Systems.*, Houston (TX, USA), October 1992. to appear.

[12] P Schwartz and A. Spector. Synchronizing shared abstract types. In *ACM Transactions on Computer Systems*, volume 2, pages 223–250, August 1984.