Bridging the Gap between Application Semantics and Group Communication Protocols

Fernando Pedone* André Schiper† Aad van Moorsel*

*Software Technology Laboratory
Hewlett-Packard Laboratories
Palo Alto, CA 94304-1126, USA
{pedone, aad}@hpl.hp.com

†Communication Systems Department
Swiss Federal Institute of Technology
CH-1015 Lausanne, Switzerland
Andre.Schiper@epfl.ch

Abstract

Group communication protocols have received great attention over the past years. From a practical viewpoint, several middleware systems have implemented the group communication abstraction; from a theoretical viewpoint, a sound theory underlying group communication has been developed and minimal solvability conditions have been identified. So far, however, group communication protocols have not taken application semantics into account, and, in some cases, the guarantees offered by group communication protocols are stronger than necessary for application correctness. In this short paper, we claim that exploiting application semantics is a promising direction for future research on group communication protocols, and discuss Generic Broadcast, an example of a group communication protocol that allows applications to express their message ordering needs.

Keywords: application semantics, generic broadcast, group communication protocols, atomic broadcast

1 Introduction

Group communication protocols were introduced in the mid-80's as a way to handle groups of processes as individual entities. Group communication protocols have received great attention over the past years from both practical and theoretical perspectives. A well-known group communication system is Isis [BSS91], and several other group communication-like systems have been built since. Furthermore, trends in middleware systems seem to confirm the important role played by group communication protocols [Gro98]. From a theoretical point of view, a sound theory underlying group communication has been developed, and minimal conditions under which Atomic Broadcast is solvable in the context of crash failures have been identified [CT96, CHT96].

According to the guarantees provided, group communication protocols can come in several different flavors, such as causal, atomic, and total order message delivery [BSS91]. For example, Atomic Broadcast enables to send messages to a set of processes with the guarantees that the destinations agree on the messages delivered, a property known as *agreement*, and on the order according to which the messages are delivered, a property known as *total order*.

Several works have advocated the use of group communication protocols to propagate requests in replicated systems (e.g., it has been shown that Atomic Broadcast is sufficient to solve Active Replication [Sch90]). However, so far, group communication protocols have treated messages in a purely syntactic basis, that is, without considering the semantic meaning of messages in the application context, and from the application's viewpoint, the guarantees offered by group communication protocols are, in some cases, stronger than necessary for correctness. As a consequence, the application ends up paying the cost of ensuring guarantees it does not need.

In this short paper, we claim that exploiting application semantics is a promising direction for future research on group communication protocols. We support our claim with Generic Broadcast, an example of a group communication protocol that allows applications to express their message ordering needs.

2 Semantics-Aware Group Communication Protocols

To illustrate the use of semantic information in group communication protocols, we present a simple replicated database system where replication is controlled by a direct application of the state machine approach [Sch90]. The database system is composed of individual database sites, $\Sigma = \{DB_1, DB_2, ..., DB_n\}$. Each site DB_i stores a copy of all data items. Clients submit requests to the system using an Atomic Broadcast primitive, and wait for the first response to come back from the database sites. Requests are single SQL statements, and can be queries (e.g., `SELECT`-like statements) or updates (e.g., `UPDATE`-like statements).

The agreement and total order properties of the Atomic Broadcast primitive and some deterministic assumptions about the behavior of the database sites guarantee that clients perceive the replicated database sites as a single highly-available database site (i.e., the replicated system is one-copy-serializable). However, since `SELECT` requests commute, ordering such requests is not necessary for the correctness of the system.[1] This is a simple observation; however, until recently, no group communication protocol fully exploited it, and any attempts toward this direction relied at best on *ad hoc* mechanisms.

2.1 Generic Broadcast

Generic Broadcast is a group communication specification that takes message semantics into account to enforce the order necessary (and not only sufficient) for the correctness of the applications. Generic Broadcast was introduced in [PS99] as a generalization of Atomic Broadcast. It uses the notion of *message conflict* to re-define the order property of Atomic Broadcast. Message conflict is based on a *conflict relation* $\mathcal{C}$ which depends on the semantic of the messages, defined by the application. Taking $\mathcal{C}$ into account, the total order property of Atomic Broadcast is replaced by the following order property.

- If processes p_i and p_j both deliver messages m and m' and (m, m') is in $\mathcal{C}$, then p_i delivers m before m' if and only if p_j delivers m before m'.

Thus, Atomic Broadcast is a special case of Generic Broadcast where all messages conflict with each other. Back to the example presented before, commutable `SELECT` requests can now be exploited by Generic Broadcast with a conflict relation expressed as

$$\mathcal{C} = \{(m, m') : m = \text{"}\texttt{UPDATE ...}\text{"} \textbf{ or } m' = \text{"}\texttt{UPDATE ...}\text{"}\},$$

[1] Requests also commute if they do not access the same records, but we do not elaborate on this point.

meaning that any two messages m and m' conflict (i.e., have to be ordered with respect to one another) if and only if at least one of them contains an `UPDATE` request.

Generic Broadcast can also be used in more general cases. For example, propagating lock requests with an Atomic Broadcast primitive can avoid distributed deadlocks in a replicated database, a serious problem that may happen if requests are sent using single message passing [GHOS96]. However, Atomic Broadcast orders all requests, even though only a portion of the database is responsible for the problem (e.g., the hot spots). Generic Broadcast is an elegant and efficient way of propagating lock requests: it is cheaper than Atomic Broadcast, and does not lead to distributed deadlocks.

3 Solving Generic Broadcast

In the following, we present the ideas behind two algorithms that solve Generic Broadcast, $\mathcal{GB}$ and $\mathcal{GB}+$. Both algorithms can deliver non-conflicting messages without using Consensus, and rely on Consensus [CT96] when conflicts are detected. $\mathcal{GB}+$ improves the performance of $\mathcal{GB}$ by allowing, in some cases, conflicting messages to be delivered without Consensus.

3.1 The $\mathcal{GB}$ Algorithm

The $\mathcal{GB}$ algorithm assumes that less than a third of processes can crash. In cases where Consensus is not necessary, processes executing the $\mathcal{GB}$ algorithm can deliver messages after two communication steps, and in cases where Consensus is necessary, after four communication steps (best case). This is a tradeoff with respect to known Atomic Broadcast algorithms that order messages in three communication steps (best case): if few messages conflict, $\mathcal{GB}$ performs better than known Atomic Broadcast algorithms, if many messages conflict, it performs worse.

Executions of $\mathcal{GB}$ are decomposed in a sequence of *stages*, divided into two *phases*: the first phase lasts as long as no conflicting messages are received, and the second phase handles the delivery of conflicting messages with help of Consensus. When a process p_i starts some stage k, p_i is initially in phase I. Phase I terminates at process p_i iff p_i receives two messages that have to be ordered. In phase II of stage k, process p_i executes a Consensus algorithm to decide on the delivery order of these messages. When Consensus terminates, p_i proceeds to phase I of stage $k+1$.

Processes determine whether two messages have to be ordered in phase I of stage k by evaluating a local $order(m, m')$ predicate. The $order(m, m')$ predicate holds at time t on p_i iff (1) p_i in stage k has received messages m and m', but (2) p_i has not delivered any of these messages in some previous stage $k' < k$, and (3) (m, m') is in the conflict relation $\mathcal{C}$. Therefore, whenever $order(m, m')$ holds, p_i relies on Consensus to order messages m and m' in stage k. The $\mathcal{GB}$ algorithm ensures that if p_i does not crash and starts a Consensus execution at stage k, then every process p_j that does not crash also starts Consensus at stage k. Finally, before a process p_j executes Consensus, p_j has to gather some information from the other processes about which messages might have been delivered in phase I. This reason is that if m and m' conflict, and some process has delivered m in phase I, no process should deliver m' before m.

3.2 The $\mathcal{GB}+$ Algorithm

The $\mathcal{GB}+$ algorithm is similar to $\mathcal{GB}$ but based on a weaker order predicate than the one used by $\mathcal{GB}$, in the sense that in some cases, even conflicting messages can be delivered without Consensus. The order predicate for $\mathcal{GB}+$ holds at time t iff (1) p_i in stage k has received messages m and m', but (2) p_i has not delivered any of these messages at time t, and (3) (m, m') is in the conflict relation $\mathcal{C}$.

To understand the difference between the two order predicates, consider an execution in which only two conflicting messages m and m' are broadcast, and assume that m is delivered by some process p_i in phase I of stage k. Assume that later, but still in phase I of stage k, process p_i receives m'. In this case, with $\mathcal{GB}$, process p_i evaluates the order predicate to true, and starts phase II to terminate the current stage. However, this is not necessary as the Consensus decision is known beforehand: m must be delivered before m'! So, while p_i executing $\mathcal{GB}$ proceeds to phase II, with $\mathcal{GB}+$, process p_i remains in phase I and may delivers m' in phase I, even though (m, m') is in $\mathcal{C}$. As delivering a message in phase I is cheaper than delivering it in phase II, $\mathcal{GB}+$ improves $\mathcal{GB}$.

4 Final Remarks

There is little doubt that using application semantics leads to powerful group communication primitives. However, exploiting this dimension faces two challenges. First, it is not clear how primitives should be specified to take application semantics into consideration (e.g., the conflict relation of Generic Broadcast allows applications to define their message ordering needs). Second, such specifications can be very hard to implement efficiently. Moreover, addressing the first issue certainly requires keeping an eye on the second. Generic Broadcast has shown that this can be done, and hopefully, it will pave the way to other semantics-aware group communication protocols.

References

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[CHT96] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[GHOS96] J. N. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Montreal (Canada), June 1996.

[Gro98] Object Management Group. Fault tolerant CORBA using entity redundancy. Request for proposal, Object Management Group, Framingham Corporate Center, Framingham (USA), April 1998.

[PS99] F. Pedone and A. Schiper. Generic broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99, formerly WDAG)*, September 1999.

[Sch90] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.