André Schiper and Michel Raynal

# *From Group Communication to Transactions in* **Distributed Systems**

Because toolkits for developing process groups do not allow applications to issue reliable multicasts to multiple groups, a new development model distinguishing between groups as logical addressing mechanisms and reliable communication primitives is needed to create reliable distributed applications.

The design of structuring concepts that facilitate development of reliable and complex applications and implementation of associated mechanisms is today one of the most important research tasks in computer science. In this context, the 1970s saw the emergence of *transactional* computing [1], while the 1980s saw the emergence of *group communication*. This short article describes a group-based system, showing that transaction-based systems and group-based systems are not antithetical. More precisely, we show that adequate group communication can support a specific class of transactions in asynchronous distributed systems.

A transaction is a sequence of operations on objects (or on data items) that satisfies the following three properties:

- *Atomicity*, also called the *all-or-nothing* property, requiring that either all the operations of the transaction (on all the objects it accesses) are performed, or none of them is performed;
- *Permanence*, requiring that, despite crashes, the effect of the performed operations is permanent;
- *Ordering*, usually called *serializability*, requiring that transactions appear to have been executed in some sequential order.

While all-or-nothing and permanence properties address fault-tolerance and are on each transaction separately, the ordering property addresses concurrency and is expressed in terms of the set of committed transactions.

Groups were first introduced in the V-Kernel [4] as a syntactical convenience to express one-to-many communication structures and have subsequently been seen as a convenient addressing mechanism. Groups for addressing are widely used by the parallel systems community (see, for example, the Parallel Virtual Machine [PVM] package [6]). However, V-Kernel and PVM addressing groups are adequate for disseminating information only in fault-free environments, as they provide no delivery guarantees in case of failures. Subsequently, the group paradigm has been extended to include strong guarantee in the presence of failures, an approach pioneered by the Isis system [2]. These extensions have promoted the use of groups as an abstraction to represent fault-tolerant services and have combined the group abstraction with various group multicast primitives.

## Groups for Fault Tolerance

Fault tolerance in asynchronous distributed systems can be achieved through replication. Implementation of fault-tolerant services relies on replication of some deterministic process, a technique often called the "state-machine approach'' [11]. Replication of a state machine can be hidden to its clients through the *fault-tolerant group* abstraction. Such an abstraction presents two principal characteristics:

- A fault-tolerant group $g$ names a set of processes that share a common state; every member has a copy of the common state. Because of process crashes and recoveries, not all members of $g$ are necessarily operational at the same time. A member of $g$ recovering after a crash receives an up-to-date copy of the state of the group $g$ from any operational member of $g$, an event called state transfer.
- Update of the state of the group requires adequate multicast primitives to ensure that the state shared by the members of $g$ is always consistent. For sake of clarity, we decompose update properties of group multicast primitives into three subproperties:
  - A *delivery permanence* property defined on a group taken individually;
  - An *all-or-none delivery* property defined on multiple groups; and
  - An *ordered delivery* property defined on a set of multicasts.

The degree of replication of a group $g$ is based on a worst-case analysis. Assume, for example, that processes fail only by crashing, and that at any time, at most $f$ processes are simultaneously crashed. With this assumption, a group of $f + 1$ members ensures permanence of the group state—any time there is at least one operational member in $g$. Actually, given that at most $f$ processes can be simultaneously crashed, a group usually consists of $2f + 1$ members. This degree of replication ensures that at any time the group contains a majority of operational members—a condition required by the implementation of the group multicast primitives defined in the following sections. From here on, we consistently assume groups have $2f + 1$ members and a maximum of $f$ simultaneous crashed processes.

**Single group multicast with delivery permanence property.** Consider the multicast of message $m$ to the group $g$. It follows from the earlier discussion that delivery of $m$ is permanent in $g$ only after a majority of its members has delivered $m$. Thus, a member of $g$ should not deliver a message $m$ before a majority of its members has agreed to deliver it. This defines the *delivery permanence* property. More precisely, the semantics associated with this property are the following: Either a majority of $g$ agrees to deliver $m$—and consequently every operational member of $g$ eventually delivers $m$—or no member of $g$ delivers m.[1] We use SEND($m$ to $\{g\}$) to denote the multicast of $m$ to $g$ with delivery permanence semantics.

**Multiple groups multicast with all-or-none delivery property.** While group-based systems have generally considered multicasts to a single group, nothing prevents us from extending the SEND primitive to multiple groups. This step is important in making explicit the link between group communication and transactions. A transaction aggregates operations on multiple objects. So, if every object is replicated and managed by a fault-tolerant group, a transaction operates on multiple groups. Consequently, if a message $m$ aggregates the operations of a transaction, message $m$ has to be multicast to all the groups concerned by the transaction described by $m$.

Given two groups $g$ and $g'$, we define SEND ($m$ to $\{g,g'\}$) by the following property: Either a majority of $g$ and a majority of $g'$ agree to deliver $m$—and consequently all operational members of $g$ and $g'$ eventually deliver $m$—or none of the members of $g$ or of $g'$

[1] This property—also called "majority agreement'' (see F. Cristian's article in this special section)—is close to uniform reliable multicast [8]. However, uniform reliable multicast is defined in a static system model in which processes do not recover after a crash. Group-based systems consider a dynamic system model in which processes can join the computation and recover after a crash.

**Transactions based on group communication primitives** *represent an important step toward extending the power and generality of group communication as a broad distributed computing discipline.*

delivers $m$. This *all-or-none delivery* property links groups $g$ and $g'$. Either the state of $g$ and the state of $g'$ are permanently updated by $m$, or neither of the states is updated (e.g., see [10]).

**Multicast with global total order property.** The `SEND` primitive can be extended by an ordering property. We denote `TO-SEND` the `SEND` primitive with an additional global *totally ordered delivery* property. Consider two messages $m_1$ and $m_2$, `TO-SEND`($m_1$ `to` {...}) and `TO-SEND`($m_2$ `to` {...}), and let $p_i$ and $p_j$ be two processes that deliver both $m_1$ and $m_2$. Then $p_i$ and $p_j$ deliver $m_1$ and $m_2$ in the same order.[2]

Most existing group-based systems provide only a total order multicast primitive to one single group (e.g., `TO-SEND` ($m$ `to` {$g$}). Total order multicasts to multiple groups (e.g., `TO-SEND`($m$ `to` {$g$,$g'$}) permits extension of such systems to address the group communication requirements of transactional applications.

**From group communication to transactions.** Consider a classical transaction that transfers $1,000 from bank account #1 to bank account #2. To achieve fault tolerance, assume that each bank account is replicated on several nodes, and assume that every replica is managed by a process. Let $g_1$ be the fault-tolerant group of processes that manage bank account #1, and let $g_2$ be the fault-tolerant group of processes that manage bank account #2. The two operations (withdrawal and deposit) can be aggregated into a single message by defining $m$ as: (*remove $1,000 from account #1; add $1,000 to account #2*). When a process in $g_1$ delivers $m$, it removes $1,000 from the bank account it manages; when a process in $g_2$ delivers $m$, it adds $1,000 to the bank account it manages. In this distributed setting, the money transfer transaction can be expressed as `TO-SEND`($m$ `to` {$g_1$,$g_2$}):

- The *all-or-none delivery* property of `TO-SEND`($m$ `to` {$g_1$,$g_2$}) corresponds to the transaction *all-or-nothing atomicity* property. Updates (money transfer) described by message $m$ are taken into account by both or neither of the two objects (bank accounts);
- The *delivery permanence property*, ensured on $g_1$ and on $g_2$, corresponds to the transaction permanence property. Processes in $g_1$ and in $g_2$ deliver $m$ only if a majority of $g_1$ and a majority of $g_2$ have agreed to deliver $m$. This property ensures permanence of the update defined by $m$; and
- The *total ordering delivery* property of `TO-SEND` ensures the serializability property of transactions.

Due to space limitations, other properties of transactions, such as unilateral abort, are not considered here. Notice that implementation of the permanence property usually requires some information to be logged into permanent storage (e.g., on disk). No such logging technique is needed here because permanence is ensured by replication.

The multicast primitive `TO-SEND` ($m$ `to` {$g_1$,$g_2$, . . . ,$g_n$}) implements a transaction $t$ on a set of $n$ fault-tolerant objects managed by the groups $g_1$, $g_2$, . . . ,$g_n$, respectively. Operations of $t$ are aggregated into message $m$. When comparing this `TO-SEND` primitive with traditional implementations of transactions, it is worth noting that the `TO-SEND` primitive integrates data locking, data update, and atomic commitment in a single operation. This primitive is more efficient than the traditional multistep implementation of the same transaction and, consequently, makes group communication an attractive implementation alternative for a specific class of transactions in distributed systems.

## Conclusion

Most existing systems restrict multicast primitives to a single group at a time. This restriction simplifies the implementation but also obscures the link between group-based systems and transactional systems. Aggregating operations in one message and providing multicast primitives to multiple groups make the link visible. Multicasting to multiple groups is available in the Totem system [9], where delivery order of two messages is determined from the messages themselves (called *born-order* in Totem). Nonborn-ordered implementations are also possible [7] and are valuable in settings where the group members might trigger unilateral aborts.

[2]Chandra and Toueg showed that the total order broadcast problem is equivalent to the consensus problem [3] and thus subject to the Fischer-Lynch-Paterson impossibility result about solving consensus in an asynchronous system (in which message transmission delays are not bounded) [5]. These problems are solvable, however, in an asynchronous system augmented with (even unreliable) failure detectors [3].

Despite the success of existing group communication primitives for asynchronous systems, there is still room for versatile systems better suited to users' needs. Introduction of transactions based on group communication primitives represents an important step toward extending the power and generality of group communication as a broad distributed computing discipline for designing and implementing reliable applications. ◘

**References**

1. Bernstein, P.A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Distributed Database Systems.* Addison-Wesley, Reading, Mass. 1987.
2. Birman, K. The process group approach to reliable distributed computing. *Commun. ACM 36*, 12 (Dec. 1993), 37–53.
3. Chandra, T.D., and Toueg. S. Unreliable failure detectors for reliable distributed systems. To appear in *Journal of the ACM.* A preliminary version appeared in *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing* (Aug. 1991, Montreal, Canada) 325–340. ACM Press, New York, 1991.
4. Cheriton, D.R., and Zwaenepoel, W. Distributed process groups in the V-kernel. *ACM Transactions on Computer Systems 3*, 2 (Feb. 1985) 77–107.
5. Fischer, M., Lynch, N., and Paterson, M. Impossibility of distributed consensus with one faulty process. *Journal of the ACM 32*, 4 (Apr. 1985) 374–382.
6. Geist, G.A., Beguelin, A., Dongarra, J.J., Jiang, W., Manchek, R., and Sunderam, V.S. *PVM: A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, MA, 1994.
7. Guerraoui, R., and Schiper, A. *Transaction Model vs Virtual Synchrony Model: Bridging the Gap.* In Theory and Practice in Distributed Systems. Springer Verlag, Berlin, Germany, LNCS 938, 121–132, 1995.
8. Hadzilacos, V., and Toueg, S. Fault-tolerant broadcast and related problems. In *Distributed Systems,* S. Mullender Ed., Addison-Wesley and ACM Press, New York, 1993, pp. 97–145.
9. Moser, L.E., Melliar-Smith, P.M., Agarwal, D.A., Budhia, R.K., and Lingley-Papadopoulos, C.A. Totem: A fault-tolerant multicast group communication system. *Commun. ACM 39*, 4 (Apr. 1996).
10. Powell, D. Distributed fault-tolerance: Lessons learned from Delta-4. *IEEE Micro 14*, 4, (Feb. 1994) 36–47.
11. Schneider, F.B. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys 22*, 4, (Apr. 1990) 299–319.

**About the Authors:**

**ANDRÉ SCHIPER** is a professor of computer science at the Federal Institute of Technology (EPFL) in Lausanne, Switzerland, where he heads the Operating Systems Laboratory. **Author's Present Address:** EPFL, Dept. d'Informatique, 1015 Lausanne, Switzerland; email: schiper@di.epfl.ch

**MICHEL RAYNAL** is a professor in the computer science department at the University of Rennes, France. He also heads the IRISA INRIA research group working on distributed algorithms, fault tolerance, and distributed systems. **Author's Present Address:** IRISA, Campus de Beaulieu, 35042 Rennes-cedex, France; email: raynal@irisa.fr