Group Communication: from practice to theory [⋆]

André Schiper

Ecole Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland

Abstract. Improving the dependability of computer systems is a critical and essential task. In this context, the paper surveys techniques that allow to achieve fault tolerance in distributed systems by *replication*. The main replication techniques are first explained. Then *group communication* is introduced as the communication infrastructure that allows the implementation of the different replication techniques. Finally the difficulty of implementing group communication is discussed, and the most important algorithms are presented.

1 Introduction

Computer systems become every day more and more complex. As a consequence the probability of problems in these systems increases over the years. To avoid this from becoming a major issue, researchers have since many years worked on improving the dependability of these systems. The methods involved are traditionally classified as *fault prevention*, *fault tolerance*, *fault removal* and *fault forecasting* [22]. *Fault prevention* refers to methods for preventing the occurrence or the introduction of faults in the system. *Fault tolerance* refers to methods allowing the system to provide a service complying with the specification in spite of faults. *Fault removal* refers to methods for reducing the number and the severity of faults. *Fault forecasting* refers to methods for estimating the presence of faults (with the goal to locate and remove them). We concentrate here on fault tolerance.

Several techniques to achieve fault tolerance have been developed over the years. The different techniques are related to the specificity of applications. For example, a centralized application differs from a distributed application involving several computing systems. We consider here distributed applications. Fault tolerance for distributed applications can be achieved with different techniques: *transactions*, *checkpointing* and *replication*.

*Transaction*s have been introduced many years ago in the context of database systems [3]. A transaction allows us to group a sequence of operations while ensuring some properties on these operations, called *ACID* properties [3]: *Atomicity*, *Consistency*, *Isolation* and *Durability*. *Atomicity* requires that either all

[⋆] The same paper will appear under the title *Dependable Systems* in *Dependable Information and Communication Systems*, to be published in the Springer LNCS series, 2006. Research supported by the Hasler Stiftung under grant number DICS-1825.

operations of the transaction are preformed, or none of them. *Consistency* is a requirement on the set of operations, namely that the sequence of operations brings the database from a consistent state to another consistent state. Transactions can be executed concurrently. The *isolation* property requires that the effect of transactions executed concurrently is the same as if the transactions where executed in some sequential order (in *isolation* from each other). *Durability* requires that the effect of the operations of the transaction are permanent, i.e., survive crashes. Durability is achieved by storing data on stable storage, e.g., on disk. Atomicity and durability are the two properties specifically related to fault tolerance. A single protocol is used to ensure these two properties, the so called *atomic commitment* protocol executed at the end of the transaction. If all the data accessed by a transaction is located on the same machine, the transaction is a *centralized* transaction. If the data is located on different machines, the transaction is a *distributed* transaction. Distributed transactions are more difficult to implement then centralized transactions. The main technical difficulty lies in the atomic commitment protocol. Except for this problem, the implementation of distributed transactions derives more or less easily from the implementation of centralized transactions. We discuss atomic commitment in Section 4.5.

Checkpointing is another technique for achieving fault tolerance. It consists of periodically saving the state of the computation on stable storage; in case of a crash, the computation is restarted from the most recently saved state. The technique has been developed for long running computations, e.g., simulations that last for days or weeks, and run on multiple machines. These computations are modelled as a set of processes communicating by exchanging messages. The main problem is to ensure that, after crash and recovery, the computation is restarted in a consistent state. We do not discuss checkpointing techniques here. A good survey can be found in [12].

Replication is the technique that allows the progress of the computation during failures (which is called failure *masking*). In a system composed of several components, without replication, if one single component fails the system is no more operational. Replicating a component C, and ensuring that the replicas of C fail independently, allows the system to be tolerant to the failure of one or several replicas of C. Replicating a component is very easy if the component is stateless or if its state does not change during the computation. If the state of the component changes during the computation, then maintaining the consistency among the replicas is a difficult problem. Surprisingly, it is one of the most difficult problems in distributed computing. We concentrate here on the problems related to replication.

While replication allows us to mask failures, this is not the case of transactions or checkpointing. However, the different techniques mentioned above can be combined, e.g., transactions can be run on replicated data. Implementing such a technique requires to combine transaction techniques and replication techniques. This will not be discussed here.

The rest of the paper is structured as follows. Section 2 introduces issues related to replication, and presents the two main replication techniques. Section 3 defines group communication as the middleware layer providing the tools for implementing the different replication techniques. The implementation of these tools is discussed in Section 4. Finally, Section 5 concludes this survey.

2 Replication

In this section we first introduce a model for discussing replication. Then we define what it means for replicas to be consistent. Finally we introduce the two main replication techniques.

2.1 Model for replication

Consider a system composed of a set of components. A component can be a *process*, an *object*, or any other system structuring unit. Whatever the component is, we can model the interaction between components in terms of inputs and outputs. A component CO receives inputs and generates outputs. The inputs are received from another component CO_{in}, and the outputs are sent to some component CO_{out}. Whether CO_{in} is equal or not to CO_{out} does not make any difference for CO. In the case $CO_{in} = CO_{out}$, the component CO is called a *server*, and the component $CO_{in} = CO_{out}$ is called a *client*. In this case we will denote the server component by S and the client component by C. The input sent by the client C to the server S is called a *request*, and the output sent by the server S to the client C is called a *response*. From the point of view of the client, the pair *request/response* is sometimes called an *operation*: for a client C, an operation consists of a request sent to a server and the corresponding response. We assume here that the client is blocked while waiting for the response.

2.2 Consistency criteria

A server S can have many clients C, C', C'', etc. For a non-replicated server S, the simplest implementation is to handle client requests sequentially, one at a time. A more efficient implementation could consist for the server to spawn a new thread for each new incoming request. However, in this case the result that the client obtains must be the same as if the operations were executed sequentially, one after the other. The same holds if the server S is replicated, with replicas S_1, …, S_n: the result that the clients obtain must be the same as if the operations were executed sequentially by one single server. This can be defined more precisely, by the consistency criterion called *linearizability* [16] (also called *atomic consistency* [24]). A weaker consistency criterion is called *sequential consistency* [19]. We discuss only linearizability, which is the consistency criterion that is usually implemented.

Linearizability: An execution σ is linearizable if it is equivalent to a sequential execution such that *(a)* the request and the response of each operation occur both at some time t, and *(b)* t is in the interval $[t_{req}, t_{res}]$, where t_{req} is the time when the request is issued in σ, t_{res} is the time when the response is received in σ. We explain this definition on two examples. A formal definition can be found in [16].

Consider a server S that implements a *register* with the two operations *read* and *write*:

- $S.write(v)$ denotes the request to write value v in the register managed by server S. The operation returns an empty response, denoted by *ok*.
- $S.read(\,)$ denotes the request to read the register managed by server S. The operation returns the value read.

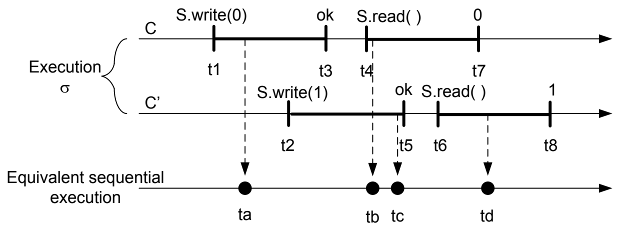

Fig. 1. A linearizable execution

Figure 1 shows an execution σ that is linearizable:

- Client C issues the request $write(0)$ at time t_1, and receives the empty response *ok* at time t_3.
- Client C' issues the request $write(1)$ at time t_2, and receives the empty response *ok* at time t_5.
- Client C issues the request $read(\,)$ at time t_4, and receives the response 0 at time t_7.
- Client C' issues the request $read(\,)$ at time t_6, and receives the response 1 at time t_8.

The bottom time-line in Figure 1 shows a sequential execution equivalent to σ that satisfies the two requirements *(a)* and *(b)* above (t_a is in the interval $[t_1, t_3]$, t_b is in the interval $[t_4, t_7]$, etc.).

Figure 2 shows an execution that is not linearizable. In an equivalent sequential execution $write(1)$ issued by C' must precede $read(\,)$ issued by C. So there is no way to construct a sequential execution in which $read(\,)$ returns 0 to C.

2.3 Linearizability *vs.* isolation

Linearizability differs from the isolation property of transactions. There are two main differences. First, linearizability is defined on the *whole sequence of operations* issued by a client process in the system, while isolation is defined on a

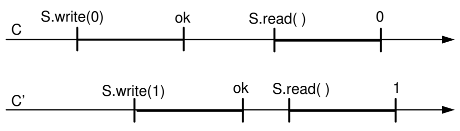

Fig. 2. A non linearizable execution

subset of the operations of a client process. Consider for example that process p issues operations op_1 and op_2 within transaction T_1, and later operations op_3 and op_4 within transaction T_2. Isolation does not require that the operations of T_1 are ordered before the operations of T_2 (they can be ordered after those of T_2). However, if op_i precedes op_j on process p, then linearizability requires op_i to be ordered before op_j.

The second difference is that linearizability does not ensure isolated execution of a sequence of operations. If process p issues operation $op_p^1 = S.read(\,)$ that returns v and later $op_p^2 = S.write(v{+}1)$, and process q issues operation $op_q^1 = S.read(\,)$ that returns v' and later $op_q^2 = S.write(v'{+}1)$, linearizability does not prevent the operation op_q^1 of q to be executed between the two operations op_p^1 and op_p^2 of p. There are basically two ways to prevent this from occurring. The first solution is for p and q to explicitly use locks or semaphores. The second solution is to add a new operation to the server S, e.g., *increment*, and to invoke this single operation instead of *read* followed by *write*. The second solution is better than the first one (locks and semaphores lead to problems in the presence of failures).

2.4 Replication techniques

In the previous section, linearizability defined the desired semantics for operations issued by clients on servers. In the definition of linearizability, servers are black boxes. This means that the definition applies to non-replicated single-threaded servers, to non-replicated multi-threaded servers, to replicated single-threaded servers and to replicated multi-threaded servers. In this section we address the question of implementing a replicated server while ensuring linearizability. We discuss only the single-threaded case. The two main replication techniques are called *active replication* and *passive replication*. Other replication techniques can be seen as variants or combinations of these two basic techniques.

Active replication: Active replication is also called *state-machine replication* [18, 28]. The principle is illustrated on Figure 3, which shows a replicated server S with three replicas S_1, S_2 and S_3. The client sends its request to all the replicas, each replica processes the request and sends back the response to the client. The client waits for the first response and ignores the others. This client's behavior is correct if we assume that the servers do not behave maliciously, and the servers are deterministic:[1] in this case all the responses are identical.

In Figure 3 there is only one client. The problem becomes more difficult with multiple clients that concurrently send their requests. In this case it is sufficient that all replicas S_i receive the clients' requests in the *same order*, as shown in Figure 4. This allow the replicas to process the clients' requests in the same order. In Section 3 we introduce a group communication primitive that ensures such an ordering of client requests.

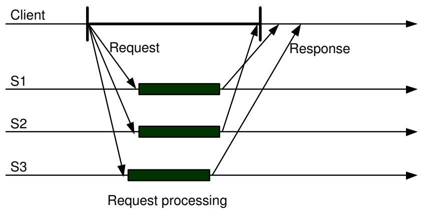

Fig. 3. Active replication

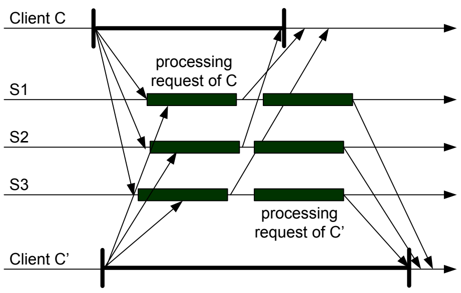

Fig. 4. Active replication: requests received in the same order

Passive replication: The principle of passive replication is illustrated on Figure 5, which shows the same replicated server S with its three replicas S_1, S_2 and S_3. One of the replicas, here S_1, is the *primary* replica; the other replicas, S_2 and S_3 are called *backups*. The client sends its request only to the primary, and waits for the response. Only the primary processes the request. Once this is done, the primary sends an *update* message to the backups, to bring them to a state that reflects the processing of the client request. In Figure 5 the update message is also sent to the primary. The reason is that, if we include failures, it is simpler to assume that the modification of the state of the primary occurs only upon handling of the update message, and not upon processing of the request.

If several clients sent their requests at the same time, the primary processes them sequentially, one after the other. Since the primary sends an update message to the backups, the processing can be non-deterministic, contrary to active replication. Note that this superficial presentation hides most of the problems related to the implementation of passive replication. We mention them in the next paragraph. With active replication, the implementation problems are hidden in the implementation of the group communication primitive that orders the clients' requests.

[1] A server is deterministic if its new state and the response depend only on the request and on the state before processing the request.

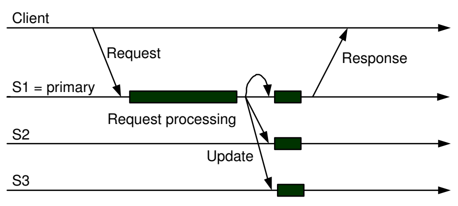

Fig. 5. Passive replication

Problems implementing passive replication: When the primary crashes, a new primary must be selected. However, requiring the failure detection of the primary to be reliable (i.e., never making mistakes) is a very constraining assumption. For this reason, solutions to passive replication that do no require a reliable failure detection mechanism for the primary have been developed. The three main problems to address are the following: (a) prevention of multiple primaries being able to process requests, *(b)* prevention of multiple executions of a request, and *(c)* reception of the update message by all replicas. Problem *(a)* is related to the unreliable failure detection mechanism. Problem *(b)* arises when the current primary is falsely suspected to have crashed. Consider a client C sending its request to the primary S_1. Assume that S_1 is incorrectly suspected to have crashed, and S_2 becomes the new primary. If this happens, and C did not receive any response, it will resend its request to S_2. This may lead to execute the client request twice. Multiple execution of a request can be prevented by attaching a unique identifier to each request (this request identifier being piggybacked on the update message). Problem *(c)* arises when the primary crashes while multicasting the update message. In this case, we must prevent the undesirable situation where the update message is received by some replicas, but not by all of them. In Section 3 we present the group communication primitive that allows us to solve the problems *(a)* and *(c)*.

3 Group communication

In the previous section we have introduced the two basic replication techniques, namely active replication and passive replication. We have also pointed out the need for communication primitives with well defined ordering properties to implement these techniques. *Group communication* is the infrastructure that provides these primitives. A group is simply a set of processes with an identifier. Messages can be multicast to the members of some group g simply by referring to the identifier of group g: the sender of the message does not need to know what processes are members of g. For example, if we consider a replicated server S with three replicas S_1, S_2 and S_3, we can refer to these replicas as the group $g_S = \{S_1, S_2, S_3\}$. As illustrated by Figure 6, group communication is a middleware layer between the transport layer and the layer that implements replication. In this section we define the two main group communication primitives for replication, namely *atomic broadcast* and *generic broadcast*. Before doing so, we introduce some concepts needed to understand the various aspects of group communication.

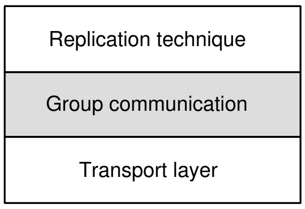

Fig. 6. Group communication

3.1 Various group models

Static group *vs.* dynamic group: A *static* group is a group whose membership is constant over time: a static group is initialized with a given membership, and this membership never changes. This is the simplest type of group. However, static groups are often too restrictive. For example consider the replicated server S implemented by the group $g_S = \{S_1, S_2, S_3\}$. If one of the replicas S_i crashes, it might be desirable to replace S_i with a new replica, in order to maintain the same degree of replication. A group whose membership changes over time is called a *dynamic* group. Dynamic groups require to manage the addition and the removal of members to/from the group. This problem is called the *group membership* problem: it is discussed in Section 3.4.

Benign *vs.* malicious faults: The group (or system) model encompasses also the type of faults that are considered. The distinction is made between *benign* faults and *malicious* faults (also called *Byzantine* faults). With benign faults, a process or a channel does its job correctly, or does not do its job. A process crash, or a channel that looses a message, are benign faults. With malicious faults, a process or a channel can behave arbitrarily.

Crash-stop *vs.* crash-recovery: In the context of benign faults, the distinction is made between the crash-stop and the crash-recovery process model. In the *crash-stop* model processes do not have access to stable storage. In this case, a process that crashes looses its state: upon recovery, the process is indistinguishable from a newly started process. In the *crash-recovery* model processes have access to stable storage, allowing them to periodically save their state. In this case, a process that crashes can recover its most recently saved state.

Combining these models: Combining these three dimensions lead to different models for group communication. The simplest model is the benign static crash-stop model. Other models have been considered in the literature, but they lead to more complexity in the specification of group communication and in the algorithms. There are some subtle differences between the different models, as we explain now.

Figure 7 shows the difference between active replication with dynamic crash-stop groups (left) and active replication with static crash-recovery groups (right). In the crash-stop model, to keep the same replication degree, a crashed process (here replica S_3) must be replaced with a new process (here S_4). The initial membership of the group g_S is denoted by $v_0(g_S) = \{S_1, S_2, S_3\}$ (v stands for *view*, see Section 3.4). When S_3 crashes, the membership becomes $v_1(g_S) = \{S_1, S_2\}$. Once S_4 is added, we have the membership $v_2(g_S) = \{S_1, S_2, S_4\}$. Note that the state of p_4 must be initialized. This is done by an operation called *state transfer*: when S_4 joins the group, the state of one of its members (here S_2) is used to initialize the state of S_4. In the static crash-recovery model (Figure 7, right), the same degree of replication is kept by assuming that crashed replicas recover (here S_3). However in this context, since S_3 remains all the time a member of g_S, a message broadcast to the group while S_3 is down *must be delivered to* S_3 (here m_2). As a result, no state transfer is needed. The static crash-recovery model is preferable to the dynamic crash-stop model whenever the state of the replicas is large.

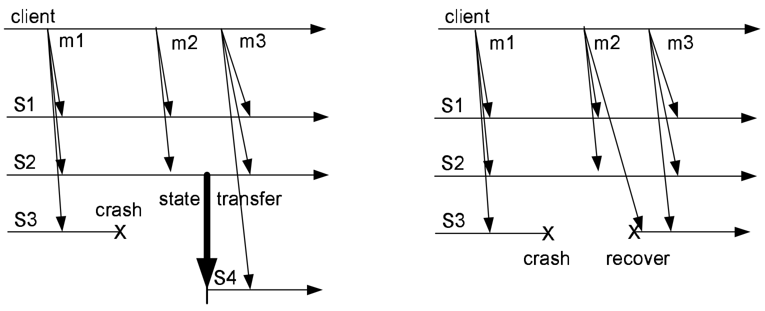

Fig. 7. Active replication with a dynamic crash-stop group (left), or a static crash-recovery group (right)

In the following we consider mainly the static crash-stop model, which is the most widely model considered in the literature, and the simplest. Dynamic groups are briefly mentioned in Section 3.4.

3.2 Atomic broadcast for active replication

One of the most important group communication primitives is *atomic broadcast* [8]. Atomic broadcast is also sometimes called *total order broadcast*, or simply *abcast*. The primitive ensures that messages are delivered ordered. To give a more formal specification of the properties of abcast, we need to introduce the following notation:

- The atomic broadcast of message m to the members of some group g is denoted by $abcast(g, m)$.[2]
- The delivery of message m is denoted by $adeliver(m)$.

It is important to make the distinction between *abcast/adeliver*, and the *send/receive* primitives at the transport layer (see Figure 8). The semantics of *send/receive* is defined by the transport layer. The semantics of *abcast/adeliver* is defined by atomic broadcast. An atomic broadcast protocol uses the semantics of *send/receive* to provide the semantics of *abcast/adeliver*.

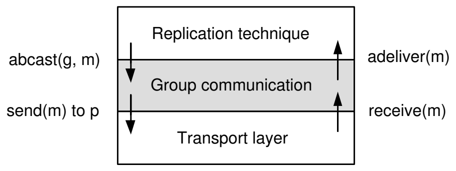

Fig. 8. *Send/receive* vs. *abcast/adeliver*

The definition of atomic broadcast in the static crash-stop model relies on the definition of a *correct* process: a process is correct if it does not crash. Otherwise it is *faulty*. Note that even though these definitions are simple, they are easily wrongly understood. Correct/faulty are predicates that characterize the *whole lifetime* of a process. This means that if some process p crashes at time $t = 10$, then p is faulty (even at time $t = 9$). With this definition, atomic broadcast in the static/crash-stop model is specified by the following four properties [15, 2]:[3]

- *Validity:* If a correct process executes *abcast(g,m)*, then some correct process in g eventually adelivers m or no process in g is correct.
- *Uniform agreement:* If a process in g adelivers a message m, then all correct processes in g eventually adeliver m.
- *Uniform integrity:* For any message m, every process p adelivers m at most once, and only if p is in g and m was previously abcast to g.
- *Uniform total order:* If process p in g adelivers message m before message m', then no process in g adelivers m' before having previously adelivered m.

[2] The primitive should be called atomic *multicast*. For simplicity, we keep the term *broadcast* here.

[3] More precisely, the specification corresponds to the primitive called *uniform atomic broadcast*. We will call it here simply *atomic broadcast*.

Validity, uniform agreement and uniform integrity define the primitive called *reliable broadcast.*[4] Atomic broadcast is defined as reliable broadcast with the uniform total order property.

It is easy to see that active replication is easily implemented using atomic broadcast. If g_S is the group of replicas that provide some service S, clients C send requests using the primitive $abcast(g_S, req)$. The validity property ensures that if C does not crash, its request is received by at least one member of g_S (unless all members of g_S crash). Combining this guarantee with uniform agreement ensures that all correct processes in g_S eventually adeliver m. The uniform total order property ensures that all replicas adeliver the clients' requests in the same order.

The response from a replica in g_S to a client is sent using a unicast message, i.e., a point-to-point message. The transport layer must ensure the following *quasi-reliable channel* property [1]: if a correct process p sends message m to a correct process q, then q eventually receives m. This property is stronger than the property provided by TCP (if a TCP connection breaks, reliability is no more guaranteed).

3.3 Generic broadcast for passive replication

Atomic broadcast can also be used to implement passive replication, but this is not necessarily the best solution in terms of cost. Atomic broadcast can be used as follows. Consider a replicated server S defined by the (static) group g_S, and assume that the members of g_S are ordered in a list. Initially, the member at the head of the membership list is the primary. The primary sends the update message to g_S using abcast. Whenever some member of g_S suspects the current primary to have crashed, it abcasts the message $\langle$*primary change*$\rangle$. Upon adelivery of this message every process moves the process at the head of the list to the tail. The new primary is the new process at the head of the list.

Passive replication can also be implemented using the group communication primitive called *generic broadcast* [25, 2], which can be cheaper to implement than atomic broadcast. While atomic broadcast orders *all* messages, generic broadcast orders only messages that *conflict.* Conflicts are defined by a relation on the set of messages. This conflict relation is part of the specification of the primitive, and makes the primitive *generic.* The generic broadcast of message m to the group g is denoted by $gbcast(g, m)$; the delivery of message m is denoted by $gdeliver(m)$. Formally, generic broadcast is defined by the same properties that define atomic broadcast, except that the uniform total order property is replaced with the following weaker property:

– *Generic total order:* If process p in g gdelivers message m before message m', and m, m' conflict, then no process in g gdelivers m' before having previously gdelivered m.

[4] More precisely, *uniform* reliable broadcast.

We have seen that passive replication can be implemented with atomic broadcast for the *update* messages and the *primary-change* messages. Consider the following conflict relation between these two types of messages:

– Messages of type *primary-change* do not conflict with messages of the same type, but conflict with messages of type *update*.
– Messages of type *update* conflict with messages of the same type, and also with messages of type *primary-change*.

This ensures enough ordering to implement generic broadcast correctly. Note that most of the time one single process considers itself to be the primary, and during this period no concurrent update messages are issued. So most of the time no concurrent conflicting messages are issued.

The implementation of generic broadcast (and atomic broadcast) is discussed in Section 4.

3.4 About group membership

With dynamic groups, the successive membership of a group is called a *view*. Consider for example a group g, with initially three processes p, q, r. This initial membership is called the *initial view* of g, and is denoted by $v_0(g)$. Assume that later r is removed from g. The new membership is denoted by $v_1(g) = \{p, q\}$. If s is added later to the group the resulting membership is denoted by $v_2(g) = \{p, q, s\}$. So the history of a dynamic group is represented as a sequence of views, and all group members must see the sequence of views in the same order. The problem of maintaining the membership of a dynamic group is called the *group membership problem* [27].

3.5 About view synchronous broadcast

View synchronous broadcast or *vscast* (sometimes also called *view synchrony*), is another group communication primitive, defined in a dynamic group model [4, 7]. However, the importance of vscast has been overestimated, and stems from a time where the difference between static groups and dynamic groups was not completely understood.

Consider some message m vscast by process p in view $v_i(g)$: vscast orders m with respect to view changes. In other words, vscast ensures that m is delivered by all processes in the same view v_j. The property is also called *same view delivery* [7]. A stronger property, called *sending view delivery*, requires $i = j$: the view in which the message is delivered is the view in which the message was sent [7].

The overestimated importance given to view synchronous broadcast has led to several misunderstandings. The first is that dynamic groups are needed to implement passive replication: Section 3.3 has sketched an implementation of passive replication with a static group. The second misunderstanding is that the specification of group communication with dynamic groups is inherently different from the specification of group communication with static groups. This is not the case, as shown in [26].

3.6 Group communication *vs.* quorum systems

In the previous sections we have shown the use of group communication for implementing replication. *Quorum systems* is another technique for replication, anterior to group communication and also more widely known. In this section we explain the advantage of group communication over quorum systems in the context of replication [11].

Definition of quorum systems: Consider a set $\Pi = \{p_1, \ldots, p_n\}$ of processes. The set of all subsets of Π is called the *powerset* of Π, and is denoted by 2^{Π}. We have for example:

$$\{p_1\}, \{p_2\}, \{p_1, p_2\}, \{p_2, p_3, p_4\}, \ldots, \{p_1, \ldots, p_n\} \in 2^{\Pi}.$$

A *quorum system* of Π is defined as any set $Q \subset 2^{\Pi}$ such that any two $Q_i \in Q$ have a non empty intersection:

$$\forall Q_1, Q_2 \in Q, \text{ we have } \mathrm{Q}_1 \cap \mathrm{Q}_2 \neq \emptyset.$$

Each $Q_i \in Q$ is called a *quorum*. For example, if $\Pi = \{p_1, p_2, p_3\}$, then the set $Q = \{\{p_1, p_2\}, \{p_1, p_3\}, \{p_2, p_3\}\}$ is a quorum system of Π; $\{p_1, p_2\}$, $\{p_1, p_3\}$, $\{p_2, p_3\}$ are quorums.

Quorum systems for implementing a fault tolerant register: The use of quorums systems for fault tolerance can be illustrated on a very simple example: a server that implements a *register*. A register is an object with two operations *read* and *write*: *read* returns the value of the register, i.e., the most recent value written; *write* overwrites the value of the register.

The register can be made fault tolerant by replication on three replicas e.g., $\Pi = \{p_1, p_2, p_3\}$ with the quorum system $Q = \{\{p_1, p_2\}, \{p_1, p_3\}, \{p_2, p_3\}\}$. Each operation needs only to be executed on one quorum of Q, i.e., on $\{p_1, p_2\}$, on $\{p_1, p_3\}$, or on $\{p_2, p_3\}$. In other words, the quorum system Q tolerates the crash of one out of the three replicas. Using the quorum system Q, linearizability of the read and write operations is easy to implement [11].

Requiring isolation: A fault tolerant register is easy to implement using quorum systems. However, clients usually want to perform more complex operations. Consider for example the operations *(a) increment* a register and *(b) decrement* a register. These two operations can be implemented as follows: (1) read the register, then (2) update the value read, and finally (3) write back the new value. However, one client C may increment the register, while at the same time another client C' decrements the register. To ensure a correct execution, the two operations must be executed in mutual exclusion. With group communication, no mutual exclusion is needed: atomic broadcast can be used to send the corresponding operation to the replicated servers.

This difference between quorum systems and group communication is illustrated in Figure 9. The left part illustrates the quorum solution, and the right part the group communication solution. In the quorum solution, the increment operation is performed by the client, after reading the register and before writing the new value. The implementation requires mutual exclusion, represented by E_{CS} (enter critical section) and L_{CS} (leave critical section). In the group communication solution, the increment operation is sent to the replicas using atomic broadcast; no mutual exclusion is required.[5] Implementing atomic broadcast requires weaker assumptions about the crash detection mechanism than implementing mutual exclusion [11].

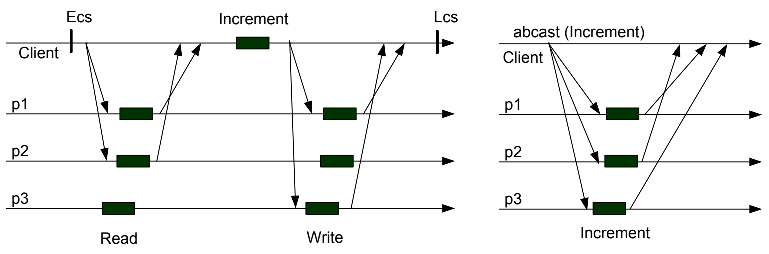

Fig. 9. Replication: quorum systems (left) *vs.* group communication (right)

4 Implementation of group communication

In the previous section we have seen the role of group communication for replication. We discuss now the implementation of the two group communication primitives that we have introduced, namely atomic broadcast and generic broadcast. We consider only static groups, non Byzantine processes and the crash-stop model.

4.1 Impossibility results

Consider a static group g, and processes in g communicating by message exchange. The most general assumption is to consider that the time between the sending of a message m and the reception of m by its destination is not bounded, i.e., the transmission delay can be arbitrarily long. Similarly, if we model the execution of a process as a sequence of steps, the most general assumption is to consider that while the slowest process performs one step, the fastest process can perform an unbounded number of steps. These two assumptions define the *asynchronous system model*. The absence of bounds for the message transmission

[5] The reader may wonder why no *increment* operation can be sent with quorum systems. Sending the *increment* operation requires atomic broadcast!

delay models an open network in which the load of the links are unknown. The absence of bounds on the relative speed of processes models processes running on CPUs with an unknown load. The asynchronous system model is the most general model, but it has a major drawback: several problems are impossible to solve in that model when one single process may crash.

One of these problems is *consensus*. The problem is defined on a set of processes, e.g., on some group g. Every process p in g starts with an initial value v_p, and all correct processes in g have to decide on some common value v that is the initial value of one of the processes. Formally, the consensus problem is defined by the following properties [6]:

- *Validity:* If a process decides v, then v is the initial value of some process.
- *Agreement:* No two correct processes decide differently.
- *Termination:* Every correct process eventually decides some value.

An explanation of *problem solvability* is needed here. Consider a distributed algorithm A_P that is supposed to solve problem P. Algorithm A_P can be launched many times. Due to the variability of the transmission delay of messages, each execution of A_P can go through a different sequence of states. However, in all of these executions, A_P must solve P. If there is one single execution in which this is not the case, then we say that algorithm A_P does not solve P. This clarification is important in the context of the consensus problem: it has been shown that consensus is not solvable by a deterministic algorithm in an asynchronous system with reliable links if one single process may crash. This result is known as the *FLP impossibility result* [13].

The FLP impossibility result is easy to extend to atomic broadcast by the following argument [9]. Assume for a contradiction that atomic broadcast can be implemented in an asynchronous system with process crashes. Then consensus can be solved as follows (in the context of some group g):

- Each process p in g executes $abcast(v_p)$, where v_p is p's initial value.
- Let v be the first message adelivered by p.
- Process p decides v.

If there is a least one correct process, then at least one message is adelivered. By the property of atomic broadcast, every correct process adelivers the same first message, and so decides on the same value. Consensus is solved, which shows the contradiction.

4.2 Models for solving consensus

Consensus and atomic broadcast are not solvable in an asynchronous system when processes may crash. We thus need to find a system model in which consensus is solvable (whenever consensus is solvable, atomic broadcast is also solvable, see Section 4.3). One such system is the *synchronous* system model, defined by the following two properties:

– There is a known bound on the transmission delay of messages.
– There is a known bound on the relative speed of processes.

Consensus is solvable in a synchronous system [23], but the synchronous system model has drawbacks from a practical point of view. The model requires to consider the *worst case*: the worst case for the transmission delay of messages, the worst case for the relative speed of processes. These bounds have a direct impact on the time it takes to detect the crash of a process: the higher these bounds are, the higher the time it takes to detect a process crash, i.e., the longer it takes to react to a crash. In a replicated service a long reaction to a crash leads to a long delay before clients get the replies.

The drawback of the synchronous model has led to look for system models weaker than the synchronous model, but strong enough to solve consensus (and so atomic broadcast). The first of these models is called the *partially synchronous* model [10]. The model considers bounds on the message transmission delay and on the relative speed of processes. There are two variants of the model:

1. There is a bound on the relative speed of processes and a bound on the message transmission delay, but these bounds are *not known.*
2. There is a *known* bound on the relative speed of processes and on the message transmission delay, but these bounds hold only from some unknown point on.

The two definitions are equivalent, but the first variant seems more appealing from a practical point of view.

A different approach was proposed later in [6]. It consists in *augmenting* the asynchronous model with an *oracle* that satisfies some well defined properties. In other words, the system is assumed to be asynchronous, but the processes can query an oracle about the status *crashed/not crashed* of processes. For this reason the oracle is called *failure detector oracle*, or simply *failure detector*. If the failure detector returns the reply *crashed* q to process p, we say p *suspects* q. Note that this information may be incorrect: failure detectors can make mistakes. The legal replies to a query of the failure detector are defined by two properties called *completeness* and *accuracy.* For example, the replies of the failure detector called $\diamond\mathcal{S}$ must satisfy the following completeness and accuracy properties [6]:

– *Strong completeness:* Eventually every process that crashes is permanently suspected to have crashed by every correct process.
– *Eventual weak accuracy:* There is a time after which some correct process is never suspected by any correct process.

Consensus is solvable in the asynchronous system augmented with the failure detector $\diamond\mathcal{S}$ and a majority of correct processes [6]. Moreover, it has been shown that $\diamond\mathcal{S}$ is the weakest failure detector that allows us to solve consensus in an asynchronous system [5]. This result shows the power of the failure detector approach and explains its popularity.

4.3 Solving consensus

The first algorithm to solve consensus in a model weaker than the synchronous model is the consensus algorithm by Dwork, Lynch and Stockmeyer for the partially synchronous model [10]. The algorithm – called here *DLS* – requires a majority of correct processes, and is based on the *rotating coordinator* paradigm. In this paradigm, the computation is decomposed into rounds $r = 0, 1, 2, \ldots$, and in each round another process, in some predetermined order, is the coordinator. Typically, with n processes $p_0, \ldots p_{n-1}$, the coordinator of round r is process $p_{r \, mod \, n}$. In each round the coordinator leads the computation in order to try to decide on a value. The algorithm is based on the notions of *locked* value and *acceptable* value. The coordinator of round r tries to lock a value, say v, and if it learns that a majority of processes have locked v in round r, it can decide v. If the coordinator of round r is suspected to have crashed, then the computation proceeds to the next round $r + 1$ with a new coordinator. Note that a process can become coordinator more than once, e.g., in rounds k, $n + k$, $2n + k$, etc. The key property of the *DLS* algorithm is that the safety properties of consensus (validity and agreeement) hold even if the properties of the partially synchronous model do not hold. In other words, these properties are only needed for liveness, i.e., to ensure the termination property of consensus.

Two other consensus algorithms had a major impact and led to the development of variations of these algorithms. The first one is the *Paxos* algorithm proposed by Lamport [20, 21]. The second one is the Chandra-Toueg consensus algorithm (denoted *CT* hereafter) based on the failure detector $\diamond\mathcal{S}$ [6]. *Paxos* and *CT*, similarly to *DLS*, require a majority of correct processes. *CT*, similarly to *DLS*, is based on the rotating coordinator paradigm. *Paxos* is also based on a coordinator, but the coordinator role is not predetermined as in the rotating coordinator paradigm, but determined during the computation (the algorithm tolerates multiple coordinators for the same round). *Paxos* and *CT* are also based on the notion of *locked* value (but there is no notion of *acceptable* value): each coordinator, one after the other, tries to lock a value v, and if it learns that a majority of processes have locked v, it can decide v. In this sense *Paxos* and *CT* are very similar. The two algorithms also share the key property of *DLS*, namely that no matter how asynchronous the system behaves, the safety properties of consensus are never violated. However, *Paxos* and *CT* differ on the following issues:

– *CT* requires reliable channels, while *Paxos* tolerates message loss (similarly to *DLS*).
– The condition for termination is rigorously defined for *CT*, namely the *eventual weak accuracy* property of $\diamond\mathcal{S}$. No such condition that ensure termination exists for *Paxos*.

Note that after the publication of *Paxos*, the failure detector Ω – which eventually outputs at each process the identity of the same correct process [5] – has been mentioned as ensuring the termination of *Paxos*. However, this makes sense only if we consider *Paxos* with reliable channels.

4.4 Implementing atomic broadcast and generic broadcast

A large number of atomic broadcast algorithms have been proposed in the last 20 years. These algorithms can be classified according to several criteria. One of those criteria is the mechanisms used for message ordering [8]: *fixed sequencer*, *moving sequencer*, *privilege-based*, *communication history*, *destinations agreement*. For example in a *fixed sequencer* algorithm, one process is elected as the sequencer and is responsible for ordering messages. Obviously this solution is not tolerant to the crash of the sequencer. The solution must be completed by a mechanism for electing a new sequencer in case the current sequencer crashes. This is usually done using a group membership service (see Section 3.4) to remove the current sequencer from the group. Once this is done, a new sequencer can be elected. Thus the solution implements atomic broadcast in the context of dynamic groups (see Section 3.1). The same comment applies to most of the implementations of atomic broadcast described in the literature. These implementations *require order to provide order*: the group membership service orders views, and this order is used to implement the ordering required by atomic broadcast.

Atomic broadcast can also be solved in the context of static groups. The solutions rely on consensus (which explains the fundamental role of the consensus problem in the context of fault tolerance computing). The consensus problem allows processes to agree on a value. This value can be of any type. Atomic broadcast can be implemented by solving a sequence of consensus problems, where each instance of consensus agrees on a *set of messages*. The idea is the following [6]. Consider a static group g and $abcast(g, m)$. Each process p in g has a variable k_p used to number the various instances of consensus. Whenever p has received messages that need to be ordered, p starts a new instance of consensus, uniquely identified by k_p, with the set of messages to be ordered as its initial value. By the properties of consensus, all processes agree on the same set of messages for consensus $\#k_p$, say $msg(k_p)$. Then the messages in the set $msg(k_p)$ are adelivered in some deterministic order (e.g., according to their IDs), and before the messages in the set $msg(k_p + 1)$. This solution for static groups can be extended to dynamic groups [26].

The implementation of generic broadcast is more difficult to sketch. The basic idea of the implementation is to control whether conflicting messages have been gbcast. As long as only non conflicting messages are gbcast, these messages can be gdelivered without invoking consensus, i.e., without the cost of consensus. However, as soon as conflicting messages are detected, the gdelivery of messages require to execute an instance of the consensus problem. More details can be found in [25, 2].

4.5 Solving the atomic commitment problem

In Section 1 we have mentioned the *atomic commitment* problem as the main problem related to the implementation of distributed transactions. The problem has similarities with the consensus problem, but also has significant differences.

In the atomic commitment problem, each process involved in the transaction votes at the end of the transaction. The vote can be *yes* or *no*. A *yes* vote indicates that the process is ready to commit the transaction; a *no* vote indicates that the process cannot commit the transaction. As in the consensus problem, all processes must decide on the same outcome: *commit* or *abort*. The conditions under which commit and abort can be decided make the difference between consensus and atomic commitment. If one single process votes *no*, the decision must be *abort*; if no failure occurs and all processes vote *yes*, then the decision must be *commit*; if there are failures, the decision can be *abort*. So "failures" can influence the decision of atomic commitment, which is not the case for consensus.

Another important difference is that, for practical reasons, the atomic commitment problem needs to be solved in the crash-recovery model (in the context of transactions, processes have access to stable storage). A third difference is related the notion of *blocking* vs. *non-blocking* solution, a difference that has not been made for consensus (the distinction between a blocking and a non-blocking solution exists only in the crash-recovery model). In the crash-recovery model, a protocol is *blocking* if a single crash during the execution of the protocol prevents the termination of the protocol until the crashed process recovers. In contrast, a non-blocking protocol can terminate despite one single process crash (or even despite more than one crash).

The most popular atomic commitment protocol is the blocking *2PC* (2 Phase Commit) protocol [3]. The first non-blocking atomic commitment protocol was proposed by Skeen [29]. At that time the consensus problem was not yet identified as the key problem in distributed fault tolerant computing. This explains that the protocol proposed in [29] does not solve atomic commitment by reduction to consensus. Today such a reduction is considered to be the best way to solve the non-blocking atomic commitment problem (see for example [14], for a solution in the crash-stop model).

5 Conclusion

More than twenty years of research have contributed to a very good understanding of many issues related to fault tolerance, replication and group communication. However, the understanding of theoretical issues is not the same in all models. For example, while static group communication in the crash-stop model has reached maturity, the same level of maturity has not yet been reached for dynamic group communication or for group communication in the crash-recovery model. More work needs also to be done to quantitatively compare different algorithms in the context of replication. Typically, while a lot of atomic broadcast algorithms have been published, little has been done to compare these algorithms from a quantitative point of view. Specifically, more work needs to be done to compare these algorithms under different fault-loads, as done for example in [30]. Addressing real-time constraints, e.g., [17], needs also to get more attention.

Acknowledgments. I would like to thank Sergio Mena and Olivier Rütti for their comments on an earlier version of the paper.

References

1. M. K. Aguilera, W. Chen, and S. Toueg. Heartbeat: a timeout-free failure detector for quiescent reliable communication. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG'97)*, pages 126–140, Saarbrücken, Germany, September 1997.
2. M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Thrifty generic broadcast. In *Proceedings of the 14th International Symposium on Distributed Computing (DISC'2000)*, October 2000.
3. P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Distributed Database Systems*. Addison-Wesley, 1987.
4. K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Trans. on Computer Systems*, 5(1):47–76, February 1987.
5. T. D. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. *Journal of ACM*, 43(4):685–722, 1996.
6. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2):225–267, 1996.
7. G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys*, 4(33):1–43, December 2001.
8. X. Défago, A. Schiper, and P. Urban. Totally Ordered Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Computing Surveys*, 4(36):1–50, December 2004.
9. D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchrony needed for distributed consensus. *Journal of ACM*, 34(1):77–97, January 1987.
10. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–323, April 1988.
11. Richard Ekwall and André Schiper. Replication: Understanding the Advantage of Atomic Broadcast over Quorum Systems. *Journal of Universal Computer Science*, 11(5):703–711, May 2005.
12. E.N. Elnozahy, L. Alvisi, Y-M. Wang, and D.B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.
13. M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of ACM*, 32:374–382, April 1985.
14. R. Guerraoui, M. Larrea, and A. Schiper. Reducing the cost for Non-Blocking in Atomic Commitment. In *IEEE 16th Intl. Conf. Distributed Computing Systems*, pages 692–697, May 1996.
15. V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. Technical Report 94-1425, Department of Computer Science, Cornell University, May 1994.
16. M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. on Progr. Languages and Syst.*, 12(3):463–492, 1990.
17. J.-F. Hermant and G. Le Lann. Fast Asynchronous Uniform Consensus in Real-Time Distributed Systems. *IEEE Transactions on Computers*, 51(8):931–944, August 2002.
18. L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. ACM*, 21(7):558–565, July 1978.
19. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C28(9):690–691, 1979.
20. L. Lamport. The Part-Time Parliament. TR 49, Digital SRC, September 1989.

21. L. Lamport. The Part-Time Parliament. *ACM Trans. on Computer Systems*, 16(2):133–169, May 1998.
22. J.C. Laprie, editor. *Dependability: Basic Concepts and Terminology*. Springer-Verlag, 1992.
23. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
24. J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Trans. on Progr. Languages and Syst.*, 8(1):142–153, 1986.
25. F. Pedone and A. Schiper. Handling Message Semanticas with Generic Broadcast Protocols. *Distributed Computing*, 15(2):97–107, April 2002.
26. A. Schiper. Dynamic Group Communication. TR IC/2003/27, EPFL, April 2003. To appear in ACM Distributed Computing.
27. A. Schiper and S. Toueg. From Set Membership to Group Membership: A Separation of Concerns. TR IC/2003/56, EPFL - IC, September 2003.
28. F. B. Schneider. Implementing Fault Tolerant Services Using the State Machine Approach: A Tutorial. *Computing Surveys*, 22(4):299–319, December 1990.
29. D. Skeen. Nonblocking Commit Protocols. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 133–142, 1981.
30. Péter Urbán, Ilya Shnayderman, and André Schiper. Comparison of failure detectors and group membership: Performance study of two atomic broadcast algorithms. In *Proc. Int'l Conf. on Dependable Systems and Networks*, pages 645–654, San Francisco, CA, USA, June 2003.