A Generic Multicast Primitive to Support Transactions on Replicated Objects in Distributed Systems *

Rachid Guerraoui André Schiper
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland

Abstract

Locking and atomic commitment are two fundamental problems underlying transaction management in fault-tolerant distributed systems. In this paper we discuss these problems in the context of an asynchronous system with replicated objects. We present a generic primitive, named Dynamic-Terminating-Multicast *(or* DTM*) of which instances enable to implement a locking protocol and an atomic commitment protocol. By unifying the mechanisms needed to implement these protocols, we believe that the* DTM *primitive can be considered as a basic building block of distributed transactional systems. We show that this primitive can be implemented in an asynchronous system based on an unreliable failure detector.*

1 Introduction

The transaction concept has proven to be very convenient for the design and implementation of fault-tolerant distributed applications [1]. A transaction can be viewed as an *atomic* group of object invocations. Transaction *atomicity* implies both *concurrency atomicity* and *failure atomicity*. Concurrency atomicity (also called *isolation* or *serializability*) means that the intermediate effects of a transaction on objects are invisible to other transactions. Failure atomicity (also called *all-or-nothing* property) means that either a transaction completes successfully, or is aborted and has no effect on the accessed objects.

Transactional systems ensure atomicity either through pessimistic or optimistic approaches. Most pessimistic approaches are based on a *locking* protocol [10] that guarantees concurrency atomicity, and on an *atomic commitment* protocol [14] that guarantees failure atomicity. Optimistic approaches exclude locking but add a *certification* step [1] to the *atomic commitment* protocol, which guarantees both *concurrency atomicity* and *failure atomicity* properties. In this paper, we discuss the problems underlying fault-tolerant locking and fault-tolerant (i.e non-blocking) atomic commitment protocols in the context of a distributed system where objects accessed by transactions are replicated. We consider an asynchronous system where no assumption is made about communication delays. One major characteristic of asynchrony is that a process that has failed cannot be distinguished from a process that is just very slow. This characteristic lies at the heart of the so called *FLP* (Fischer, Lynch, Paterson) impossibility result [11] which states that, in an asynchronous system with a single faulty process, no consensus among processes can be solved by a deterministic algorithm.

Hadzilacos has discussed the similarities between the fault-tolerant atomic commitment problem and the consensus problem [15]. He showed that the consensus problem is *reducible*[1] to the fault-tolerant atomic commitment problem. Hence, the impossibility of implementing a fault-tolerant atomic commitment protocol in an asynchronous system also holds. As we will show in this paper, the consensus problem is also reducible to the fault-tolerant locking problem. As a consequence, the locking and atomic commitment problems, on top of which most existing transactional systems are designed, do not have fault-tolerant solutions in an asynchronous system. However, Chandra and Toueg have showed that by adding a failure detector to an asynchronous system, consensus becomes solvable [6]. They show that this result holds even

*Paper appears in Proc IEEE 5th Workshop on Future Trends in Distributed Computing Systems, Cheju, Korea, August 1995. Research supported by the "Fonds national suisse" and OFES under contract number 21-32210.91, as part of the ESPRIT Basic Research Project BROADCAST (number 6360), and by PPR-IF under contract number 5003-34344.

[1] A problem A is said to be *reducible* to a problem B, if whenever B can be solved, A can also be solved.

with a weak failure detector, noted $\diamond\mathcal{W}$, which can make an infinity of false failure suspicions.

In this paper, we extend Chandra-Toueg's work by showing that the fault-tolerant locking and fault-tolerant atomic commitment problems[2] can also be solved in an asynchronous system with the unreliable failure detector $\diamond\mathcal{W}$. More precisely, we define a *generic* multicast primitive, named *Dynamic-Terminating-Multicast* (or *DTM*), of which instances enable the implementation of a fault-tolerant locking protocol and a fault-tolerant atomic commitment protocol. Roughly speaking, DTM multicasts a message m to a set of destination processes $Dst(m)$, and brings processes in $Dst(m)$ to agree on the set of replies generated by the reception of m. We show that the DTM primitive can be implemented with the failure detector $\diamond\mathcal{W}$. By unifying the mechanisms needed to implement the fundamental transactional protocols, the generic DTM primitive defines an adequate basic mechanism to support transactions.

The rest of the paper is organized as follows. Section 2 describes our model of an asynchronous system augmented with a failure detector. Section 3 presents an abstract definition of the locking and atomic commitment problems in our model. Section 4 defines the generic DTM primitive and shows that it can be implemented with the unreliable failure detector $\diamond\mathcal{W}$. Section 5 presents the instances of DTM that allow to implement a fault-tolerant locking protocol and a fault-tolerant atomic commitment protocol. Finally, Section 6 summarizes the main contributions of this paper.

2 System model

2.1 Asynchronous system

We consider a distributed system composed of a finite set of processes $S = \{p_1, p_2, \ldots, p_n\}$ completely connected through a set of channels. Communication is by message passing, asynchronous and reliable. Asynchrony means that there is no bound on communication delays. A reliable channel ensures that a message sent by a process p_i to a process p_j is eventually received by p_j, if p_i and p_j are correct (i.e. do not fail)[3]. A reliable channel can be implemented by retransmitting lost or corrupt messages. Processes fail by crashing; we do not consider Byzantine failures. A process $p_i \in S$ may (1) send a message to another process, (2) receive a message sent by another process, (3) perform some local computation, or (4) fail, which is modeled by the local event $crash_i$.

The absence of a bound on communication delays makes it difficult to handle crash failures. Chandra and Toueg [6] propose to augment an asynchronous system with a failure detector which gives hints on crashed processes. We follow their approach and assume that our asynchronous system is augmented with a failure detector. In the following, we recall abstract properties of failure detectors, which we will use to define the locking and atomic commitment problems as well as the DTM primitive.

2.2 Failure detectors

A failure detector can be viewed as an oracle. Each process has access to a local failure detector module. This module maintains a list of processes that it currently suspects to have failed. We do not make any assumption on how a failure detector is implemented. For example, the local failure detector module of a process could query other processes, and suspect those that do not respond in a timely fashion. In order to reduce the number of false suspicions, the local failure detector module can also consult other failure detector modules before suspecting a process.

Failure detectors can be abstractly characterized by their *completeness* and *accuracy* properties. Completeness requires that the failure detector eventually suspects every process that actually fails, while accuracy restricts the mistakes that a failure detector can make. According to these properties, failure detectors can be classified from the weakest to the strongest. The weakest failure detector, noted $\diamond\mathcal{W}$ (*Eventually Weak)*, is the one that guarantees (1) *weak completeness*, i.e. eventually every process that fails is permanently suspected by some correct process, and (2) *eventual weak accuracy*, i.e. there is a time after which some correct process is never suspected by any correct process. The failure detector $\diamond\mathcal{W}$ can thus make an infinity of mistakes (false failure suspicions). The strongest failure detector, noted $\mathcal{P}$ (*Perfect*), ensures (1) *strong completeness*, i.e. eventually every process that fails is permanently suspected by every correct process, and (2) *strong accuracy*, i.e. correct processes are never suspected. The failure detector $\mathcal{P}$ does not make any mistake.

In the sequel we assume the failure detector $\diamond\mathcal{S}$, which guarantees *strong completeness* (eventually every process that fails is permanently suspected by ev-

[2] In the paper we give definitions of these problems that are slightly different from the definitions usually given in the literature, i.e. in synchronous systems and without considering replication.

[3] This does not exclude link failures, if we require that any link failure is eventually repaired.

ery correct process) and *eventual weak accuracy*. According to the definition given in [6], $\diamond\mathcal{W}$ and $\diamond\mathcal{S}$ are equivalent.

3 Locking and Atomic Commitment

In this section we define the locking and atomic commitment problems in our distributed system model. We first present these problems in an intuitive way, then we define them formally.

3.1 Transactions on a replicated object

Without loss of generality and to keep the presentation simple, we consider, throughout the paper, transactions accessing *one* single replicated object O. A transaction originates at a process called *Transaction Manager* (*TM*); the TM processes are noted $T_1, T_2, \ldots, T_l$. The replicas of O, noted o_1, o_2, ..., o_n are handled by processes called *Data Managers* (*DM*), which we note $D_1, D_2, \ldots, D_n$ (D_i handles the replica o_i). A process can be both TM and DM.

A TM process performs operations on behalf of a transaction by translating these operations into messages sent to the DM processes. The aim of a locking protocol is to prevent two different TM processes from performing operations on the same object at the same time. The TM processes must first request locks on DM processes before performing any operation. The DM processes must ensure that at most one TM process acquires the lock. In other words, the DM processes must agree on one TM process: the *lock-holder*. The aim of an atomic commitment protocol is to bring the DM processes to agree on an outcome decision for a transaction (either *commit* or *abort*).

In this paper we are interested in fault-tolerant protocols. Fault-tolerant locking protocols guarantee that despite the failures of some TM or DM processes, correct DM processes eventually agree on one *lock-holder*, while fault-tolerant atomic commitment protocols guarantee that despite the failures of some DM processes, correct DM processes eventually agree on the outcome of a transaction.

The difficulty of designing fault-tolerant atomic commitment protocols has been discussed by many authors [1, 8, 20]. The two phase commitment protocol can block if the coordinator fails while all other DM processes are waiting for the outcome decision of the transaction. The three phase commitment protocol described in [1] can also be blocking if the communication delays are unpredictable, i.e. in an asynchronous system. If the difficulty of atomic commitment has been discussed, the difficulty of locking replicated objects in an asynchronous system has been underestimated. Naive implementations of locking, which consist in requiring locks on DM processes in an arbitrary way, can lead to deadlock. If locks are required in a sequential and ordered way, blocking situations can still occur: consider the case of a TM process that fails after having acquired a lock on some DM processes, preventing other TM processes from acquiring the lock. The primary-copy approach, in which only one designated DM needs to be locked, also does not work in an asynchronous system with an unreliable failure detector: the primary-copy approach requires to solve the election problem, and the election problem cannot be solved in an asynchronous system with an unreliable failure detector [18].

Intuitively, these difficulties are not surprising since the problems underlying locking and atomic commitment protocols are both agreement problems, i.e. they both require processes to reach a common decision. The so called FLP impossibility result [11] states that in an asynchronous system, there is no deterministic and fault-tolerant protocol that solves the consensus problem (a very abstract form of agreement). This leads to the conclusion that in such a system, there is no deterministic algorithm that solves the locking and atomic commitment problems[4]. Hadzilacos [15] has proved this result formally for the atomic commitment problem by showing that the consensus problem is *reducible* to the atomic commitment problem. Hence the FLP result applies also to atomic commitment. In the following, we show that this impossibility result applies also to the locking problem. To establish this result, we first formally define the locking problem in Section 3.2, and we then prove that consensus is reducible to locking. Finally, in Section 3.3, we give a formal definition of the atomic commitment problem in an asynchronous system with replicated objects.

3.2 The locking problem

A TM process T_j tries to lock a replicated object by multicasting the message *lock-request*(T_j) to the DM processes $D_1, D_2, \ldots, D_n$: we say that T_j *has requested* a lock. The locking problem consists in bringing the DM processes to agree on a *locking-value*, which is the identifier[5] of one of the TM processes: the *lock-holder*. The locking problem can formally be defined for the

[4] To simplify, we will say *locking* instead of *fault-tolerant locking*, and *atomic commitment* instead of *fault-tolerant atomic commitment*.

[5] To avoid cumbersome notation, we do not make any difference between processes and their identifiers.

set of DM processes $D_1, D_2, \ldots, D_n$ by the following conditions:

- **L-Uniform-Agreement:** All processes (correct or not) that decide, decide on the same locking value.
- **L-Termination:** Every correct process eventually decides on a locking value.
- **L-Non-Triviality:** The locking value must be the identifier of a TM that has requested the lock.

The *L-Uniform-Agreement* condition is a safety condition, ensuring that only one process can acquire a lock on a replicated object. The *L-Termination* and the *L-Non-Triviality* conditions are liveness conditions. The *L-Termination* condition guarantees that, despite failures, every correct DM process eventually terminates the locking protocol. If every DM that has decided on a locking value sends this value to the TM processes, the lock-holder will know that it has acquired the lock, and the other TM processes will know that they have not (this property is actually satisfied as long as at least one DM process is correct). The *L-Non-Triviality* condition excludes undesirable solutions to the problem, where, for example, it is always decided that the same process be the *lock-holder*, even though it did not request any lock. This condition can be viewed as a liveness condition from the application point of view.

Consensus is reducible to locking

We now show that the consensus problem is reducible to the locking problem (whenever the locking problem is solved, the consensus problem is also solved). Consider the consensus problem defined for a set of processes $S = \{p_1, p_2, .., p_n\}$: every process $p_i \in$ S initially proposes a value v_i. Processes in S must then decide on a *consensus value* v which has to be the proposition v_i of some process p_i. The consensus problem is formally defined as follows [6] [6].

- **C-Uniform-Agreement:** All processes (correct or not) that decide, decide on the same consensus value v.
- **C-Termination:** Every correct process eventually decides on the consensus value.
- **C-Non-triviality:** The consensus value v must be the proposition v_i of some process p_i.

Assuming the locking problem solvable, the protocol that solves the consensus problem is quite simple. Consider the locking problem defined on a set of DM processes $D_1, \ldots, D_n$. We can use the solution of the locking problem to solve the consensus for a set S of processes as follows:

- **lock request.** Every process $p_i \in$ S appends its proposition v_i to its identifier p_i, and multicasts *lock-request*(p_i, v_i) to the DM processes $D_1, \ldots, D_n$.
- **locking decision.** Every Data Manager D_j which solves the locking problem decides on a locking value (p_x, v_x). Process D_j then multicasts (p_x, v_x) to S.
- **consensus decision.** Every process p_i which receives (p_x, v_x) decides on the consensus value v_x.

Proposition 1. *The above protocol solves the consensus problem if at least one DM process is correct.*

PROOF. The *L-Uniform-Agreement* condition of the locking problem guarantees that all the DM processes that decide, decide on the same locking value (p_x, v_x), and hence will multicast the same (p_x, v_x) message. Therefore, all the processes in S that receive (p_x, v_x) will decide on the same consensus value v_x. The *C-Uniform-Agreement* condition of the consensus problem is thus guaranteed. The *L-Termination* condition of the locking problem guarantees that every DM process eventually decides on a *lock-holder*. If we assume that at least one DM process is correct, then every correct process in S eventually receives (p_x, v_x), and thus eventually decides on the consensus value v_x. The *C-Termination* condition of the consensus problem is thus satisfied. The *C-Non-Triviality* condition of the consensus problem also holds, because of the *L-Non-Triviality* condition of the locking problem. □

Thus, whenever the locking problem can be solved, the consensus problem can also be solved. This means that the FLP impossibility result also holds for the locking problem.

[6] In fact the given conditions define the *uniform* consensus problem. The consensus problem is defined by the (non-uniform) Agreement condition: all *correct* processes that decide, decide on the same value. It has however been shown that any algorithm solving consensus in an asynchronous system with an unreliable failure detector $\mathcal{S}$, $\diamond\mathcal{S}$ or $\diamond\mathcal{W}$, also solves the uniform consensus problem [13]. Thus the distinction between *consensus* and *uniform consensus* is not fundamental here.

3.3 The atomic commitment problem

The atomic commitment problem consists of bringing the DM processes to agree on an outcome decision for a transaction (either *commit* or *abort*). The outcome decision is based on votes (*yes* or *no*) of DM processes, depicting their ability to locally *write* the updates of the transaction. In a pessimistic approach, where a lock has been acquired by a TM process, a process D_i votes *yes* if it is able to locally perform the transaction modifications. In an optimistic approach (without locking), a DM process D_i votes *yes* if it is able to locally perform the transaction modifications, and no concurrency control conflict has been locally detected.

It is generally required [1] that the outcome of a transaction is *commit* only if the votes of *all* DM processes are *yes*. This requirement is too strong if the DM processes handle replicas of the same object[7]. In the sequel we assume a *quorum based* replication policy [12] (i.e. *write quorum/read quorum*). Each DM process is assigned a *weight*, i.e. a non-negative integer. The transaction outcome decision can be *commit* if and only if there is a (write) quorum of DM processes which have voted *yes*. For example, if all the weights are equal, the quorum is reached if a majority (i.e. more than $n/2$) of DM processes have voted *yes*.

We define the atomic commitment problem for the DM processes $D_1, D_2, .., D_n$ as follows:

- **AC-Uniform-Agreement:** All processes that decide, decide on the same outcome.
- **AC-Validity:** The outcome decision is *commit* only if there is a quorum of processes that have voted *yes*.
- **AC-Termination:** Every correct process eventually decides on the outcome.
- **AC-Non-Triviality:** The outcome decision must be *commit* if there is a quorum of processes which have never been suspected[8] and which have voted *yes*.

The *AC-Agreement* and the *AC-Validity* conditions are safety conditions. The *AC-Termination* and the *AC-Non-Triviality* conditions are liveness conditions. The *AC-Termination* condition guarantees that, despite failures, every correct process eventually terminates the atomic commitment protocol. The *AC-Non-Triviality* condition excludes the undesirable solutions where processes always decide to abort the transaction.

Remark. The above *AC-Non-Triviality* condition differs from the usual *Non-Triviality* conditions [1, 8]. Bernstein et al. [1] have considered, in the context of a synchronous system, a *Non-Triviality* condition which requires processes to decide *commit* if all votes are *yes*, and if *no process fails*. Coan and Welch [8] have considered, in the context of a partially synchronous system[9], a *Non-Triviality* condition which requires processes to decide *commit* if all votes are *yes*, *no process fails*, and messages arrive within some known fixed time bound. Besides the fact that these conditions were not defined for replicated objects, we believe that they are also too strong in an asynchronous system, since they are based on precise knowledge about failures. Our *AC-Non-Triviality* condition is more general in the sense that it does not make any assumption on failure detection accuracy. Indeed, with the perfect failure detector $\mathcal{P}$ (which never suspects a correct process), our condition is similar to the one defined by Bernstein et al. If we assume a failure detector based on local time-outs, our condition is similar to the one defined by Coan and Welch.

4 The Dynamic-Terminating-Multicast primitive

We define now the generic *Dynamic-Terminating-Multicast* primitive, abbreviated *DTM*. Instances of DTM allow to solve the locking and atomic commitment problems described in the previous section. This supports our claim that DTM is the adequate basic layer for a distributed transactional system.

4.1 Definition

Consider an initiator process p_i, a message m, and a set of processes $Dst(m)$ (which may or may not include the initiator p_i). Process p_i initially multicasts m to $Dst(m)$, requesting from every process p_j in $Dst(m)$ a reply noted $< p_j, reply_j(m) >$. For two processes $p_j, p_k \in Dst(m)$, we can have $reply_j(m) \neq reply_k(m)$. DTM consists in reaching agreement, among the processes in $Dst(m)$, on a *set* of replies to m: this set of replies is noted *Reply(m)*[10].

[7] Yes votes are required from every DM process in order to commit, if a transaction accesses non-replicated objects.

[8] We say that a process p_i is suspected on a cut c if it is suspected by at least one process on c.

[9] Coan and Welch have assumed a model where messages are usually delivered within some known time bound, but sometimes come late, and have proposed a randomized atomic commitment protocol in this model.

[10] DTM has similarities with the *Distributed Bidding* problem defined in [2]: the difference between both problems is due to the

Formally, given an initiator p_i, a message m and a set of processes $Dst(m)$, the *DTM* primitive is a multicast primitive initiated by p_i that satisfies the following conditions, where f is the maximum number of processes in $Dst(m)$ that can fail:

- **DTM-Uniform-Agreement:** If two processes $p_j, p_k \in Dst(m)$ (correct or not) decide, they decide on the same set of replies *Reply(m)*.
- **DTM-Termination:** If the initiator p_i is correct, or if one process $p_j \in Dst(m)$ has delivered m, then eventually every correct process in $Dst(m)$ decides on $Reply(m)$.
- **DTM-Non-Triviality:** $Reply(m)$ must contain at least $|Dst(m)|-f$ replies, and all the replies of processes that have never been suspected.

The *DTM-Uniform-Agreement* condition is a safety condition. The *DTM-Termination* condition is a liveness condition. The *DTM-Non-Triviality* condition excludes trivial cases such as $Reply(m) = \emptyset$, by setting a requirement on the minimal size of the set $Reply(m)$. It requires $Reply(m)$ to contain the replies of at least $|Dst(m)|-f$ processes, and the replies of every process that has never been suspected. *Reply(m)*$= \emptyset$ would typically prevent progress of the application layer.

Example. Consider an initiator p_1, a message m, $Dst(m) = \{p_2, p_3, p_4\}$, $f = 1$, and p_3 that has been suspected by p_2. *Reply(m)* can be the set $\{< p_2, reply_2(m) > , < p_4, reply_4(m) >\}$.

Remark. The name *Dynamic-Terminating-Multicast* is related to the fact that the multicast of m can be considered terminated once an agreement on *Reply(m)* has been reached, even if not every process in $Dst(m)$ has received m. In the above example the multicast of m to $Dst(m)$ is terminated once *Reply(m)* has been decided, even though p_3 may not have received m.

4.2 An algorithm implementing DTM

We show now an algorithm that solves DTM by reducing DTM to a uniform consensus problem. Thus DTM can be solved whenever uniform consensus can be solved. This is a very important result, which means that DTM can be solved in an asynchronous system augmented with the failure detector $\diamond\mathcal{S}$ (see Sect. 2.2) [6].

fact that DTM is defined in an asynchronous model, whereas the Distributed Bidding problem is defined in a synchronous model.

The algorithm implementing DTM can be split into two steps. Consider the Dynamic-Terminating-Multicast of m to $Dst(m)$, initiated by process p_i:

Step 1: Diffusion of m:

- the initiator p_i starts by sending m to all the processes in $Dst(m)$;
- when m is received by p_j, process p_j generates its reply $< p_j,\ reply_j(m) >$, and sends m and $< p_j,\ reply_j(m) >$ to $Dst(m)$.
  (Forwarding m ensures that, despite the crash of the initiator p_i, if one correct process receives m, then every correct process also receives m);
- process p_j then waits to receive replies $< p_k,\ reply_k(m) >$ from at least $|Dst(m)|-f$ processes out of $|Dst(m)|$, and from all the processes that p_j does not suspect;
- once this last condition is satisfied, p_j defines $v_j(m)$ as the set of replies it has received, and goes to step 2.

Step 2: Uniform consensus. At the beginning of step 2, the initial value $v_j(m)$ of p_j is defined. Process p_j then solves the uniform consensus problem defined on $Dst(m)$. Once p_j has decided $v(m)$, the decision becomes the decision $Reply(m)$ of the DTM problem.

Proposition 2. *The above protocol implements the* DTM *primitive.*

PROOF. By definition of step 1, every proposition $v_j(m)$ satisfies the *DTM-Non-Triviality* condition. Because of the *C-Non-Triviality* condition of the uniform consensus problem, the *DTM-Non-Triviality* condition is satisfied. If the initiator p_i is correct, or if one correct process has received m, then every correct process receives m and sends its reply to $Dst(m)$. Since no more than f processes fail, and eventually every process that fails is permanently suspected by every correct process, each correct process $p_j \in Dst(m)$ eventually defines $v_j(m)$ by step 1. Therefore every correct process eventually starts the uniform consensus problem defined by step 2. Because of the *C-Termination* condition of the consensus problem, the *DTM-Termination* condition is assured. Finally, because of the *C-Uniform-Agreement* condition of the uniform consensus problem, the *DTM-Uniform-Agreement* condition is also satisfied. □

Corollary 3. *The* DTM *primitive can be implemented using the failure detector $\diamond\mathcal{S}$, given that $f < |Dst(m)|/2$.*

PROOF. The uniform consensus problem can be solved with the failure detector $\diamond\mathcal{S}$ when less then half of the processes fail [6]. The Corollary follows directly from that result and from Proposition 2. □

4.3 Cost analysis

The algorithm to solve DTM adds one phase (step 1) to a uniform consensus algorithm (step 2). In the best case (no failure and no suspicion) the Chandra-Toueg consensus algorithm requires 1 1/2 phases. Together this makes 2 1/2 phases for the DTM algorithm.

Step 1 requires $n(n+1)$ point to point messages, where $n = |Dst(m)|$ (n messages sent by the initiator to $Dst(m)$, and n messages sent by every process $p_j \in Dst(m)$. If hardware broadcast (or multicast) is available, this reduces to $n+1$ multicast messages. If hardware broadcast or multicast facilities are not available, step 1 can be modified such that the reply $< p_k, reply_k(m) >$ is sent only to one *coordinator* process in $Dst(m)$, rather than to every process in $Dst(m)$. In this case, step 1 requires only $2n+1$ point to point messages. In the best case, the Chandra-Toueg consensus algorithm requires $3(n-1)$ point to point messages, or $n+1$ messages if hardware broadcast or multicast facilities are available. This makes a total of $2(n+1)$ messages when hardware broadcast facilities are available, and $5n-2$ messages otherwise.

Further optimizations are possible, but are not discussed here.

4.4 Generic dimension of the DTM primitive

The *DTM* primitive is *generic* in the sense that it can be customized to solve various problems. As shown in Section 5, *instances* of *DTM* can be used to solve the locking and atomic commitment problems. Given a message m and a set of destination processes $Dst(m)$, an instance of the DTM primitive is obtained by attaching a semantic to the following generic parameters:

Reply message: the $reply_i(m)$ message generated by a process p_i as the result of the reception of m;

Interpretation rule: an interpretation of the set of replies *Reply*(m) (the decision of DTM).

Examples are given in the next section.

5 Locking and atomic commitment with DTM

In this section, we show that the locking and atomic commitment problems can be solved as instances of DTM. We start with the atomic commitment problem, which is simpler than the locking problem.

5.1 Atomic commitment with DTM

The problem has been presented in Section 3.3. The atomic commitment is initiated by a TM process which multicasts the message $m \equiv$ *vote-request* to the DM processes $D_1, D_2, ..D_n$, using the following instance of DTM:

Reply message: the $reply_i(m)$ message generated by a DM process D_i after the reception of $m \equiv$ *vote-request*, is either a *positive* vote $< p_i,\ yes >$ (D_i is ready to commit the transaction), or a *negative* vote $< p_i,\ no >$ (D_i is not ready to commit the transaction).

Interpretation rule: the outcome decision is based on the set of replies *Reply*(m). The decision is *commit* if and only if (1) *Reply*(m) contains the reply of a quorum of processes in $Dst(m) = \{D_1, \ldots, D_n\}$ [11] and (2) all the replies in *Reply*(m) are *yes*. In any other case, the outcome is *abort*.

The *DTM-Uniform-Agreement* condition ensures that all the DM processes that decide, agree on the same outcome decision, i.e. the *AC-Uniform-Agreement* condition is satisfied. The *AC-Validity* condition is satisfied because of the interpretation rule. The *DTM-Termination* condition ensures the *AC-Termination* condition since every correct DM process that has received the *vote-request* message, eventually decides on *Reply(m)*, i.e. decides on the outcome of the transaction. The *DTM-Non-Triviality* condition ensures that *Reply*(m) contains the reply of all non suspected processes. Hence, if there is a quorum of processes that are non suspected and have voted *yes*, then the outcome of the transaction will be *commit*, which satisfies the *AC-Non-Triviality* condition.

Remark. It is worthwhile at that point to compare the atomic commitment protocol based on DTM, with a traditional 3PC (3 Phase Commitment protocol) [1]. If no failure occurs during the protocol, and no suspicion is generated, then the commitment protocol based

[11] Recall that we consider the case of a transaction on *one* single replicated object (see Sect. 3.1). The extension to multiple replicated objects is trivial.

on DTM has strong similarities with a 3PC protocol (same number of phases). In the case where failures occur, the termination protocol of a 3PC becomes extremely complicated [1]. In comparison, there is no termination protocol in DTM: all possible failure cases are handled within the regular protocol! The failure scenarios are thus much simpler to handle. Also, in DTM, the liveness condition *AC-Termination* is ensured by properties of the failure detector $\diamond\mathcal{S}$. There is no such rigorous characterization of liveness in the case of a traditional 3PC protocol completed with a termination protocol (to handle failures).

5.2 Locking with DTM

The locking problem is adequately solved using a *Totally Ordered* multicast primitive, which ensures that any two messages m and m', respectively multicast to $Dst(m)$ and $Dst(m')$, are delivered in the same order by the processes in $Dst(m) \cap Dst(m')$. We call this primitive *TO-multicast*.

Consider again one single replicated object O managed by DM processes $D_1, \ldots, D_n$. Each Transaction Manager T_j that tries to lock O, TO-multicasts *lock-request*(T_j) to the set of DM processes handling the replicas of O. The lock-holder is the TM process T_j whose lock-request is delivered first at a DM process. The total order semantics guarantee that all DM processes will deliver the lock-request messages (issued by the various TM processes) in the same order. Thus, the DM processes agree on the lock-holder, and the *L-Uniform-Agreement* condition is satisfied (Sect. 3.2). The *L-Termination* condition is satisfied if at least one TM process is correct (in which case it will succeed in issuing the lock request). Finally, the *L-Non-Triviality* condition is satisfied since the lock-holder is by definition a TM process that has requested the lock.

Remark. In [7] Cheriton and Skeen argue that a TO-multicast is useless, because the lock mechanisms, required to ensure consistency, obviates the need for TO-multicast. Basically we have shown here that this argument is erroneous since (1) locking of replicated objects is a difficult problem (that has been shown to be as difficult as the consensus problem, see Sect. 3.2), and (2) the TO-multicast is precisely the adequate mechanism to lock replicated objects.

TO-multicast based on DTM

Implementations for TO-multicasts have been described in numerous papers [3, 4, 5, 9, 17, 21]. Liveness is however not ensured for any of these algorithms when assuming only the properties of the failure detector $\diamond\mathcal{S}$. This is not true for the total order broadcast algorithm given in [6]. This algorithm implements however TO-*broadcast*, and not TO-*multicast*.

TO-multicast can be implemented using Skeen's algorithm [3]. Skeen's algorithm is only correct if processes do not crash. Using DTM, Skeen's algorithm can however easily be extended to handle process crashes. We sketch the basic idea of the algorithm. More details can be found in [19]. The basic idea is to attach a unique sequence number $seq(m)$, to every message m, and to deliver the messages in the order defined by their sequence number. Consider that every process p_i manages a Lamport logical clock l_i [16] [12]. Agreement among $Dst(m)$ on a sequence number $seq(m)$ for the message m can by achieved by the following instance of DTM:

Reply message: the $reply_i(m)$ message generated by a process p_i after the reception of m is $< p_i, l_i >$, where l_i is the time-stamp of the $receive_i(m)$ event.

Interpretation rule: the sequence number $seq(m)$ attached to the message m is the maximal logical clock value in $Reply(m)$.

Total order is ensured by delivering the messages according to their sequence number. This requires of course that, when a message m with $seq(m)$ is delivered by some process p_i, p_i will not learn later of the existence of a message m', such that $p_i \in Dst(m')$, and $seq(m') < seq(m)$ (i.e. m' should have been delivered before m). This is discussed in [19].

6 Concluding remarks

The paper has considered two fundamental problems related to the implementation of transactions in an asynchronous system with replicated objects: the locking problem, and the atomic commitment problem. Whereas the atomic commitment problem has been widely studied in the literature, the difficulty of locking replicated objects has been largely underestimated: we have shown that (fault-tolerant) locking of replicated objects is as difficult as the consensus problem.

To address these two problems, the paper has defined a communication primitive called *Dynamic-Terminating-Multicast* or *DTM*. Informally, DTM

[12] The unicity of the logical clocks can be ensured by appending the process identifier as a suffix to the clock value.

multicasts a message m to a set of destination processes $Dst(m)$, and brings processes in $Dst(m)$ to agree on the set of replies generated by the reception of m. We have also shown that DTM can be implemented whenever the (uniform) consensus problem can be solved. Specifically, this means that DTM can be implemented in an asynchronous system augmented with the unreliable failure detectors $\diamond\mathcal{S}$ and $\diamond\mathcal{W}$ introduced by Chandra and Toueg [6].

DTM is a generic primitive, of which instances enable (1) to solve the (fault-tolerant) atomic commitment problem for transactions accessing replicated objects, and (2) to implement a totally ordered multicast primitive which enables to lock replicated objects in a fault-tolerant way. This shows the importance of a totally ordered multicast primitive in a distributed system. In conclusion, DTM can be viewed as a basic building block for systems implementing transactions, and can be seen as the missing link between the abstract consensus problem and more practical problems such as locking and atomic commitment.

References

[1] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley. 1987.

[2] A. Gopal N. Budhiraja and S. Toueg. Early-Stopping Distributed Bidding and Applications. *Proc. 4th Intl. Workshop on Distributed Algorithms (WDAG-4)*, pages 304–320. Springer Verlag, LNCS 486, 1990.

[3] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1), pages 44-76. February 1987.

[4] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3), pages 272-314. August 1991.

[5] J. Chang and N. Maxemchuck. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems*, 2(3), pages 251-273. August 1984.

[6] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. Technical Report 93-1374, Department of Computer Science, Cornell Univ, 1993. A preliminary version appeared in the *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pages 325-340. ACM Press, August 1991.

[7] D. Cheriton and D. Skeen. Understanding the Limitations of Causally and Totally Ordered Communication. In the *Proceedings of the 14th ACM SIGOPS Symposium on Operating Systems Principles.* ACM Press, pages 44-57. December 1993.

[8] B. Coan and J. Welch. Transaction commit in a realistic timing model. *Distributed Computing*, 4(2) pages 87-103. 1990.

[9] D. Dolev, S. Kramer, and D. Malki. Early Delivery Totally Ordered Broadcast in Asynchronous Environments. In Proceedings of the *23rd Annual International Symposium on Fault-Tolerant Computing*, pages 544-553. June 1993.

[10] K. Eswaren, J. Gray, R. Lorie and I. Traiger. The Notion of Consistency and Predicate Locks in a Database System. *Communications of the ACM.* November 1976.

[11] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. Journal of the ACM (32), pages 374-382. 1985.

[12] D. Gifford. Weighted Voting for Replicated Data. *Proceedings of the 7th ACM SIGOPS Symposium on Operating Systems Principles*, pages 150-159. 1979.

[13] R. Guerraoui. Revisiting the relationship between non blocking atomic commitment and consensus. Technical Report 95/124, Ecole Polytechnique Fédérale de Lausanne, Département d'Informatique, April 1995.

[14] J. Gray. Notes on Database Operating Systems. In *Operating Systems: An Advanced Course.* Lecture Notes in Computer Science (60), Springer Verlag. 1978.

[15] V. Hadzilacos. On the relationship between the atomic commitment and consensus problems. In *Fault-Tolerant Distributed Computing.* B. Simons and A. Spector, editors, Lecture Notes in Computer Science (448), Springer Verlag. 1990.

[16] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7) pages 558-565. 1978.

[17] S. Luan and V. Gligor. A fault-tolerant protocol for atomic broadcast. *IEEE Transactions on Parallel and Distributed Systems*, 1(3) pages 271-285. July 1990.

[18] L. Sabel and K. Marzullo. Election Vs. Consensus in Asynchronous Systems. Technical Report TR95-1488, Cornell Univ., 1995.

[19] A. Schiper and R. Guerraoui. Total order broadcast vs total order multicast: the role of causal order and membership. In preparation.

[20] D. Skeen. NonBlocking Commit Protocols. In Proceedings of the *ACM SIGMOD International Conference on Management of Data*, pages 133-142. ACM Press, 1981.

[21] P. Verissimo, L. Rodrigues, and M. Baptista. AMp: A Highly Parallel Atomic Multicast Protocol. In Proceedings of the *ACM SIGCOMM'89*, pages 83-93, Austin, Texas. September 1989.