Transaction model vs Virtual Synchrony model: bridging the gap ⋆

Rachid Guerraoui André Schiper

Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland

Abstract. Two important models for building fault-tolerant applications have been independently proposed in the literature, the *transaction* model (developed within the context of database applications) and the *virtual synchrony* model (proposed initially by the Isis system to handle replication). For a better understanding of the relations between these models, we address the following question: are the basic mechanisms needed to implement both models exactly the same? The paper answers this question by defining the Dynamic-Terminating-Multicast problem, and showing that it can be seen as a generic problem, closely related to the implementation of both the transaction model and the virtual synchrony model. It is thus possible to build a system offering, in an integrated way, both the transaction model and the virtual synchrony model. Such a system could offer powerful primitives that are currently cumbersome or impossible to express in the existing systems.

1 Introduction

One of the main goals of any model for fault-tolerant distributed systems is to provide adequate consistency, while allowing to achieve a high degree of availability and concurrency. Two important consistency models have been proposed independently in the literature: the *transaction* model, developed in the context of database applications, and the *virtual synchrony* model, developed initially within the context of the Isis system to handle replication [4].

The transaction model considers processes that communicate by invoking operations on shared data. A transaction typically groups invocations on data, and ensures *concurrency atomicity* and *failure atomicity* [12, 2]. Concurrency atomicity (also called *isolation* or *serializability*) means (1) that the intermediate effects of a transaction on shared data are invisible to other transactions, and (2) that the transactions appear to be executed in some sequential order.

⋆ Paper appears in Proc. International Workshop "Theory and Practice in Distributed Systems", Schloss Dagstuhl, Springer Verlag, LNCS 938, 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.

Failure atomicity (also called *all-or-nothing* property) means that either a transaction completes successfully, or aborts and has no effect on the shared data. It is obvious that these correctness criteria are totally independent of any particular implementation. Nevertheless, any implementation of transactions relies on some well established basic mechanisms, such as locks, time-stamps, two phase commitment protocols, three phase commitment protocols, etc. [2]. We are here specifically interested in the mechanisms required to implement the transaction model, and not by the consistency criteria defined by the transaction model.

The virtual synchrony model considers processes that communicate by message passing, and defines a model in which messages are either totally ordered, or respect causal ordering [3]. Moreover, the messages considered in the virtual synchrony model are not only the multicast generated by the application, but also the messages generated by the system itself, in order to signal the crash of processes, or to signal processes joining the system. Similarly to the transaction model, the implementation of the virtual synchrony model relies on some basic mechanisms, and we are interested here in these basic mechanisms, and not in the consistency criteria defined by the virtual synchrony model.

To summarize, the purpose of this paper is not to compare the consistency criterion defined by the transaction model to the consistency criterion defined by the virtual synchrony model. The purpose of the paper is rather to understand and compare the basic mechanisms needed to implement both models. In all generality, there are three possible ways to relate both classes of mechanisms:

1. the virtual synchrony model relies on the most basic mechanisms, meaning that all the mechanisms needed to implement the transaction model can be built on top of the mechanisms underlying the virtual synchrony model;
2. the transaction model relies on the most basic mechanisms, meaning that all the mechanisms needed to implement the virtual synchrony model can be built on top of the mechanisms underlying the transaction model;
3. neither of the two previous statements are true, meaning that there exists some basic mechanism allowing to implement both the mechanisms required by the transaction model, and the mechanisms required by the virtual synchrony model.

If the first statement is true, transactions should always be built on top of the virtual synchrony model. This is obviously not the case. Moreover, if the first statement is true, transactions should be easily and efficiently implemented within a system like Isis. This is also not the case[2]. If the second statement is true, a system like Isis should be easily and efficiently implemented on top of transactions. This is also obviously not the case: a system like Isis is in many aspects more flexible than a system implementing transactions. This might suggest that the third statement is correct, which is precisely what we intend to show in the paper.

[2] A uniform reliable multicast can be used to implement the atomic commitment of a transaction [11]. The solution is however not efficient as it requires to create, at commitment time, a group including all the data managers accessed by the transaction.

To show the link between the mechanisms underlying the transaction model and the virtual synchrony model, the paper defines the *Dynamic-Terminating-Multicast* problem (noted DT-Multicast), and shows that DT-Multicast can be seen as a generic problem. Solving specific instances of the DT-Multicast problem allows to implement the various mechanisms required both by the transaction model and by the virtual synchrony model. The rest of the paper is structured as follows. Section 2 defines the DT-Multicast problem. Section 3 shows the link between DT-Multicast and transactions. Section 4 shows the link between DT-Multicast and virtual synchrony. Finally, Section 5 concludes with some final remarks.

2 The Dynamic-Terminating-Multicast problem

2.1 System model

The distributed system is composed of a finite set $\mathsf{S} = \{p_1, \ldots, p_n\}$ of processes completely connected through a set of channels. Communication is by message passing, asynchronous (there is no bound on the transmission delays), and reliable [3]. Processes fail by crashing (we don't consider Byzantine failures). A *Failure Suspector* module FS_i is attached to every process p_i [6] [4], in order to overcome the Fischer-Lynch-Patterson impossibility result [7]. A process $p_i \in \mathsf{S}$ may (1) send a message to another process, (2) receive a message sent by another process, (3) perform some local computation, or (4) crash, which is modeled by the local event $crash_i$. The process history of $p_i \in \mathsf{S}$ is a sequence of events $h_i = e_i^0 \cdot e_i^1 \cdots e_i^k$. Histories of correct processes are infinite. If not infinite, the process history of p_i is terminated with event $crash_i$.

2.2 Problem definition

We introduce the DT-Multicast problem that will establish the link between the virtual synchrony model and the transaction model. As atomic commitment and virtual synchrony are related to consensus, it is not surprising that the DT-Multicast problem is also related to consensus. We believe however that DT-Multicast is the missing link between (1) the consensus problem, and (2) the various problems related to the implementation of the transaction model

[3] A reliable channel ensures that a message sent by p_i to p_j is eventually received by p_j if p_i and p_j are correct (i.e. do not crash). This does not exclude link failures, if we require that any link failure is eventually repaired. A reliable channel can be implemented by retransmitting lost or corrupt messages.

[4] A failure suspector is a set of modules, one module attached to every process in the system. Each failure detector module maintains a list of processes that it currently suspects to have crashed. Chandra and Toueg [6] characterize the failure suspectors by two properties: *completeness* and *accuracy*. Informally, *completeness* requires that the failure suspector eventually suspects every crashed process, while *accuracy* restricts the mistakes that a failure suspector can make.

and to the implementation of the virtual synchrony model. In this respect the DT-Multicast problem is not only of theoretical interest, but has also a practical impact on the architecture of systems. We come back to this issue later in the paper.

Consider a message m sent by some process in S to all the processes of a set $\mathsf{Dst}(m)$. Assume that, for every process $p_i \in \mathsf{Dst}(m)$, when p_i receives m, it generates a reply noted $< p_j, reply_j(m) >$. The DT-Multicast problem consists in reaching agreement, among $\mathsf{Dst}(m)$, on the set of replies generated by the reception of m. The agreement is thus on a *set* of replies, and we note $Reply_i(m)$ the set of replies decided by p_i. We can formally define the DT-Multicast problem by the following three conditions:

DT1. Uniform Agreement. If two processes in $\mathsf{Dst}(m)$, correct or not, decide on a set of replies, they decide on the same set. Thus if p_i decides on $\mathsf{Reply}_i(m)$ and p_j decides on $\mathsf{Reply}_j(m)$, we have $\mathsf{Reply}_i(m) = \mathsf{Reply}_j(m)$. Because of the agreement condition, we can without ambiguity use the notation $\mathsf{Reply}(m)$ instead of $\mathsf{Reply}_i(m)$.

DT2. Termination. If $p_i \in \mathsf{Dst}(m)$ has received m and is correct, then p_i will eventually decide on $\mathsf{Reply}(m)$.

DT3. Non-Triviality. The Non-Triviality condition sets a requirement on the minimal size of the set $\mathsf{Reply}(m)$. The condition should for example exclude the trivial solution $\mathsf{Reply}(m) = \emptyset$. However, requiring $\mathsf{Reply}(m) \neq \emptyset$ is not sufficient. We do not specify here a general Non-Triviality condition, because this condition is specifically related to an instance of the DT-Multicast problem: we come back to this condition in the next Sections.

The Agreement condition (DT1) is a safety condition. The Termination condition (DT2) is a liveness condition. Finally, to understand why the problem is called *Dynamic-Terminating-Multicast*, notice that, once agreement on $\mathsf{Reply}(m)$ has been reached, the transport protocol can stop retransmitting m to processes in $\mathsf{Dst}(m)$ that might not have received m. The multicast of m is at that point *terminated* for the transport protocol.

2.3 DT-Multicast vs consensus

In the consensus problem, every process $p_i \in \mathsf{S}$ starts with an initial value v_i, and the processes have to decide on a common value v. Formally, the consensus problem is defined by the following three conditions [6]:

C1. Agreement. If two processes decide on a value v, they take the same decision [5].

[5] In an asynchronous system, the Agreement condition is indistinguishable from the condition called *Uniform Agreement*: if two processes (correct or not) decide, they decide on the same value.

C2. Termination. Each correct process eventually takes a decision.

C3. Validity. If a process decides on v, then v is the initial value of some process $p_i \in \mathsf{S}$.

The Agreement condition (C1) is a safety condition, and is in a sense similar to the Agreement condition DT1 of the DT-Multicast problem. The Termination condition (C2) is a liveness condition, and is similar to the Termination condition DT2 of the DT-Multicast problem. The Validity condition (C3) is a Non-Triviality condition. Without C3, the processes could, for example, trivially always decide on the value 0. C3 is however not similar to DT3. The specific DT3 Non-Triviality condition, that has to be defined for every instance of DT-Multicast, is the missing link between (1) the consensus problem, and (2) the mechanisms required to implement the transaction model and the virtual synchrony model.

It can be shown that the DT-Multicast problem, with the Non-Triviality condition $|\mathsf{Reply}(m)| > |\mathsf{Dst}(m)|/2$, is reducible to consensus, i.e. the DT-Multicast problem is solvable whenever the consensus problem is solvable [6]. The purpose of this paper is however not to establish this result. It is rather to stress on the genericity of the DT-Multicast problem, in order to convince the reader that DT-Multicast should be considered as a basic building block of any fault-tolerant system architecture.

2.4 Generic dimension of the DT-Multicast problem

An instance of the generic DT-Multicast problem is obtained by attaching a semantic to the parameters of the DT-Multicast problem. More specifically, an instance of the DT-Multicast problem is defined by the following parameters:

Message: the semantics of the message m.

Destination set: the $\mathsf{Dst}(m)$ set of processes to whom m has to be multicast.

Reply message: the $reply_j(m)$ message, generated by a process p_j, resulting from the reception of the message m.

Non-Triviality condition: a specific Non-Triviality condition.

Interpretation rule: an interpretation of the set of replies $\mathsf{Reply}(m)$, on which the processes in $\mathsf{Dst}(m)$ agree.

Examples of instantiations of the DT-Multicast problem are given in the next Sections.

3 The DT-Multicast problem and transactions

3.1 Transaction model

There are numerous mechanisms that help in implementing transactions. Even if we do not establish the result formally, we think that many mechanisms related to the serializability and atomicity properties of transactions can be implemented as instances of the DT-Multicast problem. We consider here one single example: the classical atomic commitment problem.

Consider the following model [2]. A transaction originates at a process called the Transaction Manager (noted TM), which accesses data by interacting with various processes called Data Managers (noted DM) located on different sites. The TM initially performs a *begin-transaction* operation, then various *write* and *read* operations (by translating writes and reads into messages sent to the DMs), and finally an *end-transaction* operation. The *end-transaction* operation can have two outcomes: *commit* or *abort*. The write operations performed by the DMs become permanent if and only if the outcome is *commit*. To ensure agreement among all the DMs on the outcome of the transaction (the so-called *failure atomicity* property), the TM must solve a *commitment problem* as part of the *end-transaction* operation. We show that the atomic commitment problem can be seen as an instance of the DT-Multicast problem.

3.2 Atomic commitment as an instance of the generic DT-Multicast problem

Consider a transaction A that has accessed the set of data managers noted $\mathsf{DM}(A)$. The atomic commitment requires a voting phase, which allows to know the status of every DM accessed by the transaction. The vote of a data manager can be either YES or NO:

- NO means that the data manager is for some reason not ready to commit the transaction. This can be the case if the data manager has detected concurrency control conflicts, or if the data manager was not able to store the updated data on permanent storage in order to recover from a possible subsequent crash;
- YES means that the data manager is ready to commit the transaction, which means that it has detected no concurrency control conflicts, and was able to store the updated data on permanent storage, in such a way that the data survives to a possible subsequent crash.

Based on the YES / NO vote, the atomic commitment problem is defined by the following conditions [1]:

AC1. All DMs that decide reach the same decision;

AC2. If any DM decides *commit*, then all DMs must have voted YES;

AC3. If all DMs vote YES and no failure occurs, then all DMs decide *commit*;

AC4. Each DM decides at most once.

AC5. Every correct DM that executes the atomic commitment protocol eventually decides.

Conditions AC1, AC2 and AC4 are safety conditions. Condition AC3 is a non-triviality condition, but can also be seen as a liveness condition from the application point of view: progress from the application point of view is ensured only by the *commit* outcome of a transaction. In this sense, AC3 can be seen as a liveness condition. Condition AC5 is also a liveness condition.

Given a transaction A, the atomic commitment problem defined by the conditions AC1-AC5 can be seen as the following instance of the DT-Multicast problem. We start by stating the obvious part:

Message: the message m is the *vote-request* message.

Destination set: the *vote-request* message is multicast to the set of data managers accessed by the transaction A, noted $\mathsf{DM}(A)$. In other words, $\mathsf{Dst}(m) \stackrel{\text{def}}{=} \mathsf{DM}(A)$.

Reply message: the reply of a data manager DM_i to the reception of the *vote-request* message is either $< \mathrm{DM}_i$, YES $>$ or $< \mathrm{DM}_i$, NO $>$, according to the semantics given above.

Interpretation rule: the outcome of the transaction is based on the set of replies $\mathsf{Reply}(m)$. The outcome is *commit* if and only if (1) $\mathsf{Reply}(m)$ contains the reply of every data manager in $\mathsf{DM}(A)$, and (2) all the replies in $\mathsf{Reply}(m)$ are YES. In any other case, the outcome of the transaction is *abort*.

The less obvious part of the instantiation of the DT-Multicast problem is the Non-Triviality condition DT3. The Non-Triviality condition has to ensure the AC3 condition. According to the Interpretation rule given above, the *commit* outcome is possible only if $\mathsf{Reply}(m)$ contains the reply of every data manager in $\mathsf{DM}(A)$. Thus the AC3 condition leads to the following Non-Triviality condition for the DT-Multicast problem:

DT3-AC. Non-Triviality for Atomic Commitment. The reply of a data manager DM_i is not in $\mathsf{Reply}(m)$, only if DM_i has crashed.

It is easy to show that this instance of the DT-Multicast problem, including the above Non-Triviality condition, satisfies the conditions AC1-AC5:

- AC1 is satisfied, as the decision is based on the $\mathsf{Reply}(m)$ set of replies, and the data managers agree on $\mathsf{Reply}(m)$ (Uniform Agreement condition DT1);
- AC2 is satisfied because of the Interpretation rule above;
- AC3 is satisfied because of the Non-Triviality condition DT3-AC, and because of the Interpretation rule;
- AC4 is satisfied because the decision is based on the set $\mathsf{Reply}(m)$;
- AC5 is satisfied because of the Termination condition DT2.

It is worthwhile noting here, that the only way to guarantee the Non-Triviality condition DT3-AC is to have a perfect failure suspector $\mathcal{P}$, defined in [6] by the following properties:

- eventually every process that crashes is permanently suspected by every correct process;
- correct processes are never suspected.

It has been shown that, if less than half of the processes crash, consensus can be solved with the weak failure suspector $\diamond\mathcal{W}$ [6], which is obviously weaker than $\mathcal{P}$ [6]. Thus the Atomic Commitment defined by the conditions AC1-AC5, i.e. by the DT-Multicast problem with the DT3-AC Non-Triviality condition, is formally harder than consensus!

4 The DT-Multicast problem and virtual synchrony

4.1 Virtual synchrony

Virtual synchrony (or VS), first introduced by the Isis system, is based on views, defined for every group G of processes. Given a group G, a view $\mathsf{View}_k(G)$ is a set of correct processes of G, as perceived by a membership service. The VS model is defined by the following two properties for every group G [3]: (1) processes in G agree on the sequence of views $\mathsf{View}_1(G), \mathsf{View}_2(G), \ldots$ (i.e. views are totally ordered), and (2) multicasts to G are totally ordered with respect to view changes: if a process p_i member of a group G delivers a multicast m while in view $\mathsf{View}_k(G)$, then no member of G delivers m in another view. Moreover, the *ABCAST* primitive, defined by the Isis system, orders totally the messages multicast within each view.

In order to keep the presentation simple, we consider in the sequel the following simplified context: we consider a single group G, and assume that every multicast is an *ABCAST* to G. Further, as usual in the VS model, ABCASTs to G are issued by members of G. We start by showing that the ABCAST primitive can be implemented as an instance of the DT-Multicast problem. We show then the link between the VS model and the DT-Multicast problem.

4.2 ABCAST as an instance of the DT-Multicast problem

Total order can be seen as the problem of attaching to every message m a sequence number $sn(m)$, and delivering the messages in the order defined by their sequence number. One could imagine using Lamport's logical clocks as the sequence number, which would lead to Lamport's total order algorithm [8]. This is unfortunately not adequate in an environment where processes can crash. Consider a group G, and a process $p_i \in G$ multicasting a message m to G. According to Lamport's algorithm, process p_i would have to wait for a time-stamped acknowledgment from every process in G before being able to deliver m. This leads to blocking in the case of a single process failure (a crashed process will not send the time-stamped acknowledgment message required by Lamport's algorithm). To avoid blocking, Lamport's protocol has to be combined with a membership protocol that changes the composition of G (this is, for example, the solution described in [9]). Views delivered by the membership protocol have however to be totally ordered [10]. In other words, the initial total order protocol relies on a second total order protocol, leading to a circularity problem.

The total order protocol described here does not suffer from this circularity problem. It is basically the ABCAST protocol of the Isis system described in [5]. The idea is again to attach a unique sequence number to every message, and to deliver the messages according to their sequence number. However, the sequence number is not defined a priori as in Lamport's total order protocol; it is defined a posteriori, and all processes in G agree on the sequence number attached to every message multicast to G.

Consider a group G and the current view $\mathsf{View}_k(G)$. Every process p_i in G manages a Lamport logical clock l_i [6]. Agreement on the sequence number of messages can be ensured by the following instantiation of the DT-Multicast problem. Consider the message m_{ABCAST} that is multicast by process p_i to G, i.e. to the current view $\mathsf{View}_k(G)$:

Message: the message m of the DT-Multicast problem is the message m_{ABCAST}.

Destination set: the message m_{ABCAST} is multicast to $\mathsf{View}_k(G)$, i.e. $\mathsf{Dst}(m) \stackrel{\text{def}}{=} \mathsf{View}_k(G)$.

Reply message: when p_j receives a multicast m, it sends its updated logical clock value as its reply: $reply_j(m) \equiv l_j$ [7].

Non-Triviality condition: As Non-Triviality condition, we require the set $\mathsf{Reply}(m)$ to contain a majority of replies from $\mathsf{Dst}(m)$:
$|\mathsf{Reply}(m)| > |\mathsf{Dst}(m)|/2$.

Interpretation rule: The sequence number $sn(m)$ attached to the message m is the maximum of the logical clock values obtained from the set of replies $\mathsf{Reply}(m)$.

Because of the Uniform Agreement on the set $\mathsf{Reply}(m)$ (condition DT1), the above protocol obviously attaches a unique sequence number $sn(m)$ to every message m. Also, given two different messages m and m', we have $sn(m) \neq sn(m')$. This is ensured by the process ID suffix attached to the sequence number, and by the incrementation rule of the Lamport clock: two messages m and m' can have the same sequence number only if the sequence number has the same process ID suffix, say the suffix of some process p_j. However, in this case, the clock value returned as a reply by p_j cannot be the same for two different messages m and m' received by p_j. Thus total order can be ensured by delivering messages according to their sequence number. This requires of course that, when message m with sequence number $sn(m)$ is delivered by p_i, there will be no later message

[6] In order to ensure the uniqueness of the logical clocks, the process ID is appended as a suffix to the clock value.

[7] If the process p_i multicasting m_{ABCAST} is in $\mathsf{Dst}(m)$, one can consider that p_i piggybacks its reply $reply_i(m_{ABCAST})$ on the message m_{ABCAST} itself.

m' with sequence number $sn(m')$ such that $sn(m') < sn(m)$. We do not discuss this point here.

4.3 VS as an instance of the DT-Multicast problem: handling failures

We have shown in the previous Section that total order can be implemented as an instance of the DT-Multicast problem. The implementation can be used to order both application messages and system messages (e.g. view change messages). DT-Multicast can thus be used to implement virtual synchrony.

Consider again the current view $\mathsf{View}_k(G)$ of a group G, and a process $p_i \in \mathsf{View}_k(G)$ that issues a totally ordered multicast m_{ABCAST} to $\mathsf{View}_k(G)$. Let us call m_{ABCAST} an *application* message (in order to distinguish it from *system* messages triggering view changes). The virtual synchrony properties (views of G totally ordered, and ordering of application multicasts with respect to view changes) can be ensured together with the total ordering of an m_{ABCAST} message as follows. Consider the algorithm of the previous Section, that allows to totally order multicasts. Consider then a specific application message m_{ABCAST} that is about to be delivered, according to its sequence number. Let $\mathsf{P\text{-}Reply}(m_{ABCAST})$ be the set of processes whose reply is in $\mathsf{Reply}(m_{ABCAST})$:

$$\mathsf{P\text{-}Reply}(m_{ABCAST}) \stackrel{\text{def}}{=} \{p_j \mid < p_j, reply_j(m_{ABCAST}) > \in \mathsf{Reply}(m_{ABCAST})\}$$

The set $\mathsf{P\text{-}Reply}(m_{ABCAST})$ allows to change the view of group G according to the following rules:

– when m_{ABCAST} is delivered, and $\mathsf{P\text{-}Reply}(m_{ABCAST})$ is not equal to $\mathsf{View}_k(G)$, then a new view $\mathsf{View}_{k+1}(G)$ is defined and delivered. This new view is delivered immediately after the delivery of m_{ABCAST}, i.e. before the delivery of the next application message. The new view $\mathsf{View}_{k+1}(G)$ is $\mathsf{P\text{-}Reply}(m_{ABCAST})$ (or more precisely, $\mathsf{View}_k(G) \cap \mathsf{P\text{-}Reply}(m_{ABCAST})$).

This satisfies the virtual synchrony properties for the following reasons. Because of the solution of the DT-Multicast problem, processes in $\mathsf{View}_k(G)$ have reached agreement on the set of replies $\mathsf{Reply}(m_{ABCAST})$. Thus they have also reached agreement on $\mathsf{P\text{-}Reply}(m_{ABCAST})$. In other words, the view of G is changed from $\mathsf{View}_k(G)$ to $\mathsf{View}_{k+1}(G)$ while ensuring the following two *virtual synchrony* properties:

– every process in $\mathsf{View}_k(G)$ delivers m_{ABCAST} before the delivery of the new view $\mathsf{View}_{k+1}(G)$;
– the processes in $\mathsf{View}_k(G)$ all agree on the next view $\mathsf{View}_{k+1}(G)$.

4.4 VS as an instance of the DT-Multicast problem: handling joins

The protocol of the previous Section allows to handle process failures, which result in removing processes from the current view. Adding processes to a view is not more difficult. Recall that, in order to keep the presentation simple, we assume that every multicast to G is an ABCAST.

Consider the current view $\mathsf{View}_k(G)$ and a process p_k wanting to join G. In order to join G, process p_k has to send a *join* message to some process, say p_i, member of the current view $\mathsf{View}_k(G)$. Process p_i then creates a message $m \equiv < join, p_k >$ that it ABCASTs to $\mathsf{View}_k(G)$. When this message is delivered by some process p_j in $\mathsf{View}_k(G)$, p_j does the following:

Next view: p_j defines the next view $\mathsf{View}_k(G)$ as $(\mathsf{View}_k(G) \cup \{p_k\}) \cap$ $\mathsf{P\text{-}Reply}(m)$.

State transfer: p_j sends its current state together with the new view $\mathsf{View}_{k+1}(G)$ to p_k (note that not every process in $\mathsf{View}_k(G)$ needs to send its state and the new view to p_k).

5 Discussion

The paper has defined the DT-Multicast problem that consists in multicasting a message m to a set of destination processes $\mathsf{Dst}(m)$, and then reaching agreement among $\mathsf{Dst}(m)$ on the set of replies generated by the reception of m. The DT-Multicast problem can be seen as a generic problem parameterized by (1) a message m, (2) a set of destination processes $\mathsf{Dst}(m)$, (3) the reply message generated by every process $p_i \in \mathsf{Dst}(m)$ when receiving m, (4) a Non-Triviality condition setting requirements on the size of the $\mathsf{Reply}(m)$ set, and (5) an interpretation of the set $\mathsf{Reply}(m)$.

Specifically, the paper has shown that the Atomic Commitment problem can be seen as an instance of the DT-Multicast problem. The paper has also shown that an ABCAST primitive (multicast with total order delivery) can also be implemented as an instance of the DT-Multicast problem. The solution does not require a membership service, avoiding thus the traditional circularity problem of ABCAST primitives built on top of a membership service (that totally orders view changes). Moreover, the $\mathsf{Reply}(m)$ set obtained as a result of the DT-Multicast problem, allows implementing virtual synchrony.

The genericity of the DT-Multicast problem has a very practical consequence: it should be possible to build a system whose basic layer, given a specific Non-Triviality condition, solves the DT-Multicast problem. On top of this DT-Multicast layer, it is then possible to implement both transactions and virtual synchrony, offering both models in an integrated way. Such a system based on DT-Multicast could hide replication within groups similarly to Isis, but would completely decouple groups from the consistency issues (strongly coupled in a system like Isis). Without this flexibility, it is necessary to create a group every time an consistency related operation has to be performed, e.g. atomic commitment of a distributed transaction (that requires a *uniform* reliable multicast

[11]), ABCAST to a set of destination processes, etc. Creating a group just to issue a single multicast is an expensive operation. Also, a system based on DT-Multicast could offer very powerful primitives (e.g. atomic operations on multiple groups) that are currently cumbersome or impossible to express within the virtual synchrony model. A system based on DT-Multicast could also offer a modular approach to transactions by defining adequate basic primitives to handle replication (e.g. static vs dynamic voting).

Acknowledgements

We would like to thank Ken Birman, Benoît Garbinato, Friedemann Mattern, Karim Mazouni and Michel Raynal for their suggestions that helped improving the paper.

References

1. O. Babaoglu and S. Toueg. Non-Blocking Atomic Commitment. In Sape Mullender, editor, *Distributed Systems*. ACM Press, 1993.
2. A.J. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Distributed Database Systems*. Addison-Welsey, 1987.
3. K. Birman. Virtual Synchrony Model. In K. Birman and R. van Renesse, editors, *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.
4. K. Birman. The Process Group Approach to Reliable Distributed Computing. *Comm. ACM*, 36(12):37–53, December 93.
5. K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, February 1987.
6. T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. Technical Report 93-1374, Department of Computer Science, Cornell University, August 1993. A preliminary version appeared in the *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 325–340. ACM Press, August 1991.
7. M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32:374–382, April 1985.
8. L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. ACM*, 21(7):558–565, July 78.
9. S. Shrivastava P. Ezhilchelvan, R. Macedo. Newtop: A Fault-Tolerant Group Communication Protocol. Technical report, Univ of Newcastle, 1994.
10. A. M. Ricciardi and K. P. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Proc. of the 10th ACM Symposium on Principles of Distributed Computing*, pages 341–352, August 1991.
11. A. Schiper and A. Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environment. In *IEEE 13th Intl. Conf. Distributed Computing Systems*, pages 561–568, May 93.
12. A.Z. Spector and P.M. Schwarz. Transactions: A construct for reliable distributed computing. *ACM Operating Systems Review*, 14(2):18–35, April 1983.