Using Virtual Synchrony to Develop Efficient Fault Tolerant Distributed Shared Memories*

Roy Friedman

Department of Computer Science
Cornell University
Ithaca, NY 14853.

March 31, 1995

Abstract

This paper shows how to define consistency conditions for distributed shared memories in virtually synchronous environments. Such definitions allow to develop fault tolerant implementations of distributed shared memories, in which during normal execution, operations can be performed very efficiently, and only those operations which take place during a configuration change must be delayed. Three well known consistency conditions, namely, linearizability, sequential consistency, and causal memory, are redefined for virtually synchronous environments. It is then shown how to provide efficient fault tolerant implementations for these definitions.

*This work was supported by ARPA/ONR grant N00014-92-J-1866

1 Introduction

Distributed shared memory is an important communication paradigm for making massively parallel computers usable, and for the successful treatment of networks of workstations as a parallel machine. Distributed shared memory is a convenient model to work with, it is a natural extension of parallel programming on uniprocessors to distributed environments, and it is easier to transform a serial program into a distributed shared memory based program, than to a message passing based program.

For these reasons and others, much research has been done on how to define and implement distributed shared memory, both from the theoretical point of view [2, 3, 6, 10, 11, 13, 22, 26, 27, 28] and from the practical point of view [14, 17, 18, 19, 21, 31, 32, 33]. However, most of this work ignores the possibility of failures and does not allow processors to join an ongoing computation, in order to speed it up. On the other hand, it is clear that if distributed shared memory is to be implemented in massively parallel computers, or over a network of workstations, the probability that at least one process will crash during a computation is too high to assume that whenever something like that happens, we can simply restart the entire system. Similarly, if a processor that was down for a while is being re-booted, we may want to add it to the set of processors that are participating in a computation, so it can help the other processors.

One way to cope with processor failures is to use a quorum for every operation [9]. However, this method has two main disadvantages: (1) the execution time and the message overhead of each operation pays a penalty in order to cope with failures (i.e., the additional time and messages required to establish the quorum), and (2) the total number of processes that can take part in a computation must not change during the computation and must be known in advance. These two drawbacks make quorum algorithms infeasible for real systems. Moreover, in modern computers, the actual frequency in which processors crash and/or being re-booted is not high enough to justify these limitations. This makes one wonder if there is a way to build a fault tolerant distributed shared memory without having to pay a penalty for it in every operation, and to allow an arbitrary number of processes to participate in different stages of the same computation.

In this paper, we claim that the answer to both of these questions is yes, if the distributed shared memory is implemented on top of a virtually synchronous communication layer. That is, by using virtual synchrony, we can implement fault tolerant distributed shared memories in which most operations are executed very efficiently, and the number of processes that participate in a single computation may change during the course of computation.

Virtual synchrony was originally defined by Ken Birman, for the ISIS project [15, 16], and is supported by several other projects, such as Horus [34], Totem [8], and Transis [7]. In virtually synchronous systems, each process is presented with a data structure called *view*, that includes the list of all participants in the computation. Since some processes may fail, while other processes may want to join the computation, these views may change with time. However, it is guaranteed that between two consecutive views, all the members that appear in both views will receive exactly the same messages. Hence, during "normal" execution, operations can be executed very efficiently, and only operations that are invoked during view changes need to pay for the fact that the configuration is changing.

Unfortunately, conventional definitions of consistency conditions cannot be applied directly to virtually synchronous environments. In order to cope with the Fischer, Lynch, and Patterson lower bound [23], virtually synchronous environments sometimes arbitrarily remove live processes from the views of the rest of the processes. However, a removed live process may have invoked an operation immediately before it is removed from the view. Since this process has not really crashed, it is unclear how to treat these operations. In particular, it is possible that part of these operations have been executed by another process that was also removed from the view, but none of the processes that remain in the view were aware of these operations.

In this paper we show how to formally define consistency conditions in virtually synchronous environments, using a framework which is a combination of the framework developed in [13] for defining consistency conditions, and the framework developed in [25] for defining virtual synchrony. We demonstrate our definition method for three well known consistency conditions, namely *linearizability* [28], *sequential consistency* [30], and *causal memory* [5]. However, our definition method is general, and can be applied to other consistency condition as well. Following this, for each of these consistency conditions, we present an efficient implementation which is based on an underlying virtually synchronous communication system, and prove its correctness. We believe that these proofs can shed a light on how to make rigorous claims about application that are built on top of a virtually consistent underlying systems.

The definition of virtual synchrony that we use, is the one supported by Horus. This definition includes the property that each operation must be delivered within the view in which it was sent. This property is not supported by other virtually synchronous communication systems, including ISIS, Totem, and Transis. However, the correctness of our implementations depends on this property.

The rest of this paper is organized as follows: Section 2 presents the formal model and the major assumptions. In Section 3 we explain how to use our model to define consistency conditions, and give the definitions of linearizability, sequential consistency, and causal memory. The implementation of these conditions are presented in Section 4. We conclude with a discussion in Section 5.

2 The Model

We assume a finite, but possibly unbounded, set of *application programs*, running concurrently and communicating via virtual shared memory, which consists of a collection of objects. We assume that each application program runs on a different node, which consists of a *memory consistency system* (mcs) and a *virtually synchronous communication layer* (vscl). A correctness condition is defined between the application program and the mcs; we call the correctness condition a *consistency condition.* The application program invokes operations on the shared

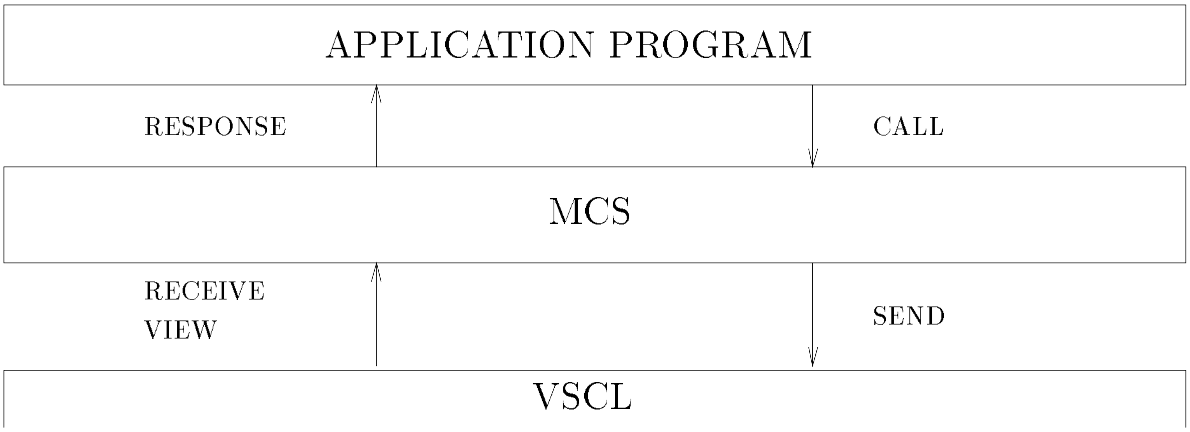

Figure 1: System architecture

memory by issuing *call* events to the mcs. Upon accepting a call event, the mcs is responsible for taking the appropriate actions, including communication with the mcs processes at other nodes, in order to be able to generate a *response* event that obeys the correctness condition. The mcs at each node can communicate with mcs processes at other nodes by issuing *send* events to the vscl, and by accepting *receive* events from it. Moreover, every now and then, the vscl may issue a *view* event to the mcs. We say that call, receive, and view are incoming events, while send and response are outgoing events. (An illustration of the system architecture appears in Figure 1.)

Each of the above events includes a list of parameters:

- A call event include the type of operation that is being invoked, the object on which the operation should be applied, and a list of parameters for this operation. For example, in the case of read/write objects, the type of the operation would be read or write, the object would be the object that is being accessed, and in the case of a write, there would also be a value, which is the value to be written to the object.
- A response event includes similar parameters to call events. For example, in the case of read/write objects, the type of the operation would be read or write, the object would be the object that was accessed, and in the case of a read, there would also be a value, which is the value being returned from the object.
- A view event includes a unique identifier, denoted by *vid*, and a list of processes, denoted by *view*.
- A send event includes a message, denoted by *msg*, a view identifier, denoted by *vid*, and a list of recipients, denoted by *target*. We assume that the contents of every message is unique (this can be achieved by timestamping the messages). Also, the *vid* field in every

message is the same as the *vid* field of the last previous view event that occurred in its invoking process, before it was invoked, or $\perp$, if there is no such view event. We say that a send event *invokes* the message that appears in *msg*. We assume also, that send events can be labeled as **atomic**, **causal**, or **unordered**.

- A receive event includes a message, denoted by *msg*, the name of the process that sent it, denoted by *origin*, and a view identifier *vid*, which is the same as the view identifier field of the send event that invokes *msg*. (If there is no such send event, both *origin* and *vid* are $\perp$.[1])

The mcs process at each node is an automaton with a (possibly infinite) set of states, including an initial state, and a transition function. Each incoming event causes an application of the transition function. The transition function is a function from states and incoming events to states, set of response events, and set of send events. A *step* of a process p is a tuple (s, ev, s', R, M), where s and s' are states, ev is an incoming event, R is a set of response events and M is a set of send events. A *history* of a process p is a sequence of events as they occur in p. In particular, the old state in the first step in a history of a process p is the initial state of p, and the old state in each subsequent step in this history is the new state of the previous step. With each event ev in a history h of a process p, we associate the real time at which ev has occurred in p. However, p does not necessarily know what this real time is.

An *execution* is a collection of histories in which there is a one to one correspondence from the messages received by p_i from p_j to the messages sent by p_j to p_i. (Note that some messages may never be received.) An *infinite execution* is an execution in which some histories are infinite. Given two executions σ and σ', we say that σ' is a *subexecution* of σ if every event in σ' is included in σ.

We assume that process may crash, and then either remain crashed, or become alive again. For simplicity, we assume that *crash* and *live* are incoming events too, and that the first event in each history is a live event. Moreover, between every two crash events in the same history there has to be a live event, and between every two live events in the same history, there has to be a crash event. Also, there can be no other events between a crash event and the following live event of the same history; if there is a crash event ev in a history h which is not followed by a live event, then ev is the last event in h. Finally, we assume that between every two call events in the same history there is either a response event, a crash event, or a view event.

Given a history h and two view events ev and ev' in h, we say that ev and ev' are *consecutive* view events in h if $ev \xrightarrow{h} ev'$ and there does not exists another view event ev'' in h such that $ev \xrightarrow{h} ev'' \xrightarrow{h} ev'$.

In the definition below, and the rest of the paper, we denote by h_i the history that corresponds to process p_i, for every i. An intuitive explanation of the definition below is given in the text that follows the definition.

[1] Later in this paper, we forbid receive events that do not have a corresponding send event. However, since at this point spontaneous receive events are still allowed, we need to make sure that the definition holds for them too.

Definition 2.1 (Virtual Synchrony) *An execution σ is virtually synchronous if the following holds:*

1. *For every history h_i in σ and every view event ev in h_i, p_i is included in ev.*
2. *If ev and ev' are two view events in h_i such that $ev \xrightarrow{h_i} ev'$, then $ev.vid < ev'.vid$.*
3. *For every history h_i in σ, if h_i includes a live event ev, then h_i includes another event ev' such that ev' is either a crash event or a view event and $ev \xrightarrow{h_i} ev'$.*
4. *For every history h_i in σ, if h_i includes a view event followed by a crash event ev', then every history h_j that includes ev, must include another event ev'' such that $ev \xrightarrow{h_j} ev''$ and ev'' is either a crash event, or a view event in which $p_i \notin ev''.view$.*
5. *For every history h_i in σ and every receive event ev in h_i, if there exists a view event $ev'.vid = ev.vid$ and there exists a view event ev'' in h_i such that $ev''.vid > ev'.vid$, then $ev \xrightarrow{h_i} ev''$.*
6. *Let ev be a send event, ev' a view event such that $ev.vid = ev'.vid$, and ev'' be a consecutive view event in a history h_i such that $p_i \in ev.target$ and $ev.pid \in ev'.view \cap ev''.view$, then there exists a receive event ev''' in h_i such that $ev.msg = ev'''.msg$.*
7. *Let ev be a send event, ev' the corresponding receive event ev' and ev'' and ev''' two view events for which there exists a history h_i in which ev'' and ev''' are two consecutive view event, and $ev'' \xrightarrow{h_i} ev' \xrightarrow{h_i} ev'''$, then for every history h_j, such that $p_j \in ev.target \cap ev''.view \cap ev'''.view$, $ev' \in h_j$.*
8. *For every two consecutive view events ev and ev' in a history h_i and every process p_j, if p_j appears in $ev.view \cap ev'.view$, then h_j includes ev.*
9. *For every history h_i and two send events ev and ev' such that for some history h_j, $ev \xrightarrow{h_j} ev'$, $ev.vid = ev'.vid$ and $p_i \in ev.target \cap ev'.target$, if h_i includes the receive event that corresponds to ev', denoted by ev''', then h_i includes the receive event that corresponds to ev, denoted by ev'', and $ev'' \xrightarrow{h_i} ev'''$.*
10. *For every two histories h_i and h_j that include two receive events ev and ev', whose corresponding send events are labeled as* **atomic**, *if $ev \xrightarrow{h_i} ev'$, then $ev \xrightarrow{h_j} ev'$.*
11. *For every history h_i and every two receive events ev and ev', if the corresponding send events ev'' and ev''' are labeled as* **causal**, *and ev'' causally precedes ev''' in σ, then $ev \xrightarrow{h_i} ev'$.*

The first condition in the definition of virtual synchrony requires that every process will be included in all of its local views. The second condition requires that the *vid* field of view events will reflect their relative order within a history. (Note that the *vid* field is used in the definition of virtual synchrony only for bookkeeping.) The third condition requires that if a process is awakened, then either it will crash again, or eventually receive a view event. The fourth condition requires that every process that crashes will eventually be removed from the views of all processes, if it appeared in their views. The fifth condition requires that every message must be delivered within the view in which it was sent. This is a very strong requirement, which is supported by Horus, but is not supported by other systems like ISIS, Totem, and Transis. The sixth condition requires that every message sent by a process that remains in the view of other processes, is delivered by these processes. This requirements guarantees reliable delivery of messages. The seventh condition requires that all the members of two consecutive views will agree on the set of messages delivered between these two views. The eighth condition requires that if process p_j appears in two consecutive views of another process p_i, denote the first one of these views by V, then p_j must have seen view V as well. By this requirement, processes must agree on the views in which they appear. The ninth condition requires that between two consecutive view events, horus simulates a fail stop model among messages sent to the same destinations, and that these messages are delivered in fifo order. The tenth condition requires that messages that are sent with atomic broadcast (**atomic**), are delivered everywhere in the same order. The eleventh condition requires that messages that are sent with causal broadcast (**causal**), are delivered in an order which agrees with the causal order of events in the execution [29].

We call the interval of events in a history h of a process p between a view event and the first following event, which is either a crash event or another view event, a *phase* of p. If h has a view event ev which is not followed by either another view event or by a crash event, then the (possibly infinite) interval of events that starts in ev is a phase. Since a history is a sequence of events, we can talk about the ith phase of a process p.

Each pair of a *call* event and a subsequent matching *response* event forms an *operation.* The call event marks the start of the operation, while the response event marks the end of that operation. An operation op is *invoked* when the application process issues the appropriate call for op; op *terminates* when the mcs process issues the appropriate response for op, when a following view event occurs, or when a following crash event occurs, whichever happens first. An operation is *pending* if it was invoked and did not terminate. Note that due to our assumptions on executions, each node can have at most one pending operation at a time. We use the notation op_i, to denote an operation that is invoked by p_i. If there are several such operations, we may use superscripts to distinguish between them, e.g,. op_i^1, op_i^2, etc.

Given an execution σ and a subexecution σ', we say that σ' is a *continuous subexecution with respect to phases of* σ, if the following holds: If σ' includes an operation op_i that was invoked during some phase k of p_i, then σ' includes every operation by p_i that was invoked during phase k of p_i and appears before op_i in σ.

Each object is assumed to have a *sequential specification* (cf. [28]), defining a set of *oper-*

ations, which are ordered pairs of call and response events, and a set of operation sequences, which are the allowable sequences of operations on that object. For example, in the case of read/write objects, the ordered pair of events [call(Read,x), response(read,x, v)] forms an *operation*, for any x and v, as does [call(write,x, v), response(write,x)].

A sequence τ of operations for a collection of objects is *legal* if, for every object x, the restriction of τ to operations of x is in the sequential specification of x.

An execution σ implies a partial order, $\xrightarrow{\sigma}$, on the operations which appear in σ; $op_i^1 \xrightarrow{\sigma} op_j^2$ if the real time associated with the response event of op_i^1 in σ is earlier than the real time associated with the call event of op_j^2 in σ. Given an execution σ, a sequence of operations τ is a *serialization* of σ if it is a permutation of σ. A serialization τ of σ is a *linearization* if it extends $\xrightarrow{\sigma}$; that is, if $op_i^1 \xrightarrow{\sigma} op_j^2$ then $op_i^1 \xrightarrow{\tau} op_j^2$.

Let σ be an execution. We denote by $\sigma \mid i$ the restriction of σ to operations invoked by p_i; assuming read/write objects, we denote by $\sigma \mid (i + w)$ the restriction of σ to operations invoked by p_i and to writes by any process. Similarly, if τ is a sequence of operations, we denote by $\tau \mid i$ the restriction of τ to operations invoked by p_i; assuming read/write objects, we denote by $\tau \mid (i + w)$ the restriction of τ to operations invoked by p_i and to writes by any process.

3 Defining Consistency Conditions

In this section we show how to define consistency conditions in virtually synchronous environments. Definitions of consistency conditions in virtually synchronous environments are basically the same as in the framework developed in [13]. That is, we make ordering requirements on sets of operations in the execution. Here, however, the requirements on the ordering of operations do not necessarily apply to all operations in the execution. Instead, we need to choose the set of operations that were observed by the healthy processes, and make the ordering requirements only on these operations. This is somewhat similar to what is done in [10], were the ordering requirements are enforced only for the set of operations that have a permanent effect. However, in [10], the reason for not including operations in the set of operations for which an ordering is required is that these operations were a result of a false branch prediction. In this paper, on the other hand, we decide to exclude operations if they were invoked by processes that are removed from the main view, before being noticed by the rest of the processes.

An execution σ is said to be *benign* if in every history of σ there exists at most one view event, the view events of all the histories of σ in which a view event occurs are the same, and histories that include a view event do not include a crash event.

Definition 3.1 (Linearizability) *An execution σ is linearizable if there exists a continuous subexecution σ' with respect to phases of σ such that there exists a legal linearization τ of σ' such that for every i, $\tau \mid i = \sigma' \mid i$. Also, if σ is benign, then $\sigma = \sigma'$. (This is a non-triviality requirement.)*

An implementation $\mathcal{A}$ of a distributed shared memory is *linearizable*, if every execution generated by $\mathcal{A}$ is linearizable.

Definition 3.2 (Sequential Consistency) *An execution σ is sequentially consistent if there exists a continuous subexecution σ' with respect to phases of σ such that there exists a legal serialization of σ' such that for every i, $\tau \mid i = \sigma' \mid i$. Also, if σ is benign, then $\sigma = \sigma'$. (This is a non-triviality requirement.)*

An implementation $\mathcal{A}$ of a distributed shared memory is *sequentially consistent*, if every execution generated by $\mathcal{A}$ is sequentially consistent.

The following definition of causal memory is a translation of the definition given in [6] to our framework. We start by introducing the following definitions: Given an execution σ, we denote by $\mapsto$ any relation on operations in σ that obeys the following requirements:

- for every two operations $op_1(x,v)$ and $op_2(y,u)$ in σ, if $op_1(x,v) \mapsto op_2(y,u)$, then $x = y$, $v = u$, $op_1(x,v)$ is a write and $op_2(y,u)$ is a read.
- for any operation $op_2(y,u)$, there is at most one operation $op_1(x,v)$ such that $op_1(x,v) \mapsto op_2(y,u)$.
- if $op_2(y,u)$ is a read operation, and there does not exist a write operation $op_1(x,v)$ such that $op_1(x,v) \mapsto op_2(y,u)$, then u is the initial value of y.

Given an execution σ, we denote by $\leadsto$ an acyclic relation which is the transitive closure of a $\mapsto$ relation and $\cup_i \xrightarrow{(\sigma \mid i)}$. That is, for every two operations $op_i^1(x,u)$ and $op_j^2(y,u)$ in σ, $op_i^1(x,v) \leadsto op_j^2(y,u)$ if one of the following conditions holds:

- $op_i^1(x,v) \xrightarrow{\sigma \mid i} op_j^2(y,u)$,
- $op_i^1(x,v) \mapsto op_j^2(y,u)$, or
- there is another operation $op_k^3(z,w)$ such that $op_i^1(x,v) \leadsto op_k^3(z,w) \leadsto op_j^2(y,u)$.

Note that it is not guaranteed that for every execution the relations $\mapsto$ and $\leadsto$ will be defined.

Definition 3.3 (Causal Memory) *An execution σ is causal if there exists a continuous subexecution σ' with respect to phases of σ, such that there exists a relation $\leadsto$ such that for every process p_i, there exists a legal serialization τ_i of $\sigma' \mid (i+w)$ that respects $\leadsto$. Also, if σ is benign, then $\sigma = \sigma'$. (This is a non-triviality requirement.)*

An implementation $\mathcal{A}$ of a distributed shared memory is *causal*, if every execution generated by $\mathcal{A}$ is causal.

4 Implementing Fault Tolerant Distributed Shared Memories

In this section we describe how to implement a fault tolerant distributed shared memory on top of a virtually synchronous communication layer. The following implementations assumes that the interface between the mcs process and the vscl includes the following additional features:

- Whenever there is a need for a configuration change, every mcs process receives from the vscl a *flush* event. Once a flush event occurs, the mcs is not supposed to send any messages until the following view event occurs.
- One of the mcs processes of any view, receives a *flush_ok* event when the new configuration has been computed, but before the view event is sent to any of the members of that view. This process is called the *coordinator* of the view, and it is guaranteed to be a member of the biggest view, prior to the view which is being constructed. (For this purpose, a process that was just awakened is considered to have a singleton view, with only itself in this view.) Following the flush_ok event, the coordinator can pass some information to all the mcs processes, including itself; this information will be delivered to the mcs processes together with the view event.
- Each mcs process can ask the vscl whether a certain message, or event, has been received by all the members of its view. If a message, or an event, has been received by all the members, we say that this message or event is *stable*.

These additional features are all supported by Horus [34], and are part of the Horus Uniform Group Interface. We assume also that the mcs can pass a *stop* event to the application program, which in return must not invoke any new operations until it received a *resume* event from mcs process.

In the implementations below, we use three sending primitives, **abc**, **cbc**, and **send**, in correspondence with the possibility of labeling send events in the definition of virtual synchrony as **atomic**, **causal**, or **unordered**. That is, messages that are sent with **abc** are supposed to be delivered to all their recipients in an atomic order; messages that are sent with **cbc** are supposed to be delivered to all their recipients in a causal order; messages that are sent with **send** are sent point-to-point to their destination, and are delivered without any ordering constraints. Also, we use Return(x, v) to denote a response event for a read that returns a value v from object x and Ack(x) to denote a response event for a write that accessed object x.

Note that the correctness of these protocols depends on the fact that messages are guaranteed to be delivered within the view in which they were sent, a guarantee that is provided by Horus. Note also, that the implementations of linearizability and sequential consistency can be easily generalized to any type of responsive object, e.g., read-modify-write, along the lines of [24].

4.1 Linearizability

The code for implementing linearizability appears in Figure 2, in an event driven form. We assume that each part of the memory is being held by several processes. The exact number depends on the level of fault tolerant that the system is intended for. (A typical number would be 2 or 3 mcs processes for each memory location, although it can be up to n in some cases.) Each mcs process knows who are the mcs processes that are responsible for each memory location, and has a fast way of checking this. This can be done by using a hash function, which in the most simple case would split the memory into continuous pages. In the code, we use the function $resp(x)$ to denote the list of processes which are responsible for memory location x.

Each process holds the following variables: *lts* which is an integer counter that is initialized to 0, *pending* which is an integer register that is initialized to $\perp$, *mem* which is the local copy of the memory, *view* which is the list of processes in the current view, and *state* which indicates the state of the protocol: **running** or **transferring**.

Note that the code in Figure 2 includes a *state transfer* part, which is not defined. In the state transfer part of the algorithm, the copies of the memory held by the different mcs processes must be updated, to ensure that once we restart execution, the behavior of the algorithm will continue to be correct. We now discuss one possible way of implementing a state transfer code.

During a state transfer, we want to reassign copies of the memory to the mcs processes. Typically, what the coordinator would do, is to recalculate which mcs process will be responsible for which memory location. Then, the coordinator would send this information to the rest of the mcs processes that would be in the new view. Each mcs process would respond by sending copies of the memory locations that it held to mcs processes that should now be responsible for them. When the copies are received by every mcs process that needs them, the state transfer is completed, and the mcs processes my discard copies of memory locations that they are no longer responsible for.

We now turn to the proof of correctness. We start with several definitions that will be used in the proof itself.

Given a set of view events E in an execution σ and two view events ev and ev' in E, we say that ev and ev' are *consecutive events in E w.r.t. σ* if in every history h_i that includes both ev and ev', $ev \xrightarrow{h_i} ev'$ and there does not exist another view event ev'' in σ such that $ev \xrightarrow{h_i} ev'' \xrightarrow{h_i} ev'$. A set of view events V in an execution σ is called a *sound partition* if for every pair of consecutive events ev and ev' in V w.r.t. σ, the following holds:

1. $ev.view \cap ev'.view \neq \emptyset$.
2. The coordinator of ev' appears in $ev.view$.
3. For each object x, there exists a process $p_i \in ev.view \cap ev'.view$ such that $p_i \in resp_{ev}(x)$.

Note that with enough redundancy in handling memory objects, there can be more than one sound partition, while if there is not enough redundancy, then a sound partition may not exist. Given a sound partition V, each pair of consecutive view events ev and ev' in V define a *life cycle of* V; ev is called the *beginning of* C and is denoted by $beg(C)$, while ev' is called the *end of* C and is denoted by $end(C)$. The processes that appear in $ev.view \cap ev'.view$ are called the *core* of the life cycle. A set of events E is called the *set of events of a life cycle* C if it is the minimal set of events for which the following holds:

- E includes every event that occurred at a core process of C between $beg(C)$ and $end(C)$.
- E includes every event ev that appears in a history h_i of a non core process of C for which $beg(C) \xrightarrow{h_i} ev$ and ev causally precedes another event in E.

An operation *op* belongs to the set of *operations of a life cycle* C if its call event appears in the set of events of C. The state transfer that is initiated by the coordinator of $beg(C)$, immediately before $beg(C)$, is called the *state transfer of* C. The set of processes in which a state transfer of a life cycle C is finished is called the *base group of* C. Given a life cycle C and an object x, the *initial value of* x *w.r.t.* C in a process p_i which is included in $resp_{beg(C)}(x)$ and in the base group of C is the value of the local copy of x when the state transfer is finished at p_i.

Lemma 4.1 *Any execution generated by the algorithm in Figure 2 for which there exists a sound partition is linearizable.*

Proof: Let σ be an execution generated by the algorithm in Figure 2 for which there exists a sound partition, and let V be one such sound partition. Due to the locality of linearizability [28], it is enough to show that the restriction of σ to operations on any object x (and their related events) is linearizable.

Now, suppose that for each life cycle C, the initial value of x is the same at each process that appears in $resp_{beg(C)}(x)$ and in the base group of C. In this case, σ^x_C, the restriction of $\sigma \mid x$ to the events of $C \mid x$ is linearizable. For this, we can take all the operations of $C \mid x$ and order them in the same order as their corresponding receive events are ordered in σ. Clearly this order, τ^x_C, is a linearization of σ^x_C and for every process p_i, $\sigma^x_C \mid i = \tau^x_C \mid i$.

Note that σ'^x, the collection of all σ^x_C, is a continuous subexecution of $\sigma \mid x$ w.r.t. phases. Hence, due to the state transfer, and since x is initiated to its initial value, the collection of the τ^x_Cs, denoted by τ^x, is a linearization of σ'^x and $\sigma'^x \mid i = \tau^x \mid i$. Hence, by the locality of linearizability, τ is a legal linearization of of σ' and $\sigma' \mid i = \tau \mid i$. ∎

The reason that we send messages to every process in the view, and not just to $resp(x)$, is that the definition that we use does not guarantee a simulation of a fail stop model among messages sent to totally different destinations. Horus has layers that support a definition in which the fail stop model is simulated for all message sent in the same view. By using these layers, we could have sent the messages only to *resp*(x). However, in order to support this definition, protocols that implement it may need to employ one of the following policies, which increases the latency of messages, and may cause a substantial message overhead: (1) the vscl may not allow the layer above it to issue a send whose target does not overlap with a previous send, until it is guaranteed that the recipients of the previous send have either received it, or were removed from the view, or (2) the vscl may buffer incoming messages until it is clear that either all previous messages by the same process have been received by their recipients, or the recipients of these messages have been removed from the view.

4.2 Sequential Consistency

As been noted Attiya and Welch [13], if every process holds a copy of the entire memory, it is possible to implement sequential consistency in such a way that every read is executed immediately and without generating any messages. In such an implementation, only writes may need to be delayed and only writes generate messages. Figure 3 presents a code for such an implementation. This code is basically the same algorithm as the one developed by Attiya and Welch in [13], but modified for a vscl environment.

Lemma 4.2 *Any execution generated by the algorithm in Figure 3 is sequentially consistent.*

The proof of 4.2 is based on the proof of a similar lemma to the benign case, given by Attiya and Welch is [13]. Note that since every process in the algorithm in in Figure 3 holds a copy of every object, then for every execution generated by this algorithm there exists a sound partition.

The entire state transfer of the algorithm in Figure 3 is included in the view event and for every object x, *resp*(x) is the entire view. Keeping this in mind, the definition of the initial value of an object x in a life cycle is the value of x immediately after the view event.

Proof: Let σ be an execution generated by the algorithm in Figure 2 and let V be a sound partition of σ. Now, suppose that for each live cycle C, the initial value of of every object x in C is the same at each process in the base group of C. In this case, σ_C, the restriction of σ to events in C is sequentially consistent. For this, we can take all the writes in C and order them in the same order in which their receive events are ordered in σ, to create a sequence of operations τ_C. Then, we order every read r_k in C between the writes w_i^1 and w_j^2 in τ_C for which the call event of r is ordered in h_k between the receive events that correspond to w_i^1 and w_j^2, breaking ties among reads by the same process by ordering them in the same order as their call events in their invoking process and reads by different processes arbitrarily. (If there is no such w_i^1, then r_k is placed before w_j^2; if there is no such w_j^2, then r_k is placed after w_i^1; if there is neither such w_i^1 nor such w_j^2, then r_k is placed somewhere in τ_C.) Clearly τ_C is a legal serialization of σ_C and for every process p_i, $\sigma_C \mid i = \tau_C \mid i$.

Read(x):
    $lts := lts + 1$; $pending := lts$
    abc $<$ `read`, x, $lts >$ to *view*

Write(x,v):
    $lts := lts + 1$; $pending := lts$
    abc $<$ `write`, x, v, $lts >$ to *view*

received $<$ `read`, $x, t >$ from p_j:
    if I am included in $resp(x)$ then
        send $<$ `val`, x, $mem[x]$, $t >$ to p_j
    endif

received $<$ `write`, $x, v, t >$ from p_j:
    if I am included in $resp(x)$ then
        $mem[x] := v$; **send** $<$ `done`, $x, t >$ to p_j
    endif

received $<$ `val`, $x, v, t >$ from p_j:
    if $pending = t$ then generate Return(x, v); $pending := \perp$ endif

received $<$ `done`, $x, t >$ from p_j:
    if $pending = t$ then generate Ack(x); $pending := \perp$ endif

received **flush**
    generate Stop

received **flush_ok**
    tell the vscl that it can deliver the view
    start a state transfer

received **view**
    update my *view*; *state* := **transferring**

if *state* =**transferring** and the state transfer is stable then
    state := **running**; generate Resume
endif

Figure 2: Linearizable implementation — code for process p_i.

Read(x):
    generate Return(x, $mem[x]$)

Write(x,v):
    abc $<$ `write`$, x, v >$ to *view*

received $<$ `write`$, x, v >$ from p_j:
    $mem[x] := v$
    if $p_i = p_j$ then generate Ack(x) endif

received **flush**
    generate Stop

received **flush_ok**
    tell the vscl that it can deliver the view and pass my copy of the
    memory to every other process in the view, together with the view.

received **view**
    update *view* and *mem*

Figure 3: Sequentially consistent implementation — code for process p_i.

Note that σ', the collection of all σ_C, is a continuous subexecution of σ w.r.t. phases. Hence, since copies of the memory are sent by the coordinator with each view to all the members, and since every object is initiated to its initial value, the collection of the τ_Cs, denoted by τ is a legal serialization of σ' and $\sigma' \mid i = \tau \mid i$. ■

In order to reach a higher degree of memory efficiency, it is possible to use algorithms in which the mcs process at each node holds only part of the memory. Modifying these algorithms to virtually synchronous environments can be done very similarly to what we have done with the protocol of Attiya and Welch. For reasons of fault tolerance, the algorithm would have to make sure that each memory location is always being held by more than one mcs. Also, such algorithms would need a more elaborated state transfer phase, similar to the one discussed in Section 4.1. However, as been noted by [22], such a solution for sequential consistency may require that each send event, which is labeled as **atomic**, will be labeled as **causal** too.

4.3 Causal Memory

The code for implementing causal memory appears in Figure 4. Here, too, we assume that each process has a copy of the entire memory. It is basically the same code as in Figure 3,

Read(x):
    generate Return(x, $mem[x]$)

Write(x,v):
    cbc $<$ `write`$, x, v >$ to *view*

received $<$ `write`$, x, v >$ from p_j:
    $mem[x] := v$
    if $p_i = p_j$ then generate Ack(x) endif

received **flush**
    generate Stop

received **flush_ok**
    tell the vscl that it can deliver the view and pass my copy of the
    memory to every other process in the view, together with the view.

received **view**
    update *view*
    if I am a new member in this view then update *mem* endif

Figure 4: Causal memory implementation — code for process p_i.

but it uses **cbc** instead of **abc** to broadcast messages. Note that **cbc** can always do a local delivery of a message immediately. Hence, both reads and writes are executed immediately. Also, note that by using the **cbc** service of vscl, the code of the algorithm becomes very simple since, unlike other implementations of causal memory [4, 5, 6], we do not need to implement causal broadcast as part of our memory consistency protocol. The proof of this algorithm is very similar to the proof of the algorithm for sequential consistency, and is therefore dropped from this paper.

5 Discussion

In recent years it becomes evident that distributed shared memories are important in order to be able to utilize large scale multiprocessors, and to treat a cluster of workstations as one parallel machine. However, it is also clear that in order to do so, one must develop highly efficient fault tolerant algorithms for implementing distributed memory in asynchronous networks. In this paper we proposed to do so by using virtually synchronous communication systems, located between the distributed shared memory protocols and the network itself. We showed how to redefine consistency conditions for environments that provide virtual synchrony, and demonstrated this definition method for three well known consistency conditions, namely, linearizability, sequential consistency, and causal memory. Finally, we showed how to turn existing protocols that implement these consistency conditions into protocols that work correctly and efficiently in virtually synchronous settings, and proved their correctness.

However, this work is only one step towards achieving this goal. For example, the work discussed in this paper assumes that each node may have at most one operation pending at a time, and it would be interesting to develop a framework that allows pipelining and out-of-order execution of operations, as well as branch predictions, and would support virtual synchrony. In particular, it would be interesting to merge the framework developed in [10] with the one developed here. It may also be interesting to apply this framework to recent commercial architectures, like DEC-Alpha [1] and PowerPC [20], the way the framework of [10] is applied to the DEC-Alpha in [12].

This work did not address the issue of how the application program can deal with failures. This issue includes several important questions. For example, what to do with tasks that were executing on machines that crashed? Do we need to restart these tasks, or is there a way to create task redundancy like we do with memory? These questions are left open for further research.

References

[1] *The Alpha Architecture Handbook*. Digital Equipment Corporation, 1992.

[2] S. Adve and M. Hill. Weak Ordering—A New Definition. In *Proc. of the 17th International Symposium on Computer Architecture*, pages 2–14, May 1990.

[3] S. Adve and M. Hill. A Unified Formalization of Four Shared-Memory Models. Technical Report 1051, Computer Science Department, University of Wisconsin, Wisconsin-Madison, September 1991.

[4] M. Ahamad, J. Burns, P. Hutto, and G. Neiger. Causal Memory. In *5th International Workshop on Distributed Algorithms*, Greece, October 1991.

[5] M. Ahamad, P. Hutto, and R. John. Implementing and Programming Causal Distributed Shared Memory. Technical Report TR GIT-CC-90-49, Georgia Institute of Technology, December 1990.

[6] M. Ahamad, G. Neiger, P. Kohli, J. Burns, and P. Hutto. Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing*, 9(1).

[7] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *Proc. of the 22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[8] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella. Fast Message Ordering and Membership Using a Logical Token-Passing Ring. In *Proc. of the 13th International Conference on Distributed Computing Systems*, pages 551–560, May 1993.

[9] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing Memory Robustly in Message Passing Systems. In *Proc. of the 9th ACM Symposium on Principles of Distributed Computing*, pages 363–375, 1990.

[10] H. Attiya, S. Chaudhuri, R. Friedman, and J. Welch. Shared Memory Consistency Conditions for Non-Sequential Execution: Definitions and Programming Strategies. In *Proceedings of the 5th ACM Symposium On Parallel Algorithms and Architectures*, pages 241–250, June/July 1993. Also: Technical Report LPCR #9306, Department of Computer Science, The Technion. Submitted for publication.

[11] H. Attiya and R. Friedman. A Correctness Condition for High-Performance Multiprocessors. In *Proc. 24th ACM Symp. on the Theory Of Computing*, pages 679–690, May 1992. Also: Technical Report #767, Department of Computer Science, The Technion. Submitted for publication.

[12] H. Attiya and R. Friedman. Programming DEC-Alpha Based Multiprocessors the Easy Way. In *Proc. 6th ACM Symp. on Parallel Algorithms and Architectures*, pages 157–166, June 1994. Also: Technical Report LPCR #9411, Department of Computer Science, The Technion. Submitted for publication.

[13] H. Attiya and J. Welch. Sequential Consistency versus Linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, May 1994.

[14] J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proc. 2nd ACM Symp. on Principles and Practice of Parallel Processing*, pages 168–176, 1990.

[15] K. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[16] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[17] W. Brantley, K. McAuliffe, and J. Weiss. RP3 Processor-Memory Element. In *Proc. Internation Conf. on Parallel Processing*, pages 782–789, 1985.

[18] L. M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Trans. on Computers*, C-27(12):1112–1118, 1978.

[19] W. W. Collier. Architectures for Systems of Parallel Processes. Technical Report IBM TR 00.3253, Poughkeepsie, NY, January 1984.

[20] F. Corella, J. Stone, and C. Barton. A Formal Specification of the PowerPC Shared Memory Architecture. Technical Report Research Report #RC18638, IBM, 1994.

[21] M. Dubois and C. Scheurich. Memory Access Dependencies in Shared-Memory Multiprocessors. *IEEE Trans. on Software Engineering*, 16(6):660–673, June 1990.

[22] A. Fekete, F. Kaashoek, and N. Lynch. Providing Sequentially Consistent Shared Objects Using Group and Point-To-Point Communication. In *Proc. 15th International Conference on Distributed Computing Systems*, 1995. To appear.

[23] M. Fischer, N. Lynch, and M. Patterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.

[24] R. Friedman. Implementing Hybrid Consistency with High-Level Synchronization Operations. In *Proc. 12th Annual ACM Symp. on the Principles of Distributed Computing*, pages 229–240, August 1993. Also: Technical Report #781, Department of Computer Science, The Technion. Submitted for publication.

[25] R. Friedman and R. van Renesse. Strong and Weak Virtual Synchrony in Horus. Technical Report TR95–1491, Department of Computer Science, Cornell University, March 1995.

[26] P. Gibbons and M. Merritt. Specifying Non-Blocking Shared Memories. In *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures*, pages 306–315, July 1992.

[27] P. Gibbons, M. Merritt, and K. Gharachorloo. Proving Sequential Consistency of High-Performance Shared Memories. In *Proc. of the 3rd ACM Symposium on Parallel Algorithms and Architectures*, pages 292–303, July 1991.

[28] M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.

[29] L. Lamport. Time, Clocks and the Ordering of Event in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[30] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-28(9):690–691, 1979.

[31] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. on Computer Systems*, 7(4):321–359, 1989.

[32] S. Min and J. Baer. A Timestamp-Based Cache Coherence Scheme. In *Proc. International Conf. on Parallel Processing*, pages I–23–32, 1989.

[33] U. Ramachandran, M. Ahamad, and M. Y. Khalidi. Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer. In *Proc. International Conf. on Parallel Processing*, pages II–160–169, 1989.

[34] R. van Renesse, T. M. Hickey, and K. P. Birman. Design and performance of horus: A lightweight group communications system. Technical Report 94-1442, Cornell University, Dept. of Computer Science, August 1994.