Fundamentals for Consistent Event Ordering in Distributed Shared Memory Systems

Jörg Preißinger, Tobias Landes
Institut für Informatik
Technische Universität München
Germany

Abstract

A large number of tasks in distributed systems can be traced down to the fundamental problem of attaining a consistent global view on a distributed computation. This problem has been addressed by a number of studies which focus on systems with message passing as their only means of interprocess communication. In the paper at hand we extend this restricted system model by additionally accounting for an abstract memory to be shared by the processes. We specify necessary and sufficient conditions for constructing a consistent global view on such systems and present helpful definitions, which are meant to be a solid formal base for further studies.

Keywords: *distributed system, distributed shared memory, consistency, event order, observation*

1 Introduction

In distributed computing there is a large number of tasks which can all be traced down to the fundamental problem of constructing a consistent global view on a distributed computation. Examples of these tasks are monitoring, breakpointing, debugging, and the detection of deadlocks or global predicates in general. All of these tasks are part of the necessary effort to better understand, design, and control distributed systems.

The fundamental problem of constructing a consistent global view derives from the need to issue meaningful statements about a whole distributed computation on the one hand, and the lack of a global common time base on the other. The latter issue must be overcome by ordering the events of the computation based on their mutual causal dependencies rather than a global clock. Several approaches and solutions have been given, with varying focus and background, in the related work of Lamport [2][5] and many others. But all of these authors based their examinations on a system model that has its distributed processes communicate solely via messages sent from one process to another.

With the paper at hand, we focus on an extension of this common model by accounting for an abstract memory to be shared by the distributed processes. Some related work has been presented by Babaoğlu/Marzullo in [1] and by Li/Girard in [15], both in the context of general consistency models and their verification respectively. Our goal is the consolidation of all the relevant parts of the related work on the one hand, and the suggestion of a set of useful definitions, interconnections, and basic considerations on the other. This work is meant as a base to be worked upon and to be further refined in future research.

The document is organized as follows. In section 2 we present the concepts, results, and terminology in the context of message passing systems since these are an important base for our work. In section 3 we gradually extend these concepts by introducing a shared memory and the resulting problems, and by giving and refining basic definitions. We also prove a theorem that states necessary and sufficient conditions for a consistent global order. Section 4 summarizes the paper.

2 Mere Message Passing Systems

Virtually all existing considerations regarding consistent views on distributed computations focus on a system model that has its distributed processes communicate solely via messages sent from one process to another. Starting with the milestone paper [5] of Chandy and Lamport in 1985, the underlying system models of distributed consistency considerations, whether they aim at system monitoring, breakpointing or detecting global predicates for debugging purposes, have been widely the same and vary only in minor assumptions. Since our work is an extension of these examinations, this section will briefly summarize the most important concepts, results and terminology concerning consistent views on mere message passing systems.

2.1 System Model

In the basic model, a distributed computation consists of a finite set $P = \{p_1, p_2, \ldots, p_n\}$ of *n processes*. The pro-

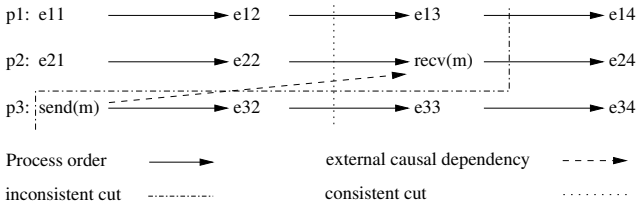

Figure 1: Dependencies and Cuts

cesses communicate only by sending and receiving *messages*, which are assumed to be delivered reliably and with a finite delay. Sometimes it is also assumed that the messages be delivered exactly in the order they have been sent, which establishes the notion of *FIFO channels*. FIFO channels can be implemented by using logical or vector clocks [2][1].

Any process p_i consists of a sequence of *events* $E_i = \{e_i^1, e_i^2, \ldots\}$ which are totally ordered by an ordering relation $\rightarrow$ called the *program order*. Each event is atomic on the viewed abstraction level and changes the *state* of the process. Of particular interest for considerations regarding the global behaviour of systems with interacting processes are events representing the sending or receiving of a message, i.e. *send* and *receive events*. This is because these events establish synchronization dependencies among the processes and thus extend the local program order to a partial global ordering $\Rightarrow$ of events. Lamport [2] called this the "happened before" relation, and defined it in the following way as the transitive closure of the program order and the natural causal send-receive dependencies:

Definition 2.1. The *"happened before" relation* $\Rightarrow$ is the smallest relation satisfying the following conditions: If $e_i^x \rightarrow e_i^y$, then $e_i^x \Rightarrow e_i^y$. If e_i^x is a send event and e_j^y is the receive event of the same message, then $e_i^x \Rightarrow e_j^y$. If $e_i^x \Rightarrow e_j^y$ and $e_j^y \Rightarrow e_k^z$, then $e_i^x \Rightarrow e_k^z$.

If $e_i^x \Rightarrow e_j^y$, then e_j^y is regarded as being *causally dependent* on e_i^x, since it can only be executed if the execution of e_i^x has already been finished. Therefore, e_i^x can also be seen as a *precondition* to e_j^y. Intuitively, this means for example that a message can not be received before it has been sent. If $e_i^x \not\Rightarrow e_j^y$ and $e_j^y \not\Rightarrow e_i^x$, then e_i^x and e_j^y are said to be *concurrent*, and may be executed in parallel since none of them can causally affect the other. We denote concurrency by $e_i^x \| e_j^y$.

2.2 Consistent Views

Given this system model, one can begin to examine the issues of retrieving a consistent view on a specific computation. The view has to reflect the *global state* of the computation, which is, naturally, the set of the states of all the processes involved. The fundamental problem is to assemble the local process states in a way that guarantees a meaningful global state, which is not trivial due to the fact that the system lacks a global clock. A meaningful or *consistent global state* or *consistent cut*[1] is one that may have occurred in an actual *run* of the computation (though there is no way to ensure it actually has). Essentially, this means that for every local process state present in the global state, all events on which it causally depends must also be reflected by local states present in the global state. For example, the global state must not include a process state that reflects the process to have received a message, when the included state of the sender does not yet reflect the corresponding send event. Figure 1 gives an example showing a consistent and an inconsistent cut for a computation involving two processes. A cut divides the computation into a "past" and a "future" section, defining the set of process states it consists of[2] as the "present". The global state is consistent if no causal dependency (according to Definition 2.1) points from the future to the past.

Algorithms that allow an *observer* or *monitor* process to assemble a consistent global state or adapt the general problem to generating distributed breakpoints have been given in [5] to [14]. Due to the restricted system model and since the local process events are already totally ordered by the program order, the problem is essentially reduced to catching the causal dependencies arising from message passing as the only means of interprocess communication.

3 Systems with Distributed Shared Memory

In this section we are going to expand the basic system model described above by accounting for a shared memory as a very powerful and intuitive means of interprocess communication. We will not consider the nature of this memory; it may be physically distributed with the sharing implemented by an underlying abstraction software as well as a non-distributed physically shared memory. In the following we will refer to the memory as a distributed shared memory (DSM). As in the previous sections, the goal is to establish a consistent view on a specific distributed computation, which leads to the basic problem of constructing a consistent order of events. Systems with DSM are abbreviated DSMS in contrast to MPS which denotes mere message passing systems as described in the previous section.

3.1 Differences to Message Passing Systems

The task of ordering the computational events consistently is significantly more complex in DSMS than in MPS

[1] Since in mere message passing systems both cuts and global states can be given by a set of local states, one for each process, the terms are used interchangeably throughout this section. For a more detailed discussion see section 3.3.

[2] The arrows between the events of a process can also be seen as the process's states.

for several reasons. The important issue in both cases is the passing of information among the processes. Any possible flow of information makes the receiver potentially dependent on the provider, and the detection of these causal dependencies is essential for deriving a consistent event order. Obviously, the flow of information is much harder to trace in DSMS than in MPS. Whereas a message is unique and delivers information from one sender to one specific receiver, memory objects are generally used to pass information anonymously to an unspecified group of (possible) readers and could be replicated on several physical locations at the same time, each copy possibly holding a different value (depending on the consistency model of the DSM).

We will apply the knowledge of MPS to suit the additional needs of analyzing DSMS. In particular we expand the system model by introducing read and write events, which are assumed to atomically access *shared memory locations*. We will refine the "happened before" relation in order to support our goals concerning DSMS. The refined "happened before" relation will be based on formally defined sufficient requirements to totally order the observed events consistently. We will further elaborate the dependency relation by adding requirements that serve to reduce the complexity of ordering the events consistently and are based on the work of Gibbons and Korach [18].

The variety of consistent orders depends on the consistency model of the DSM. A given order of events may be considered consistent in a system implementing release consistency, that may never have occurred in system respecting sequential consistency. The consistency model specifies the behaviour of a DSMS regarding concurrent memory accesses and thus is essential for the decision whether an order of the access events is to be considered consistent or not. For a survey on DSM consistency models see Adve and Gharachorloo's tutorial [19]. In the next section the consistency model independent order restrictions are presented. Afterwards, we explain the influence of the consistency model on finding a consistent event order, based on the widely used sequential consistency.

3.2 Consistency Model Independent Order

The program order, as given by Lamport, allows two different interpretations for distributed systems. In the research of consistency model verification the program order is interpreted as the total order of all events on one processor [16][17][18] without a distinction by process. We will call this the *processor order* to mark the contrast to the *program order* $\rightarrow$, which we see strictly as the total order of local events for each process. It is obvious that only global orders respecting the program order can be consistent. For consistency model verification, the distinction between processor and program order is unimportant because one can assume that only one process is running on each processor. But when observing distributed computations, the distinction is very important because the concern is the behaviour of all processes, including multiple processes on one processor. Considering the processor order instead of the program order would mean a loss of information regarding the potential concurrency of causally independent events on a given processor. The lost information could be significant for some applications of the consistent view, for example load balancing based on possible performance gains by maximizing parallelism through process migration. Since our goal is to gather more information about a distributed system, our only choice is to base our work on the program order, which we will from now on call *process order* to highlight the contrast to the processor order.

Analogous to the send-receive dependencies, there exist write-read dependencies for shared memory locations. We abbreviate a write event / read event that writes/reads a value a to/from the logical memory location x as $W(x)a/R(x)a$. A value can only be read from a location after it was written to it, so any read event $R(x)a$ of a value a from location x must be ordered after the first respective write event $W(x)a$ of that value to that location. This is similar to the send-receive dependency, with the difference that several identical write events may have occurred. So we claim that at least one matching write event must have happened before the first read event. There is one exception to this rule: Every memory location has an undefined and thus arbitrary state before the first value is written to it. As a consequence, a read event could potentially read any value from a location if it only could occur before the first write event, with all other causal dependencies left unharmed. Regardless of the sense of such improperly synchronized system behaviour is this case possible and therefore has to be considered consistent. Any further read events reading other values from the same location must of course be ordered after any such questionable read event. We regard this case with the objective of completeness, even if it won't often occur in real systems.

We extend the definition of "happened before" to the following *preliminary DSMS causality relation* $\overset{p}{\Rightarrow}$ with $E_{W(x)a}/E_{R(x)a}$ meaning a subset of all events E, consisting of all events $W(x)a/R(x)a$:

Definition 3.1. Let $E_{R(x)a}$ be the set of all read events reading the value a from location x and let $E_{W(x)}$ be the set of all write events to location x. The *preliminary DSMS causality relation* $\overset{p}{\Rightarrow}$ is the smallest relation satisfying the following three conditions:

$\forall e_i, e_j \in E$: if $e_i \Rightarrow e_j$, then $e_i \overset{p}{\Rightarrow} e_j$.

$\forall e \in E_{R(x)a}$: one of the following two cases must hold:

(i) $\exists e_w \in E_{W(x)a} : e_w \overset{p}{\Rightarrow} e$.

(ii) $\forall e_x \in E_{W(x)} : (e \overset{p}{\Rightarrow} e_x)$ and
$\forall e_r \in E_{R(x)b}$: case (i) must match, for $a \neq b$.

$\forall e_i, e_j, e_k \in E$: if $e_i \overset{p}{\Rightarrow} e_j$ and $e_j \overset{p}{\Rightarrow} e_k$, then $e_i \overset{p}{\Rightarrow} e_k$.

3.3 Consistency Model Dependent Order

The consistency model of a distributed shared memory specifies what orders of potentially concurrent memory access events are allowed to effectively occur. Adve and Gharachorloo recapitulated in [19] that "effectively, the consistency model places restrictions on the values that can be returned by a read in a shared-memory program execution." Consequently, a consistent observation requires to construct the observed event order with respect to the given consistency model. In fact, the consistency model provides an abstraction from the actual implementation.

Sequential consistency is a widely used and semantically simple consistency model, so, in favor of better understanding, we will use it to demonstrate how the consistency model affects the process of ordering events consistently. This proceeding should also give a basic idea of how to apply a similar approach to other models in continuative work.

Sequential consistency was defined by Lamport [20], and cited by Raynal [21] informally as follows: Sequential consistency "states that a multiprocess program executes correctly if its results could have been produced by executing that program on a single processor system. This means that an execution is correct if we can totally order its operations in such a way that (1) the order of operations in each process is preserved, and (2) each read gets the last previously written value, *last* referring here to the total order." Ordering events consistently regarding the process order is not a problem since it is already included in the "happened before" relation (Definition 2.1) which was defined for MPS, but is used as a base for all our DSMS relations in this section. Providing the necessary information can be done using the common techniques based on logical clocks, just like in MPS. In addition, we have to order all write and read events. Events accessing different memory locations are causally independent (if there are no other dependencies) so that any order respecting their process order and message passing dependencies can be considered consistent. This is why from now on we focus on events respectively accessing the same memory location. The assumption that the DSM implements sequential consistency enables us to abstract from actual memory locations or replicas.

The second part of Raynal's sequential consistency definition (2) means that every read event of a specific value must be ordered after a matching write event, but before the next write event to that location, "next" referring here to the total order and "matching" to any write event that wrote the read value to the memory location, without differentiating between identical write events. The necessary precondition for reading a value a from location x is the existence of value a in location x. Thus we do not regard the read event as causally dependent on the specific write event that actually wrote the value, but as dependent on the state of the memory location, which in turn is dependent on the last write event that accessed it. So we claim that a read event $R(x)a$ could have occurred after any write event writing the value a to location x. Additionally, we have to account for the case that the unknown initial value of the memory location could be read as described in chapter 3.2. The preliminary DSMS causality relation defined above is not sufficient for these demands, so we state the following theorem:

Theorem 1. Let $E_{R(x)a}$ be the set of all read events reading the value a from location x, let $E_{W(x)a}$ be the set of all write events writing a to location x. Respecting the following three conditions is necessary and sufficient for any total order $\succ$ to be consistent:[1]

(1) $\forall e_i, e_j \in E:$ if $e_i \rightarrow e_j$, then $e_i \succ e_j$.

(2) $\forall e_i^x, e_j^y \in E:$ if e_i^x is a send event and e_j^y is the receive event of the same message, then $e_i^x \succ e_j^y$.

(3) $\forall e \in E_{R(x)a}$ one of the following cases must hold:

(i) $\exists e_w \in E_{W(x)a}: (e_w \succ^* e)$ and $(\forall e_x \in E_{W(x)} \setminus \{e_w\}: (e_x \succ^* e_w)$ or $(e \succ^* e_x))$.

(ii) $\forall e_x \in E_{W(x)}: (e \succ^* e_x)$ and $\forall e_r \in E_{R(x)b}:$ case (i) must match, for $a \neq b$.

Proof: First we prove that every condition is necessary for any total order to be consistent. Remember that an order is consistent if and only if it could have occurred during an execution of the system. The first two conditions are directly derived from Lamport's happened-before relation and their necessity is known from MPS [5]. Suppose the total order $\dot{\succ}$ is consistent but harms condition (3). During the system's execution a read event could read either a value that was initially in the memory, if no write event accessed that memory location so far, or a value that was written by a write event. There are two possibilities for $\dot{\succ}$ to harm condition (3). In the first case $\dot{\succ}$ orders two read events $e_i \in E_{R(x)a}$ and $e_j \in E_{R(x)b}$, reading different values, before the very first write event to location x, which could not happen in an actual execution and thus leads to a contradiction to $\dot{\succ}$ being consistent. In the second case, $\dot{\succ}$ orders a write event $e_x \in E_{W(x)b}$ between a write event $e_w \in E_{W(x)a}$ and following read events $e \in E_{R(x)a}$. This means that a read event returns an already overwritten value, which could not happen in a sequentially consistent shared memory system and thus leads again to a contradiction.

Now we prove that the conditions stated in Theorem 1 are sufficient for any total order to be consistent. Suppose to have a total order $\dot{\succ}$ respecting all the conditions that is inconsistent. Consequently, $\dot{\succ}$ orders an event e_i after e_j without harming our conditions, whereas this sequence could not occur in an actual execution. There are three possibilities for an order $\dot{\succ}$ to be inconsistent. In the first case, e_i happens before e_j according to the process order (both events belonging to the same process), but is ordered $e_j \dot{\succ}^* e_i$. In this case the assumption that $\dot{\succ}$ holds condition

[1] $e_i \succ^* e_j$ is the common notation for a transitive path in $\succ$ from e_i to e_j.

(1) leads to a contradiction. The second case is similar, with e_j being the receipt and e_i the sending of the same message, and contradictory to condition (2). Third, a read event $e_j \in E_{R(x)a}$ can only occur if the memory location x holds the value a, which is only the case if the last write event, according to $\dot{\succ}$, that accessed location x wrote value a. Thus the – by assumption – inconsistent order $\dot{\succ}$ would not order read event $e_j \in E_{R(x)a}$ after a write event $e_i \in E_{W(x)a}$ without other write events in between, or it would order at least two different read events $e_i \in E_{R(x)a}$ and $e_j \in E_{R(x)b}$ before the very first write event accessing that location. Both cases lead to a contradiction due to condition (3). □

This leads to our final definition of the *DSMS causality relation* $\overset{c}{\Rightarrow}$, with $E_{W(x)}$ denoting the set of all write events to memory location x:

Definition 3.2. A *DSMS causality relation* $\overset{c}{\Rightarrow}$ is any transitive order relation satisfying the conditions demanded by Theorem 1.

Due to Theorem 1, any partial or total event order respecting this relation is consistent.

Another problem is the constructive building of a partial or total order that respects this relation. As Gibbons and Korach showed in their work about the verification of consistency models [18], the problem of finding a sequentially consistent total order of events in shared memory systems is, without further information, np-complete. Two problems remain for finding such a total order:

- The order of several write events $e_w \in E_{W(x)}$ to the same memory location.
- The mapping of a read event $e_r \in E_{R(x)a}$ to a write event $e_w \in E_{W(x)a}$, after which it can be ordered without harming other dependencies.

Gibbons and Korach introduced the *write-order* and the *read-mapping*, and proved that by providing this additional information a sequentially consistent total order can be constructed in $O(n\ log(n))$. This leads to even more order restrictions and therefore to less total orders possibly detected. But, and this is good enough for many applications, it always yields a consistent total order – assuming the shared memory actually implements sequential consistency – because the additional order restrictions are won from the actual occurrence of the events in the observed execution. We will now explain their results, and then adjust our definition of the DSMS causality relation to an order restricting version as a basis for efficient implementations.

The write-order is the total order of all write events to the same memory location as occurred in the observed system execution. This information must be collected during the execution and then be provided to the ordering algorithm. The total order is constructed with respect to the write-order, which is abbreviated $\succ_{wo}$, $e_1 \succ_{wo} e_2$ meaning that e_2 is ordered after e_1.

In real systems identical values can be written to the same memory location by different processes. A read event reading such a value can generally be ordered after any of those identical write events. Although these ordering possibilities are often restricted by other dependencies, a read event can naturally always be ordered after the write event that in fact wrote the read value during the observed execution. The read-mapping is a function $f_{rm} : E_{R(x)a} \mapsto E_{W(x)a} \cup \{\bot\}$ that assigns to each read event the unique actually corresponding write event. Again the read-mapping must be collected during the system execution to provide its serviceable information for the construction of a total event order in which each read event is ordered after the mapped write event. The read events themselves are only ordered by process order (and possibly indirectly by message passing) because reads of different processes are (potentially) concurrent and may be executed in parallel. If a read event reads the initial value of a memory location, the read-mapping maps it to $\bot$, in which case it must be ordered before the first write event to the corresponding memory location.

Given this additional information we can define a *restricted DSMS causality relation* $\overset{r}{\Rightarrow}$ based on Definition 3.2, the provided write-order $\succ_{wo}$, and the read-mapping f_{rm}:

Definition 3.3. Let $\succ_{wo}$ totally order all write events $E_{W(x)}$ to the same location, respectively. Let $f_{rm} : E_{R(x)a} \mapsto E_{W(x)a} \cup \{\bot\}$ be a read-mapping function that maps every read event to its corresponding write event, or to $\bot$ if no write event accessed that location before. The *restricted DSMS causality relation* $\overset{r}{\Rightarrow}$ is the smallest relation satisfying the following four conditions:

$\forall e_i, e_j \in E$: if $e_i \Rightarrow e_j$, then $e_i \overset{r}{\Rightarrow} e_j$.

$\forall e_i, e_j \in E_{W(x)}$: if $e_i \succ_{wo} e_j$, then $e_i \overset{r}{\Rightarrow} e_j$.

$\forall e \in E_{R(x)a}$ one of the following two cases must match:

(i) $\exists e_w \in E_{W(x)a} : (f_{rm}(e) = e_w)$ and $(e_w \overset{r}{\Rightarrow} e)$ and $(\forall e_x \in E_{W(x)} \setminus \{e_w\} :$ if $(e_w \succ_{wo} e_x)$, then $(e \overset{r}{\Rightarrow} e_x))$.

(ii) $(f_{rm}(e) = \bot)$ and $(\forall e_x \in E_{W(x)} : e \overset{r}{\Rightarrow} e_x)$.

$\forall e_i, e_j, e_k \in E$: if $e_i \overset{r}{\Rightarrow} e_j$ and $e_j \overset{r}{\Rightarrow} e_k$, then $e_i \overset{r}{\Rightarrow} e_k$.

Note that the total event orders satisfying the restricted DSMS causality relation are only a subset of all possible consistent total orders, which is a drawback in comparison to the DSMS causality relation (Definition 3.2). The benefit of the restricted DSMS causality relation is that it enables us to construct a consistent total order efficiently. This is why we base the remainder of this paper on the restricted relation.

Now that we can construct a total order of events, we can define consistent cuts and consistent global states for DSMS as a global state or cut that does not violate the restricted DSMS causality relation in the sense that it reflects the effect of a dependency but not its cause or origin:

Definition 3.4. Let $P = \{p_1, p_2, \ldots, p_n\}$ be the finite set of processes and E the set of all events. Let C be a cut consisting of the set of local process states $S = \{s_1, s_2, \ldots, s_n\}$ and $E_c = \{e_1, e_2, \ldots, e_n\}$ be the set of events respectively preceding the states in S. Let $E_p = E_c \cup \{e_i \mid e_i \overset{r}{\Rightarrow} e_j,\ e_j \in E_c\}$ be the set of all events in the "past" of the cut. C is a *consistent cut* if the following condition holds: $\forall e_i, e_j \in E$: if $e_i \overset{r}{\Rightarrow} e_j$ and $e_j \in E_p$ then $e_i \in E_p$.

In mere MPS any cut as defined above is equivalent to a global state since the state of the system is properly characterized by a complete set of the local states of the processes,[1] as is, by definition, the cut. In DSMS the situation is inherently different because a global state, seen as a comprehensive description of the system's state providing all the information necessary, for example, to take a snapshot to be used as a rollback breakpoint, is required to reflect not only the process states, but also the state of the shared memory. Since acquiring a memory state consistent with a given cut is a complex and difficult task, in this paper we will capture it implicitly by defining the global state based on the totally ordered history that has led to the cut's process states.[2] This history is given by the $\overset{r}{\Rightarrow}$ relation and implicitly provides us with the means to reconstruct the memory state by answering the question which one was the last write to each memory location, respectively. Note that we require the initial state of each memory location, as only then we can reconstruct it in the case that no write operation has been performed up to the given global state.

Definition 3.5. Let P, C, S, E_c and E_p be defined as of Definition 3.4. A *global state* is a tuple $T = (E_p, \succ)$, where E_p is totally ordered by $\succ$. If C is consistent and $\succ$ respects all the conditions demanded by Theorem 1 (which both $\overset{c}{\Rightarrow}$ and $\overset{r}{\Rightarrow}$ do), T is a *consistent global state*.

3.4 Illustrations

We now provide two figures illustrating the discussed results. Figure 2 shows three processes $p1$, $p2$ and $p3$, several read and write events, and a passed message. The events are only ordered by process order, other dependencies are not displayed and thus consistently ordering the events totally, in order to identify a consistent cut or global state, would be an np-complete task. Without further information no global states of the system can be specified and thus no conclusions about the underlying system can be drawn.

Figure 3 presents the same events with the restricted DSMS causality relation drawn in. The relation is split up into the parts derived from "happened before" relation, write-order and read-mapping. The events are already

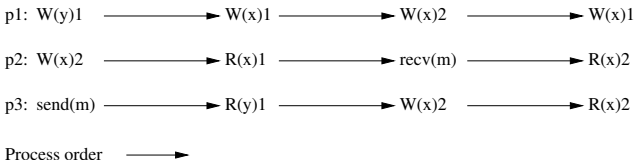

Figure 2: Events ordered by process order

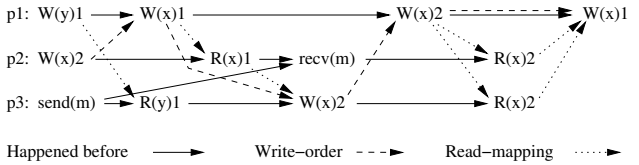

Figure 3: Events ordered by the restricted DSMS causality relation

spaced out according to the rule that the causal effect must be ordered right of its dependencies, so that they can be viewed as being partially ordered with respect to the relation. To say it informally, the more to the left an event is arranged the earlier it happened in the causality chain. The normal arrows denote the "happened before" relation as known from MPS, which is still the base of our relations. The $send(m)$ and $recv(m)$ events indicate that the message passing paradigm and its ordering rules are applicable as known. The dashed arrows display the write-order included in the restricted DSMS causality relation: All write events to the same location are ordered totally as observed during system execution. The read-mapping included in the relation is shown by the dotted arrows. This ensures every read event being ordered after the corresponding write event, but before any further write events to the respective location. For example, the two read events $R(x)2$ of $p2$ and $p3$ are ordered after the second write $W(x)2$, but before location x is overwritten by $W(x)1$ of $p1$. The events displayed vertically one below the other are causally independent and thus concurrent; no directed path leads from one to the other.

Any total order respecting the relation is consistent. To respect the relation means graphically that for every arrow the event at its origin is ordered before (left of) the event it points to. Every set of concurrent events (each displayed in one vertical row) can be ordered to any sequence. By permuting the concurrent events, all consistent total orders respecting the restricted DSMS causality relation can be constructed, i.e. the total order can start with either event $W(y)1$ of p1, $W(x)2$ of p2 or $send(m)$ of process p3.

Every line that could be drawn in Figure 3, separating the events of each process into two sets – the "past" to the left and the future to the right – without having an arrow beginning in the future and ending in the past, would display a consistent cut. If, additionally, the events in the past are totally ordered with respect to the relation, the cut defines

[1] That is, if every message sent is saved with the corresponding state so that messages currently in transit as to a given cut can be properly resent. This feature can be postulated w.l.o.g., which we will do for the remainder of this paper.

[2] This is actually more information than required for a global state.

a consistent global state consisting of the ordered events in the past, which also determines the memory content.

4 Conclusion

In this paper, we presented a formal basis for constructing consistent views on distributed computations on top of a DSM. Based on research work in the field mere message passing systems, we have formalized the relations that cover the causal dependencies in DSM systems. We proved that the conditions respected by the DSMS causality relations are necessary and sufficient for total event order to be consistent, and thus for a consistent observation of these systems. According to the research in the field of the verification of shared memory consistency models, we refined our relations to a more restrictive version in favor of efficient implementations: the restricted DSMS causality relation. This relation enables to construct a consistent total event order in acceptable time, based on additional information collected during the observation. In future work, efficient methods for this information collection must be developed. We have defined consistent global states and consistent cuts based on the restricted DSMS causality relation. The formal basis formed in this paper is the first step towards the realization of observing distributed DSM systems and analyzing behaviour and properties of these systems, which will lead to DSM systems more manageable, understandable, and fault-free than they are now.

Acknowledgments

We thank Prof. Dr. P.P. Spies and Dr. C. Rehn for their suggestions and valuable comments in discussions on our work.

References

[1] Özalp Babaoğlu, Keith Marzullo. Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. In Sape Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. Addison Wesley, 2nd edition, 1993.

[2] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. In *Communications of the ACM*, 21(4), pages 558–565, July 1978.

[3] Friedemann Mattern. Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. In *Journal of Parallel and Distributed Computing*, 18(4), pages 423–434, August 1993.

[4] Neeraj Mittal, Vijay K. Garg. On Detecting Global Predicates in Distributed Computations. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 3–10, April 2001.

[5] K. Mani Chandy, Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. In *ACM Transactions on Computer Systems*, vol. 3, no. 1, pages 63–75, February 1985.

[6] Jerry Fowler, Willy Zwaenepoel. Causal Distributed Breakpoints. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 134–141, 1990.

[7] Madalene Spezialetti, Phil Kearns. Efficient Distributed Snapshots. In *Proceedings of 6th International Conference on Distributed Computing Systems*, pages 382–388, 1986.

[8] S. Venkatesan. Message-optimal incremental snapshots. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 53–60, Newport Beach, CA, June 1989.

[9] Hon F. Li, T. Radhakrishnan, K. Venkatesh. Global State Detection in Non-FIFO Networks. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 364–370, September 1987.

[10] Ten H. Lai, Tao H. Yang. On Distributed Snapshots. In *Information Processing Letters*, 25(5), pp. 153–158, May 1987.

[11] Carroll Morgan. Global and Logical Time in Distributed Algorithms. In *Information Processing Letters*, 20(5), pages 189–194, May 1985.

[12] Barton P. Miller, Jong-Deok Choi. Breakpoints and Halting in Distributed Programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 141–150, June 1988.

[13] Dieter Haban, Wolfgang Weigel. Global Events and Global Breakpoints in Distributed Systems. In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, Volume II, pages 166–175, IEEE Computer Society, January 1988.

[14] Richard Koo, Sam Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. In *IEEE Transactions on Software Engineering*, SE 13(1), pp. 23–31, January 1987.

[15] Hon F. Li, Gabriel Girard. A Hierachy of View Consistencies and Exact Implementations. In *Proceedings of 1999 Workshop on Software Distributed Shared Memory*, Rhodes, pages 109–114, June 1999.

[16] Anne E. Condon and Alan J. Hu. Automatable verification of sequential consistency. In *Proceedings of the 13th annual ACM symposium on Parallel algorithms and architectures*, pages 113–121, 2001.

[17] Shaz Qadeer. Verifying Sequential Consistency on Shared-Memory Multiprocessors by Model Checking. In *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 8, pages 730–741, 2003.

[18] Phillip B. Gibbons and Ephraim Korach. Testing Shared Memories. In *SIAM J. Comput.*, vol. 26, no. 4, pages 1208–1244, 1997.

[19] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. In *IEEE Computer*, vol. 29, no. 12, pages 66–76, 1996.

[20] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. In *IEEE Transactions on Computers*, vol. 28, no. 9, pages 690–691, 1979.

[21] Michel Raynal. Sequential Consistency as Lazy Linearizability. In *Proceedings of the 14th annual ACM symposium on Parallel algorithms and architectures*, pages 151–152, 2002.