Realizing Consistent Event Ordering in Distributed Shared Memory Systems

Tobias Landes, Jörg Preißinger
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. Based on our previous theoretical work concerning consistent event ordering in systems featuring both message passing and distributed shared memory facilities, in the paper at hand we discuss the more practical issues, solutions, and results presenting themselves during the process of actually realizing and implementing the construction of consistent global views on such systems.

Keywords: *distributed system, distributed shared memory, consistency, observation, implementation*

1 Introduction

The transition from conventional, pseudo-parallel systems to distributed systems is characterized by the benefit of real parallelism on the one hand, and a significantly higher level of complexity on the other. While it is relatively straightforward to determine the current system state or the effects of the last operation in a pseudo-parallel system, in a distributed system, because of real parallelism, this is (in general) a non-trivial problem.

The lack of global time and the asynchronous parallel execution of concurrent instructions in different processes prevent the system's current state from being characterizable by a simple global glance on the execution. However, many tasks in controlling and managing distributed systems need to establish a global system state, e.g. monitoring, breakpointing, debugging, or automated management.

So what we need is a consistent global view on the system. "Consistent" means that what we see is a state that is meaningful in the sense that it has, or could have, occurred in the system execution, considering all the causal dependencies among the states of the individual execution activities. Such a consistent global view is generally sufficient as a base for the tasks mentioned above.

How to construct a consistent view on mere message passing systems has already been examined by Lamport and others [2, 3, 4]. In prior work [1] we have extended these considerations to a system model featuring a distributed shared memory and thus created a formal base for the implementation of these concepts. In the document at hand we explain and discuss the proceedings, problems, and practical experiences during the realization. We are not aware of any related work presenting an actual realization of the construction of a consistent global view on a system featuring distributed shared memory.

This document is structured as follows. In section 2 we briefly summarize the most relevant theoretical results of our previous work concerning consistent event ordering in distributed shared memory systems [1]. Section 3.1 generally describes the practical measures that must be taken to gather all the relevant information. In section 3.2 we briefly introduce the experimental system which served as the base for our implementation. Section 3.3 discusses how the consistent global view is to be obtained. Section 3.4 describes how the gathering of information is done in our particular system, and section 3.5 discusses the resulting performance issues. Section 3.6 illustrates the realization by means of an example execution, and section 4 summarizes the paper.

2 Consistent Event Ordering In Theory

Assembling a consistent view on a running system requires the comprehension of all relevant computational events and their mutual causal dependencies. This section briefly summarizes the theoretical conditions for the gathering of the required information, as we elaborated in detail in [1]. The granularity of the events taken into consideration generally depends on the application of the consistent view to be generated, but should allow for certain events which are crucial to the tracing of the causal dependencies among the events (i.e. send, receive, write, and read events). In the next subsection a description of our system model is given, along with all events of special interest mentioned above.

2.1 System Model

In our system model, a distributed computation consists of a finite set $P = \{p_1, p_2, \ldots, p_n\}$ of *n processes*. The processes communicate with each other in two ways. First, they can communicate by sending and receiving *messages*, which are only assumed to be delivered reliably and with a finite delay. The second way is established by a *distributed shared memory* (DSM) which allows for passive memory objects or addresses to be shared among the processes through the basic operations of reading and writing. We call systems with DSM *distributed shared memory systems* (DSMS).

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 chosen abstraction level and changes the *state* of the process.

2.2 Causal Dependencies

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 of events. Lamport [2] called this the "happened before" relation, and defined it as the transitive closure of the program order and the natural causal send-receive dependencies. In our model, due to the presence of DSM, there exist even more events that, in analogy to the send and receive events, establish dependencies thus extending the "happened before" relation and reducing the number of consistent total event orderings. These are the *read* and *write* events, which describe reading or writing access to a shared memory address. In [1] we therefore explained and defined the *DSMS causality relation* as follows:

Definition 2.1. 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. The *DSMS causality relation* $\stackrel{c}{\Rightarrow}$ is the smallest transitive order relation satisfying the following three conditions:[1]

(1) $\forall e_i, e_j \in E:$ if $e_i \rightarrow e_j$, then $e_i \stackrel{c}{\Rightarrow} e_j$.

(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 $e_i^x \stackrel{c}{\Rightarrow} e_j^y$.

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

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

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

If $e_i^x \stackrel{c}{\Rightarrow} e_j^y$, 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 (or, a memory value can not be read before it has been written). If $e_i^x \not\stackrel{c}{\Rightarrow} e_j^y$ and $e_j^y \not\stackrel{c}{\Rightarrow} e_i^x$, 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 $e_i^x \| e_j^y$. In [1] we proved that any total event order respecting the DSMS causality relation is consistent with the events' causal dependencies.

Since the constructive building of an event order respecting the above relation is np-complete, we defined a *restricted DSMS causality relation* $\stackrel{r}{\Rightarrow}$ based on write-order and read-mapping as proposed by Gibbons and Korach [14]. The *write-order* is the total order of all write events to the same memory location as occurred in an observed system execution. The *read-mapping* is a function that assigns to each read event the corresponding write event as actually observed during execution. Read-mapping and write-order enhance the general causality relation by providing valuable information that allows to construct a sequentially consistent total event order in $O(n\ log(n))$.

Definition 2.2. Let $\succ_{wo}$ totally order all write events $E_{W(x)}$ to the same location, respectively. Let $f_{rm}: E_{R(x)a} \mapsto E_{W(x)a} \cup \{\perp\}$ be a read-mapping function that maps every read event to its corresponding write event, or to $\perp$ if no write event accessed that location before. The *restricted DSMS causality relation* $\stackrel{r}{\Rightarrow}$ is the smallest relation satisfying the following five conditions:

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

$\forall e_i^x, e_j^y \in E:$ if e_i^x is a send event and e_j^y is the receive event of the same message, then $e_i^x \stackrel{r}{\Rightarrow} e_j^y$.

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

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

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

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

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

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 2.1). The benefit of the restricted relation is that it enables us to construct a consistent total order efficiently, because of additional information collected during system execution. For more details on these topics, see [1].

2.3 Event Lattice

A suitable tool for the visualization of events and their mutual dependencies as they occur in our system environment (see next section) is an *event lattice* as proposed in [10]. The lattice structure originates from the nesting of processes and the dependencies between the creation of a process through its "father" and its own starting event (initialization). An analogous dependency is

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

established through the fact that the termination event of a process has to occur before the destruction of the process through its "father". This synchronous termination concept establishes the complete nesting not only in process creation but in the dependencies and information flow as well, and distinguishes the event lattice from other event graphs. Additionally we add to the lattice the causal dependencies deriving from process cooperation, as described in section 2.2. An Example of an event lattice will be explained in section 3.6 and illustrated in figure 1.

3 Realizing a Consistent Event Ordering

Based on the formal definition concerning causal dependencies and consistency given in section 2, we now describe problems, experiences, and solutions occurring in the process of realization. Section 3.1 explains issues and possible solutions which are independent of the specific system environment used for implementation. Then we briefly describe MoDiS, our experimental system, and its concepts as far as they are relevant for the work presented in this paper. In the sections following thereafter we will discuss the actual realization and illustrate the results by means of an example.

3.1 System Independent Realization Issues

Some general problems in tracing and recording the events and dependencies described in section 2 present themselves independently of a given implementation environment. One fundamental problem is, of course, the effect on system performance. This issue will be discussed in section 3.5. In the following two subsections we will point out the problems in dynamic process systems and in observing concurrent memory accesses.

3.1.1 Dynamic Process System. Our system model assumes a finite set of processes as description of the activities in a computation. In the literature concerning consistent event ordering in message passing systems this set of processes is always assumed as being constant over time (e.g. in [2, 3, 4]), which in practice is actually an exception. In our realization we therefore consider a dynamically changing set of active processes as a subset of all processes present in the system. This renders usual mechanisms like vector clocks [5, 6, 7] unusable. Landes [8] proposes an extension of vector clocks which is meant to be suited for dynamic process systems and could be used for our implementation. But even this solution is not quite unproblematic. Let the system provide a service that processes incoming requests by spawning respective service processes. The set of active processes at a given time should not be very high, but the overall set of active and terminated processes would be potentially unlimited, which requires explicit measures to prevent the clock values to also grow unchecked. Landes [8] gives an extensive discussion on this issue and states that it is resolvable, but only with considerable effort. In section 3.3.1 we explain how this problem can, in our specific application of constructing the event lattice, be worked around by using direct event references.

3.1.2 Observing Memory Access. Recording the events described in section 2 and their mutual causal dependencies is complicated especially by the presence of a shared memory. In our system, the usual sending and receiving of messages is already implemented, in suitable libraries which can easily be extended by recording mechanisms. The same approach is used, in some systems with low DSM usage, for the shared memory. But in MoDiS, the shared memory access is partly based on direct machine instructions. The distributed shared memory is mapped directly into the virtual memory of a process by using the page fault mechanism. There it remains as long as it is valid. So, after the initial access, any further access can not be distinguished from a local memory access any more. Furthermore, it must be recognized whether an access is a reading or writing one, given that these have different causal dependencies, as explained in section 2.2.

We examined several possible approaches to recording DSM access events. One of these is based on appropriate hardware support. Standard ix86 computers feature hardware debug registers which are able to trigger controlled exception handling whenever a memory address is accessed. However, this feature is not sufficient as a solution because there are only four such registers each of which can monitor an address range of 4 Byte. Clearly, this would cut down the DSM way too much.

Another approach can be derived from debugging techniques, as they are used, for example, by the gdb (GNU Project debugger). Software watchpoints can be used to monitor memory access. However, this requires the execution of a process in single step mode and drops performance by factor 100, which is totally unacceptable, at least for our purpose.

The only really satisfying solution is modifying the compiler. The drawback, regarding the transferability to other systems and applications, is the necessity to recompile all applications for the distributed system using the modified compiler. With our experimental system MoDiS, however, this is no problem at all, due to the language based approach of the system itself. During the compiler analysis of the high-level programming language, reading and writing memory references can be additionally examined. At this point, the compiler can generate additional code for memory access event recording. So the analysis can be performed statically, and only the necessary recording operations have to be executed in runtime, which minimizes the performance loss.

3.2 Implementation System: MoDiS

In this section we briefly describe a few relevant aspects of the experimental system MoDiS, which is the

basis and environment for our implementation of consistent event recording.

MoDiS (**M**odel **o**riented **Di**stributed **S**ystems, developed at the chair for operating systems and system architecture of the Munich Technical University) is best characterized as a language-based top-down driven approach to developing distributed systems. The instructions defining the application are specified in the object-based high-level programming language INSEL (**In**tegration and **S**eparation Supporting **E**xperimental **L**anguage). MoDiS pursues a single system approach: compiler, runtime environment, DSM manager, and communicator are part of the system, as is operating system functionality. The gcc (GNU Compiler Collection) based compiler gic (GNU INSEL compiler, [13]) transforms the abstract specification into an executable program containing both application and management components. All transformation mechanisms and all information gathered during the transformation are part of the system. This concept ensures high availability of information and thus supports automated management for application oriented usage of the distributed hardware resources.

A detailed explanation of the MoDiS concepts can be found in [11] and [12]. Crucial to the work presented in this paper are the integrated INSEL compiler gic, the communicator, and the DSM manager, as they could be extended to suit our realization of consistent event recording.

3.3 Constructing the Event Lattice

Our purpose is to gradually visualize the progress of the computation in execution using an event lattice (see 2.3). During runtime, this lattice is generated step-by-step from the recorded events. Therefore, the graph consists of events that have already occurred and directed edges between them. An edge from event e to event e' means $e \stackrel{\Gamma}{\Rightarrow} e'$. To ensure the consistency of such a partly constructed graph G one must be careful to only add events e' to the graph whose dependencies are fulfilled, i.e. every (transitive) predecessor event is already part of the graph: $e' \in G$ only if $\forall e | e \stackrel{\Gamma}{\Rightarrow} e' : e \in G$. The necessary edges have to be added accordingly.

3.3.1 Vector Clocks vs. Direct-Dependence. The central task in constructing consistent views in distributed concurrent systems is to trace the causal dependencies among the events, as specified by the relations given in section 2.2. We consider two different approaches to achieve this: The dependencies can be traced implicitly using logical clocks, i.e. dynamic vector clocks as mentioned in section 3.1.1, or explicitly by direct references to causal predecessor events.

A logical clock adjusts the incrementing rules of the clock value to the causal dependencies. Because, for example, the receive event of a message is always dependent on the send event of the same message, a vector time value of the send event is piggybacked on each message and the receive event is assigned a higher value. In this way it is ensured that an order over the clock values implicitly reflects the causal dependencies. An event e with a time value greater than that of an event e' is causally dependent on e' (see, for example, [4]). This logical time can be used as base for the construction of the event lattice. For each entry x in the vector time stamp of an event e it has to be checked whether the event of the respective process with the local time stamp x is already part of the graph. If this is the case for all vector entries, e can be added to the graph.

The other option is to gradually record direct (non-transitive) dependencies. If we assume that every event e in the system has a unique identifier i_e, then we are able to record with each event a list of its direct causal predecessors. Since we assume a granularity such that each event may send or receive at most one message, the length of this list may only vary between 1 and 2. Let e be a send event with identifier i_e. With the corresponding receive event e' we only have to record the identifier i_e in order to be able to trace the direct dependence of e' on e. The unique event IDs can be realized as a combination of a node identifier, a process identifier, and a process-local event counter. The size in bytes of these event IDs may vary with the size of the system but is constant in any particular system. The event lattice construction based on events marked in this way can be performed as follows: For each recorded event e it is checked whether all the (1 or 2) entries in its direct dependency list reference events that are already part of the graph. In this case e can be added to the graph without harming consistency. Otherwise the same check is performed recursively for the list entries.

Comparing the two options one has to consider two different aspects. First, the space needed for the event recording and, second, the suitability for the construction of the event lattice as our consistent view on the system. The great advantage of vector clocks is the fact that each time stamp contains information not only about direct but also about indirect (transitive) dependencies. Given two events and their time values one can easily and definitely decide whether they are concurrent or dependent (and in which way). This is not possible with the direct-dependence method. A path search has to be performed, in both directions, to decide whether one can be reached from the other or not. However, if the events are not to be compared in general but to be used for the particular purpose of constructing the event lattice during runtime (or similar applications), then comparing arbitrary events is not necessary and the great disadvantage opposite vector clocks does not matter.

The space required to record direct dependencies is constant, whereas vector time stamps grow with the number of processes created in the system. A garbage collection as proposed in [8] is expensive and reduces the advantage of vector clocks as mentioned above, since only events remain comparable between which no garbage collection has been carried out.

For our purpose, which is to construct a gradually growing event lattice representing the progress of the

computation, the direct-dependence method is clearly to be preferred. We do not need the informational advantage (transitivity) of the clock based approach and can make use of the superior efficiency of tracing only direct (non-transitive) dependencies.

3.4 Collecting Necessary Information

Prerequisite to the construction method explained above are the events of the running system, marked with the IDs of their direct causal predecessors. The following section describe how to collect this information with respect to the event dependencies as given in by the restricted DSMS causality relation. All events relevant for process synchronization are recorded, which are, in MoDiS, creation and destruction of child processes, message passing, and distributed shared memory access. Dependent on the specific use meant to be made of the generated event lattice, one could just as well design and implement events of finer granularity. However, this would induce considerably more effort since more events would have to be recorded. For our purpose this would not be justified.

3.4.1 Process Order. Process order is captured implicitly for all recorded events, because the unique event identifier contains, amongst others, a scalar event counter on a per-process base (see 3.3.1).

3.4.2 Message Passing System. Capturing the send-receive dependency is implemented by piggybacking the identifier i_e of the send event on the message. The receiver registers the receive event along with i_e as one of its two direct dependencies. Both of these measures can be taken by the MoDiS communicator module. MoDiS supports a special case of message passing, which is the so-called operation oriented rendezvous concept. The dependencies deriving from this concept are explained by means of an example in section 3.6.

3.4.3 Distributed Shared Memory. In section 3.1 we explained why, at least for our purpose, the only reasonable way to monitor memory access is to have the compiler produce additional code. The INSEL compiler gic analyzes every memory access and generates code when it detects access to DSM as occurring in INSEL instructions like, for example, variable assignments, composite expressions, or function calls. The read-write dependencies described in section 2.2 are realized using system-unique object identifiers. Each object in the DSM is assigned such an ID, which is, in practice, the virtual address in the shared address space along with a scalar counter. Read and write access are recorded by the runtime environment. The access event is marked with the object ID and the incremented scalar time stamp of the access. By means of mutual exclusion we ensure the atomicity of the memory access and its recording. The size of the memory objects depends on the memory implementation and language concepts. In MoDiS there are no integrated synchronization mechanisms for shared memory objects (i.e. memory object access is interruptible), so we have to regard single memory addresses as objects. In systems featuring access synchronization, any number of memory references might be modeled as a single access event, as long as they can be performed atomically.

The recorded events are combined to the event lattice as explained in section 3.3. The following sections give some performance considerations and an illustrating example.

3.5 Performance

Monitoring a running system always induces some amount of performance loss. This can generally be justified by (at least) two different arguments.

First, one can decide to use the system monitoring only in debugging mode where the performance loss is not relevant. However, as Schrödinger showed in quantum physics by means of his famous thought experiment with the cat [15], that an observed system may act differently from an unobserved one. So it may be that not all errors occurring in the productive system show in the observed debugging mode (or vice versa).

Second, there are many applications in which performance plays a minor role in comparison to other requirements, e.g. availability, security, or reliability. Not only in systems controlling dangerous facilities can a system fault be disastrous, but also in economical fields. Observing such systems can be a very reasonable measure in fulfilling quality requirements, even at the expense of system performance.

Processing the recorded events by constructing the event lattice and using it for analysis and visualization of the system's execution can be done in a process external to the system on a machine which is not part of the system, so it is of no relevance apart from a certain increase in network load, which might suit a given application better than increased system load. What remains is the gathering of information which is actually significant for the performance penalties.

In the cases of process creation and message passing the recording of events and their IDs can be ignored, given the very small overhead which is only a fraction of the time needed to create a process or send a message over a standard network.

The only really significant contribution to the overhead arises from the memory observation. Every DSM access requires between 4 and 6 additional memory accesses, and an increment instruction. A (very generous) upper bound for the performance loss can therefore be given as a factor of six as compared to the normal system execution. However, as proven in practice, the behaviour is significantly better than that because, first, only a part of all program instructions are actually memory operations, and, second, only a part of all memory operations reference the DSM. In order to be able to describe the performance penalties more precisely, measurements

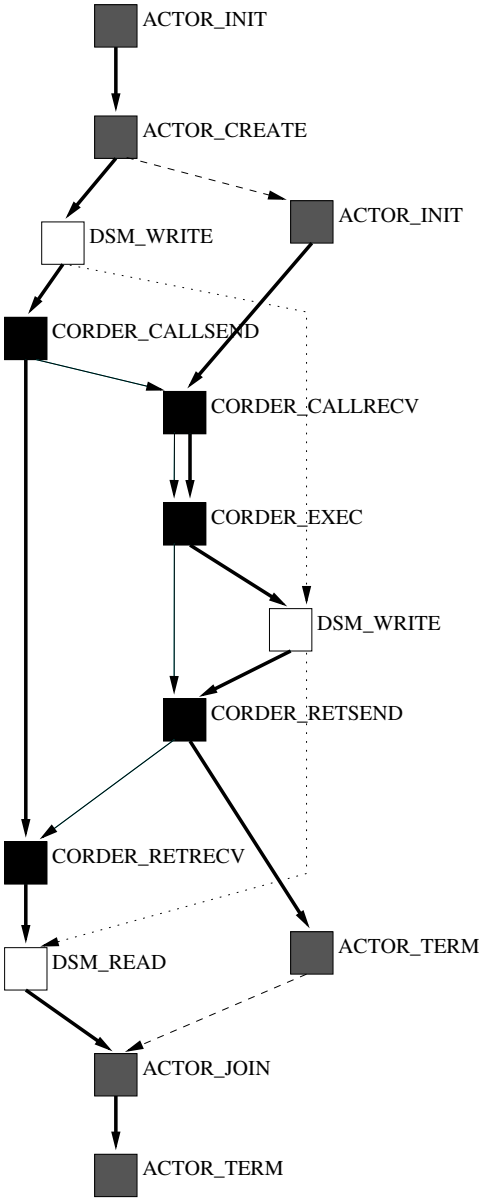

Figure 1: Constructed Event Lattice

and comparisons of representative applications with and without observation will yet have to be performed.

3.6 Example

Figure 1 shows an event lattice of a simple MoDiS system, constructed as described in this paper. The events are visualized as squares, their mutual dependencies as directed edges. Memory access events are colored white, process creation and termination events grey, and rendezvous events (a special case of message passing) black.

The operation oriented rendezvous, called C-order, is a message based process synchronization mechanism in MoDiS. First, the caller has to execute a call event, which is blocking. The callee has to perform an event for the receiving of the message. Then the callee executes the requested function and returns the result to the caller, who is then allowed to continue.

Figure 1 shows an initial process, who creates another process just after its own ACTOR_INIT event. The first event of the new process is, again, ACTOR_INIT, but causally dependent on the father's ACTOR_CREATE, the dependencies resulting of process creation and termination are shown as dashed edges. The respective process order, i.e. the concurrent execution threads are marked as slightly thicker black edges. The father process writes to the distributed shared memory and then synchronizes with its child process using a C-order. The C-order is initiated by an event called CORDER_CALLSEND, on which the event CORDER_CALLRECV is dependent. During the execution of the called function the caller is blocked, which is visible in the figure through the dependencies of the event CORDER_RETRECV on both CORDER_CALLSEND and CORDER_RETSEND. The C-order dependencies are shown as thin black edges. The function called in this rendezvous performs a write access on the memory location that has previously been accessed by the caller. The write-order dependencies (see section 2) are drawn as dotted edges. At the end the father process captures its terminated child in the event ACTOR_JOIN and then terminates itself.

As can be observed, the event lattice constructed using the methods presented in this paper, clearly shows all and causal dependencies (as demanded in section 2). Thus one can, for example, easily decide which events are concurrent, and use this information for automated system management, debugging or other purposes.

4 Conclusion

In this paper, we presented a realization of constructing consistent views on distributed computations featuring both message passing and DSM facilities. This requires that events along with their mutual causal dependencies as described by relations given in [1] be recorded and processed. We examined different methods for capturing memory access events and explained their respective issues. Furthermore we compared, with respect to dynamic process systems, vector-based clocks and a direct-dependence approach, discussing the advantages and disadvantages of each.

The actual realization of the event recording and event lattice construction have been explained in the context of the experimental system MoDiS and illustrated by an example.

The captured event lattices are the basis for two directions in future research. On the one hand, we have to further examine the event capturing itself with respect to granularity and performance. On the other hand, it must be explored in how far the view on system gained through these methods is suited as a tool to analyze and control distributed systems, in order to render them more manageable, fault-free, and secure.

Acknowledgments

We thank Prof. Dr. Peter Paul Spies and Dr. Christian Rehn for their suggestions and valuable comments in discussions on our work. Further we thank Sebastian Haas for the practical work he did within the scope of his diploma thesis [9].

References

[1] Jörg Preißinger and Tobias Landes. Fundamentals for Consistent Event Ordering in Distributed Shared Memory Systems. In Hamid R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA '05*, pages 890–896, Las Vegas, NV, 2005.

[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] 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.

[4] 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.

[5] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In M. Cosnard et al., editor, *Proceedings of the Workshop on Parallel and Distributed Algorithms*, pages 215–226, Elsevier Science Publishers B.V., North-Holland, 1989.

[6] Colin Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *Proceedings of the 11th Australian Computer Science Conference*, pages 55–66, February 1988.

[7] Colin Fidge. Logical Time in Distributed Computer Systems. In *Computer*, 24(8), pages 28–33, August 1991.

[8] Tobias Landes. Dynamic Vector Clocks for Consistent Ordering of Events in Dynamic Distributed Applications. *Document submitted for publication to PDPTA '06*, written 2005.

[9] Sebastian Haas. Erfassung konsistenter Sichten von verteilten, nebenläufigen Systemen (german only). *Diploma Thesis, Technische Universität München, Institut für Informatik*, 2005.

[10] Peter P. Spies. Ereignisverbände - ein flexibles Beschreibungsinstrumentarium für die Entwicklung verteilter Systeme (german only). In *FBT'98-Fachgespräch*, Cottbus, 1998.

[11] Peter P. Spies et al. Concepts for the construction of distributed systems (german only). In *SFB-Bericht 342/09/96 A TUM-I9618*, technical report, Technische Universität München, 1996.

[12] C. Eckert and M. Pizka. Improving resource management in distributed systems using language-level structuring concepts. In *Journal of Supercomputing*, 13(1), pages 33–55, January 1999.

[13] Markus Pizka. Design and implementation of the gnu insel-compiler (gic). In *SFB-Bericht 342/09/97 A TUM-I9713*, technical report, Technische Universität München, 1997.

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

[15] E. Schrödinger. Die gegenwärtige Situation in der Quantenmechanik. In *Naturwissenschaften 23*, pp. 807–812; 823–828; 844–849 (1935).

[16] 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.

[17] 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.