A Non–Blocking Lightweight Implementation of Causal Order Message Delivery

Friedemann Mattern and Stefan Fünfrocken

Technical Report No. TR-VS-95-01

March 1995

Department of Computer Science, Technical University of Darmstadt,
Alexanderstr. 10, D 64283 Darmstadt, Germany
Email: {mattern, fuenf}@isa.informatik.th-darmstadt.de

A Non–Blocking Lightweight Implementation of Causal Order Message Delivery

Friedemann Mattern and Stefan Fünfrocken

Department of Computer Science, Technical University of Darmstadt,
Alexanderstr. 10, D 64283 Darmstadt, Germany
Email: {mattern, fuenf}@isa.informatik.th-darmstadt.de

Abstract. This paper presents an algorithm to implement point–to–point causal order message delivery in distributed systems which does not force the sender to wait and which does not piggyback control information (such as timestamps) on messages. The algorithm is based on a message transmission protocol using low–level acknowledgements between FIFO buffers. We show that on the one hand causal order can easily and efficiently be realized in that way, but that on the other hand the loss of knowledge—induced by not using dependency matrices as in previously known protocols—leads to a slight restriction with respect to the applicability of the new protocol. The advantages of our scheme, however, are obvious because it is non–blocking and the piggybacking of huge control information on messages is avoided. Furthermore, the new algorithm is easily implementable since many distributed systems and low–level transmission protocols already provide message buffers and explicit or implicit message acknowledgements.

1 Causal Order

In distributed systems, efficient message passing with appropriate semantics is of prime importance, since sending and receiving messages is the only way for processes to cooperate, to exchange data, and to gain knowledge about the state of other system components. For many applications and operating system tasks it is mandatory that messages respect the potential causality relation among events. Operationally, this means that direct communication between two processes should always be faster than indirect communication where messages are routed via intermediate processes. A computation exhibiting this "triangle inequation", which is a stronger property than the more local FIFO property along single communication channels, is called a *causally ordered computation* [CMT94]. Protocols which guarantee this behavior are implementations of the so-called *causal order message delivery* property. The use of causal order in message delivery was introduced in [BJ87], where also protocols for broadcasts are presented. Specific algorithms for point–to–point communications were later given in [SES89] and [RST91].

The causal order property is useful, among other things[1], for the following

[1] Applications of causal *broadcasts* may be found in the collection of papers on the Isis System [BR94].

purposes:

- *Consistent snapshot generation*: compute the global state of a distributed system in such a way that it does not show an "effect" (i.e., the receipt of a message) without showing its "cause" (i.e., its sending) [AB92, AV94].
- *Observing distributed systems*: display a sequence of events of a distributed computation which respects their causal dependencies (e.g., where the receipt of a message is never observed before its sending) [SM94].
- *Fair distributed resource allocation management*: requests for a resource should be satisfied in the (causal) order they were issued by the processes [KK89].
- *Garbage collection in distributed systems*: Here, reference counters must be updated in a causally consistent way, otherwise non-garbage objects may be identified as being garbage [TM93].

In order to precisely define the causal order property and to be able to prove the correctness of the algorithm described in later sections, we need a more abstract notion of a *distributed computation*. Such a computation is formed by a set of processes, each generating a sequence of *events* (i.e., abstractions of actions performed by the processes). To depict a computation, one may use *space-time diagrams* consisting of lines and dots (each line representing a process and each dot standing for an event), where messages with their pairwise corresponding send-receive events are represented by arrows (see, e.g., Fig. 4).

In a canonical way, the events of a process are *totally ordered* since processes do sequentially execute one event at a time. We denote this order by $\prec_i$ for each process i. If $a \prec_i b$, then event b happens later than event a on the same process. Messages (i.e., send events and their corresponding receive events) introduce additional dependencies between the sender and the receiver since the sending of a message must always precede its receipt. Furthermore, indirect dependencies among events are caused by transitivity (e.g., by chains of messages). This gives rise to the following definition of a general *causality relation*, denoted by '$\prec$', among all events of a distributed computation (originally called "happened before" by Lamport [Lam78]):

Definition: *The* causality relation $\prec$ *is the smallest transitive relation that satisfies the following two properties for any two events* a, b*:*

- *If* $a \prec_i b$*, then* $a \prec b$*.*
- *If* a*,* b *are corresponding send and receive events, then* $a \prec b$*.* $\Box$

If $a \prec b$, then event b causally depends on a in the sense that b is not executed if the computation blocks at a. Note that $\prec$ is a *partial* order: there may exist events not ordered by $\prec$, which are said to be *concurrent*. To specify that two events a, b happen at the same process, we write $a \sim b$.

With the help of $\prec$ we can now exactly define causal order message delivery (i.e., what it means that no message between two processes is overtaken by a chain of other messages).

Definition: *A distributed computation is called a* causally ordered computation*, if for all pairs (s,r) and (s',r') of corresponding send-receive events*

$$(r \sim r') \wedge (s \prec s') \ \Rightarrow\ r \prec r'. \qquad \square$$

Intuitively, this means that all messages delivered to a process respect the causality relation $\prec$. Note that the causal order property trivially implies the FIFO property $(s \sim s') \wedge (r \sim r') \wedge (s \prec s') \ \Rightarrow\ r \prec r'$ as a special case. In [CMT94] it is shown that causally ordered computations fall between synchronous computations (i.e., computations where communication is always synchronous) and FIFO computations (i.e., computations where communication between two processes respects the FIFO property). They preserve the inherent causal order property of synchronous computations, but exhibit enough properties of general asynchronous computations to overcome the phenomenon of communication deadlocks often found in synchronous computations. In fact, non–blocking implementations of causal order are possible as will be shown in the following sections.

2 Earlier Realizations of Causal Order

To implement causal order, one has to consider the communication characteristics of the underlying message transport system. Upon this system, a suitable protocol is then superimposed to ensure that messages are always delivered in accordance to causal order. A related, although much simpler problem is the realization of the FIFO property. One way to implement FIFO on asynchronous systems is to resort to synchronous communication and *block* the sender until an acknowledgement from the destination process is received. This works because computations with synchronous communications are always causally ordered [CMT94]. A better and also well-known solution to guarantee FIFO message delivery consists in using *sequence numbers* on each communication channel. The delivery of messages with higher sequence numbers, which arrive too early at the destination process, can be delayed until all messages with lower sequence numbers have arrived. The causal order delivery protocols described in this and the next section can, in some sense, be viewed as generalizations of these two principles.

A first implementation of causal *broadcasts* was described by Birman and Joseph [BJ87] in the context of the Isis system [BR94]. Isis originally realized causal broadcasts by conceptually piggybacking all causally preceding broadcast messages on each message [BJ87], but then applying some optimizations which trim the piggybacked causal history in an effective way. More recent versions of Isis are based on vector time, which represents a substantial improvement because of the reduced overhead [BSS91]. For causal order delivery of *point–to–point* messages, one could of course also piggyback the whole causal history on each message. However, in that case a so–called conservative solution is used in the Isis system, where the sender is blocked until reception of the message is acknowledged [BSS91]. (The same "synchronous" implementation is also used for broadcasts in situations where piggybacking is too costly.)

Kearns et al. [KK89] presented a protocol to ensure point–to–point causal order message delivery in a distributed system with a single server, to which messages from all other processes (so-called clients) are delivered in causal order. The protocol uses message-count-vectors which are located at each process and which are piggybacked on each message. A *client* j receiving a message with attached vector V_k from another client k updates its vector V_j by setting it to the componentwise maximum of V_k and V_j. Before a client i issues a message to the server, it increments its component $V_i[i]$ of its message-count-vector V_i and attaches the vector to the message. When the *server* receives a message m from process i, it first compares its own message-count-vector V_s with V_m, the vector accompanying message m. If $V_s[j] < V_m[j]$ for some $j \neq i$, then there is a message still missing from another client (on which message m causally depends), and message m is delayed until the missing message has arrived. Otherwise the server consumes the message after updating its own message-count-vector to the componentwise maximum of the vector attached to the message and its own vector.

Note that $V_i[j]$ represents the "knowledge" process i has about the number of messages sent by process j to the server so far. Taking the componentwise maximum when receiving a message from another client thus incorporates indirect knowledge, reflecting the fact that causality is transitive. This is the same principle the so-called *vector timestamps* are based on [Fid88, Mat88], and in fact message-count-vectors are only a slight variant of vector timestamps. Delaying the receipt of a message until all causally preceding messages have arrived is a canonical extension of the sequence number method to guarantee FIFO delivery.

If not only messages towards a single server, but all messages at all processes should be delivered in causal order, then the scheme just mentioned can be generalized by considering each process to be such a server. Such a general point–to–point causal order delivery protocol would use n vectors of length n (where n is the total number of processes) at each process and in each message.

In fact, the implementation of causal order presented by Schiper at al. [SES89], which was found independently of Kearn's more restrictive solution, is based on this idea. A slight variant which uses *integer matrices* of size $n \times n$ (instead of at most n vectors of length n in each message) was later given by Raynal, Schiper, and Toueg [RST91]. In this protocol, each process i has such a matrix M_i. When a process i sends a message to process j, it increments $M_i[i,j]$ and attaches M_i to the message. Whenever a process j receives a message together with its attached matrix M, M_j is updated to the componentwise maximum of M and M_j. By that, the matrix of a message encodes the knowledge of all send events (i.e., other messages) it causally depends on. Therefore, a message from process i with matrix M that arrives "out of causal order" can easily be detected by the receiving process j: The message is simply not delivered until $\forall k \neq i : M[k,j] \leq M_j[k,j] \;\wedge\; M[i,j] = M_j[i,j] + 1$, which is guaranteed to happen eventually if messages are not lost. Messages which have to be delayed according to this condition remain in a buffer at the receiver's site until they can be delivered.

The solutions for point–to–point causal order message delivery presented so far need $O(n^2)$ space in all messages and processes. Although some optimizations are possible (e.g., sending only delta-increments or omitting certain matrix components if the communication topology has some known properties), the space overhead seems to be prohibitive for most large systems. Our aim was therefore to find a solution which is more space-efficient. The idea is that since the class of causally ordered computations lies between the class of synchronous computations and the class of FIFO computations, it should be possible to arrive at causal order delivery protocols by weakening protocols that guarantee synchronous message passing in contrast to the classical approach sketched above of strengthening protocols that guarantee the FIFO property.

In fact, protocols which implement synchronous message transmission on top of asynchronous systems (such as the above mentioned "wait and acknowledge" scheme) do guarantee causal order message delivery in a natural way because messages which are "virtually instantaneous" cannot be overtaken by a chain of other messages [CMT94]. In general, however, one would probably not be happy with such a protocol since an application which executes without problems using a traditional non–blocking causal order delivery protocol might deadlock when running on such a simulated synchronous system. Hence, we tried to make the usual synchronous message passing protocol more efficient and less prone to blocking situations by introducing buffers, but limiting the degree of asynchrony by using acknowledgements in an appropriate way. Since many distributed systems use message buffers anyway and message transport layers often use some low-level acknowledgement scheme to guarantee reliable transmissions, such a protocol should be easy to implement.

3 A Non–Blocking Low Overhead Protocol

In this section we first give an informal view of our protocol. Then we describe it in detail and provide a proof that it guarantees causal order.

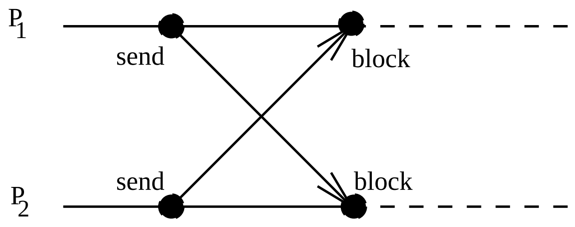

Fig. 1. Deadlock in a Synchronous System

3.1 Informal Description

In synchronous systems, deadlocks occur in situations like the one depicted in Fig. 1, where each process is waiting for the delivery of its own message before accepting the other message. Hence both processes cannot proceed in their computation. If, however, we equip each process with an input buffer and an output buffer, the buffers might synchronize their actions on behalf of the processes and the processes are free to proceed with their computation. Since we want to guarantee at least the FIFO property (in fact, we have to guarantee a stronger property in order to realize causal order message delivery), we require all message buffers to be implemented as FIFO queues. Communication between a sender's output buffer and the receiver's input buffer should remain synchronous. On top of an asynchronous system, this can be realized as usually with a handshake protocol using acknowledgement messages.

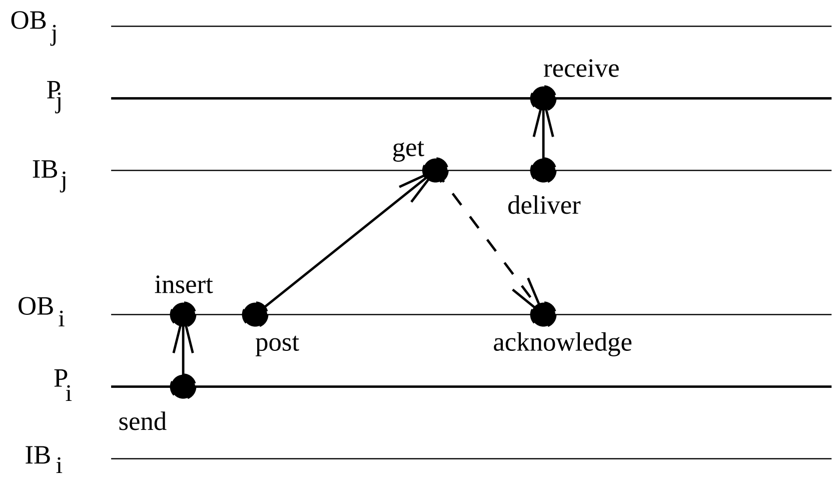

Fig. 2. A Single Message-Passing Operation

The causal order delivery protocol now works as follows (see Fig. 2): To send a message, the sender process P_i hands the message to its dedicated output buffer OB_i and continues with its computation (i.e., the send is non–blocking). If the output buffer is empty, the message is immediately transmitted to the unique input buffer IB_j of the receiver P_j, then the output buffer waits for an acknowledgement from the receiver's input buffer. If, however, the output buffer is not empty, the message is enqueued at the output buffer. The crucial point is that an output buffer may transmit its next pending message only if it has received an acknowledgement for its previously sent message. Hence, the transmission of a message (by an output buffer) is sometimes delayed for a short time, but (assuming large enough input buffers) no deadlock is possible. Figure 3 shows this for a situation where a purely synchronous system would deadlock (see Fig.1).

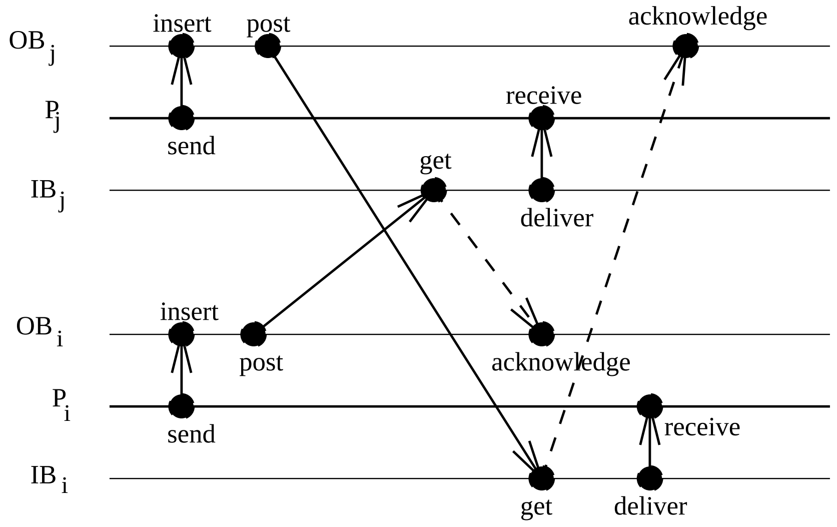

Fig. 3. No Deadlock with the Buffer Protocol

When a message arrives at the receiver's input buffer, the buffer immediately sends back an acknowledgement to the sender's output buffer. (Note that such acknowledgements are implicit if communication between buffers is synchronous.) When the receiver process encounters a receive operation, it retrieves the first message from its input buffer (which is the oldest message) if the buffer is not empty, otherwise it waits for a message to arrive. Since we use FIFO buffers, messages are received by the process in exactly the same order they are inserted into the input buffer.

At first sight it might be surprising that "non–blocking synchronous communication" (i.e., synchronous communication with input/output buffers) yields causal order. (Recall that only "pure" synchronous communication trivially implements causal order, and that such pure schemes are occasionally used for causal message delivery in the Isis system, as mentioned above.) Informally, however, it is easy to see that causal order is guaranteed by our protocol: First note that a message m' sent after a message m by some process cannot overtake m since it is inserted into the output buffer *after* m and therefore transmitted only after the insertion of m into the receiver's input buffer has been acknowledged. Then observe that this is also true for a chain of messages of which m' is the first message—m' is only released when m already "safely" arrived at the receiver's FIFO buffer. Hence, there are also no *indirect* overtakings of messages, a property which is characteristic for causally ordered computations. We will give a formal proof at the end of this section; the next subsection prepares this by describing the "buffer protocol" more precisely.

3.2 The Protocol

We consider a system of n processes $P_1, \ldots, P_n$ which communicate by reliable messages. Each process P_i has its own unique input buffer IB_i and output buffer OB_i, both realized as FIFO queues. Process and buffer behavior is specified in the following operational way:

```
PROCESS P_i:
    send message m to P_j:
        call $OB_i$ to insert $m$ and resume computation
    receive a message:
        call $IB_i$ to deliver its next message (and wait for it)

INPUT BUFFER IB_i:
    deliver request by P_i:
        wait until buffer is not empty;
        dequeue the next message from the buffer and deliver it to $P_i$
    getting a message from OB_j:
        send an acknowledgement to $OB_j$;
        enqueue message in buffer

OUTPUT BUFFER OB_i:
    insert-request for message m by P_i:
        enqueue $m$ in buffer
    posting messages:
        loop forever
            wait until buffer is not empty;
            dequeue the next message and transmit it to $IB_j$;
            wait for an acknowledgement from $IB_j$
```

Enqueues to a buffer by a *get* or *insert* operation should always be possible, even when a *deliver* or *post* operation is delayed by an empty buffer[2]. Note that we do not stipulate that the input buffers or output buffers are separate control threads or autonomous active objects—depending on the implementation, their actions might be carried out by the processes themselves (e.g., using an interrupt-driven control scheme).

Consider a single message-passing operation from process P_i to process P_j, as shown in Fig. 2. To transmit a message from one process to another, two

[2] Hence we assume unlimited buffer capacity. Clearly, too small buffers in a concrete implementation may cause enqueues to block and thus give rise to deadlocks.

application-level operations are executed: a *send operation* by the sender and a *receive operation* by the receiver. In our protocol, these two operations induce certain *events* which are used to describe its behavior and to prove its correctness:

send: process P_i initiates the sending of message m by calling its output buffer.
insert: message m is inserted into the output buffer, now waiting to be transmitted to the input buffer of destination process P_j.
post: message m is now actually transmitted to P_j. Further sending from this output buffer (to any input buffer) is inhibited until an acknowledgement arrives.
get: m is inserted into the input buffer of P_j, which sends back an acknowledgement to the output buffer of P_i.
deliver: the message is delivered to P_j.
receive: P_j receives message m and processes it.
acknowledge: the acknowledgement, sent by the input buffer of P_j, arrives at P_i's output buffer.

Each message transmission therefore generates a *send*, *insert*, *post*, and *acknowledge* event at the sender's site, and a *get*, *receive*, and *deliver* event at the receiver's site. Often, *send* and *insert* (as well as *deliver* and *receive*) can be considered to be a single event, since the two operations are executed in a synchronous way on the same site. Note that after *send/insert* the sender is free to continue with its execution, in particular it may send other messages before the previous message is received (or even posted).

3.3 Proof of the Causal Order Property

For ease of notation we establish:[3]

P_i	process i ($i = 1, \ldots, n$)
OB_i	output buffer of P_i
IB_i	input buffer of P_i
s_i	send event at P_i (i.e., start of P_i's send operation)
i_i	insert event at OB_i
p_i	post event at OB_i
a_i	acknowledge event at OB_i
g_i	get event at IB_i
d_i	deliver event at IB_i
r_i	receive event at P_i (i.e., completion of P_i's receive operation)
x_i^m	$x \in \{s, i, p, a, g, d, r\}$, event related to message m at location i (i.e., at P_i, OB_i, or IB_i).

To prove that our protocol is *safe*, we have to show that all possible computations generated by the protocol are causally ordered.

[3] We omit the index from an event if its location is not relevant or if it is clear from the context.

Proposition 1. *For all send events s^m, $s^{m'}$ and corresponding receive events r^m, $r^{m'}$ of any computation driven by the protocol, one has*

$$s^m \prec s^{m'} \;\wedge\; r^m \sim r^{m'} \;\Rightarrow\; r^m \prec r^{m'}.$$

We first show three lemmas which hold for all such computations. With their help we then prove the proposition.

Lemma 2. $p_i^x \prec p_i^y \;\Rightarrow\; g^x \prec g^y$.

Proof. Output buffer OB_i can only post a later message y when the insertion of the earlier message x into the appropriate input buffer is acknowledged. Hence, $p_i^x \prec g^x \prec a_i^x \prec p_i^y$. From $p_i^y \prec g^y$, it follows by transitivity that $g^x \prec g^y$. □

Lemma 3. $s_i^x \prec s_i^y \;\Rightarrow\; g^x \prec g^y$.

Proof. Because message x is inserted into the output buffer OB_i before message y, we have $i_i^x \prec i_i^y$. Because OB_i is a FIFO buffer, $p_i^x \prec p_i^y$ follows. Lemma 2 then yields $g^x \prec g^y$. □

Lemma 4. $g_j^x \prec g_j^y \;\Rightarrow\; r_j^x \prec r_j^y$.

Proof. Input buffer IB_j is a FIFO buffer, hence messages are delivered (and received) in the same order they are inserted. □

The *proof of Proposition 1* is now divided into two cases.

case (a) $\neg(s^m \sim s^{m'})$:

> *Proof.* Because $s^m \prec s^{m'}$ by hypothesis, there must exist a causal chain of events $s_i^m \prec \ldots \prec s_i^z \prec g^z \prec \ldots \prec s^{m'}$ with a first message z. From Lemma 3, $g^m \prec g^z \prec \ldots \prec s^{m'}$ follows. Because $s^{m'} \prec g^{m'}$, we have $g^m \prec \ldots \prec g^{m'}$ by transitivity. Because messages m and m' are received by the same process (i.e., $r^m \sim r^{m'}$), Lemma 4 applies, yielding $r^m \prec r^{m'}$.

case (b) $(s^m \sim s^{m'})$:

> *Proof.* Because the sender and the receiver of the two messages are identical, Lemma 3 and Lemma 4 directly apply. By transitivity we get the desired result $r^m \prec r^{m'}$. Note that this is the FIFO property, a degenerated case of the general causal order property. □

Informally, the *liveness* property (i.e., the fact that each message handed to the output buffer by a send operation is eventually inserted into the input buffer at the receiver's site by the protocol) is also easy to see: Recall that we assume reliable message transmission and unlimited buffer capacity. Furthermore, we assume that the scheduling of events is fair. (This means that whenever a *post* is possible because the buffer is not empty and the acknowledgement of the previous message has arrived, it will eventually be carried out.) The only situation where the protocol might block is the waiting for an acknowledgement. However, the first *post* does not have to wait for an acknowledgement and since each *post* induces the receipt of an acknowledgement, it follows by induction that no *post* will be delayed forever.

4 Discussion

As we have seen, it is possible to implement causal order without piggybacking any additional information on the messages and without blocking the sender until remote delivery occurs. The driving idea was that it should be possible to realize causal order by weakening the conditions for synchronous computations. The intuition behind this was that since the class of causally ordered computations is located between the synchronous and FIFO computations, causal order should be "reachable" from both sides — instead of generalizing the piggybacking of FIFO sequence numbers to matrices, we added input/output buffers to the synchronous acknowledgement scheme. In this section we compare our protocol to the matrix protocol, discuss some limitations, and comment on performance issues.

4.1 Comparison to the Matrix Protocol

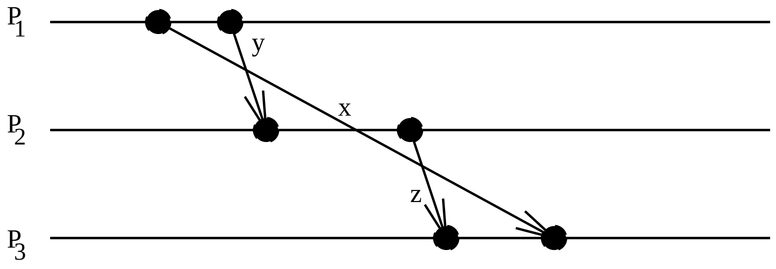

Fig. 4. A Non-Causally Ordered Execution of the First Program

In order to compare our "buffer protocol" to the classical point–to–point causal order delivery protocol based on dependency matrices [SES89, RST91], consider the following distributed program with three processes:

P_1: `send` x `to` P_3`; ... send` y `to` P_2`;...`
P_2: `receive; ... send` z `to` P_3`;...`
P_3: `receive; ... receive;...`

Figure 4 shows a possible computation if this program is executed on a general asynchronous system. Since the computation is not causally ordered, a causal order delivery protocol must *avoid* this particular execution. The *matrix protocol* does this by not immediately delivering message z when it arrives at P_3 (because the matrix attached to z would show that z depends on x which has not yet arrived). Instead of delivering message z, it would keep it in a buffer until message x has arrived. Thus, the computation depicted in Fig. 5 would be generated. The *buffer protocol* yields the same causally ordered computation, but as Fig. 6 indicates this is achieved by delaying message y (i.e., the *first* message

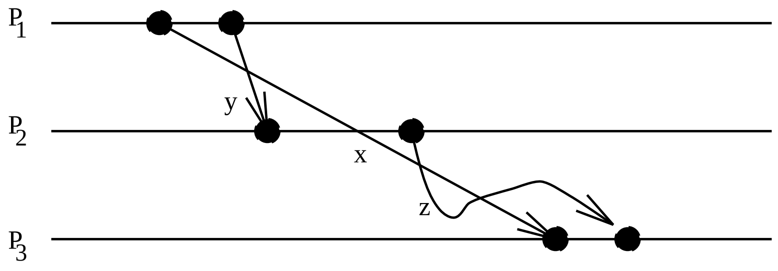

Fig. 5. An Execution Generated by the Matrix Protocol

of the message chain potentially overtaking message x), and not the last message z. Hence, the matrix protocol potentially delays a message at the receiver's site, whereas the buffer protocol does this at the sender's site. The buffer protocol therefore behaves conservatively compared to the more optimistic matrix protocol. The latter, however, must label the messages with causal dependency information so that it can be checked at the receiver's site whether the optimism was justified.

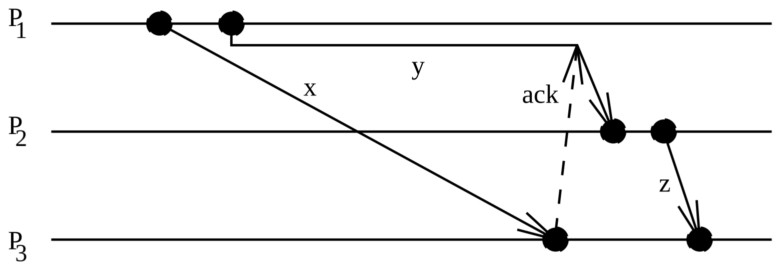

Fig. 6. An Execution Generated by the Buffer Protocol

Note that the receive statements of the program shown above are *non-selective*—they do not specify the sender's name (or select the message to be received according to some other criterion). If we allow *selective* (sometimes also called *conditional*) receive statements, process P_3 could be programmed as follows:

P_3: `receive from` P_2; `...` `receive from` P_1;`...`

Here, process P_3 would specify that it first wants to receive message z, and only then message x. Clearly, this is in contradiction to the causal order property! How would the two protocols behave? The *matrix protocol* would not deliver message z before message x is received. However, message x is not a message from P_2 as requested by the first (blocking) receive statement of P_3. Hence, the computation would block at this point. The *buffer protocol* would put message x into P_3's input buffer, message z will be inserted *after* x. Since messages in the FIFO queue cannot be reordered and P_3 is not willing to accept the first message (because it comes from the wrong sender), the computation will also block at the first receive statement of P_3.

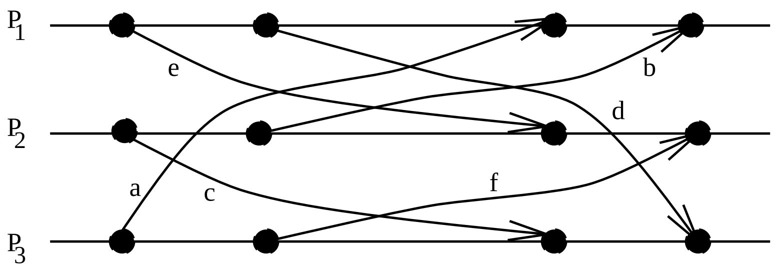

Fig. 7. An Execution of the Second Program with Overlapping Crowns

4.2 The Problem of Selective Receive Statements

The example of the previous subsection shows that it is dangerous to make use of selective receive statements when at the same time requiring causal order messaged delivery—the delivery order conditions imposed by the two mechanisms might contradict each other, thus allowing no execution of the program. If the conditions of selective receive statements happen to be in accordance with the causal order property, we would expect no problems. Interestingly, however, the next example shows that when using selective receive statements there are subtle cases where the two protocols differ in an essential way. Consider the following distributed program:

P_1: `send` e `to` P_2`;` `send` d `to` P_3`;` `receive; receive;`
P_2: `send` c `to` P_3`;` `send` b `to` P_1`;` `receive; receive;`
P_3: `send` a `to` P_1`;` `send` f `to` P_2`;` `receive; receive;`

Note that all possible computations generated by the program are causally ordered—no message can be overtaken in an indirect way since for each process there are no sends after the first receive. Figure 7 and Fig. 8 show two possible computations, both contain so-called *crowns* (i.e., substructures of k corresponding send-receive events s_i, r_i such that $s_1 \prec r_2,\ s_2 \prec r_3,\ \cdots,\ s_{k-1} \prec r_k,\ s_k \prec r_1$, see [CMT94]). Figure 7 can be generated by the buffer protocol in the following way: First, the initial sends of each process are executed and messages e, c, a are inserted into the receivers' input buffers. Then the next three sends are executed and messages d, b, f are inserted into the input buffers behind the first messages. The messages are now ordered in such a way that they can be correctly delivered with the subsequent receives.

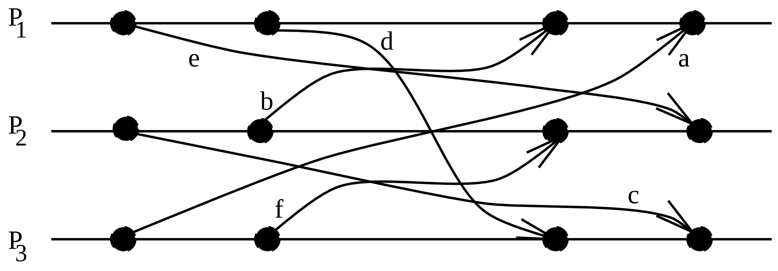

Fig. 8. An Execution with Nested Crowns

On the other hand, the computation depicted in Fig. 8 *cannot* be generated by the buffer protocol. We prove this by showing that there is a cyclic dependency among the get events:

$g^b \prec g^a$ (from $r_1^b \prec r_1^a$ and the FIFO property of IB_1)
$g^c \prec g^b$ (from $s_2^c \prec s_2^b$ and Lemma 3)
$g^d \prec g^c$ (from $r_3^d \prec r_3^c$ and the FIFO property of IB_3)
$g^e \prec g^d$ (from $s_1^e \prec s_1^d$ and Lemma 3)
$g^f \prec g^e$ (from $r_2^f \prec r_2^e$ and the FIFO property of IB_2)
$g^a \prec g^f$ (from $s_3^a \prec s_3^f$ and Lemma 3)

By transitivity, the contradiction $g^a \prec g^a$ follows.

If selective receive statements are used to enforce the computation depicted in Fig. 8, the buffer protocol would therefore block, whereas it is easy to see that the computation can be executed with the matrix protocol. Hence there exist *causally ordered computations which cannot be generated by the buffer protocol.* However, this fact should only be relevant when selective receive statements are used which specify the sender of a message to be received. As we have shown above, such selective receive statements are problematic anyhow when used in connection with the causal order property. If no selective receive statements are used, the buffer protocol will always yield a causally ordered execution of the program.

Selective receive statements do also cause problems to the buffer protocol in other situations. Consider the following distributed program:

P_1: `send` x `to` P_2;
P_2: `receive from` P_1; `receive from` P_3;
P_3: `send` y `to` P_2;

Messages x and y are concurrent—depending on various "nondeterministic" circumstances (e.g., message transmission times, processor speed, system load), x may arrive at P_2's input buffer IB_2 before y or vice-versa. In the first case, the buffer protocol will successfully execute the program, in the second case process P_2 blocks at its first receive statement. Although the send events of the two messages do not causally depend on each other, the buffer protocol cannot reorder the messages in the input buffer. The reason is that IB_2 has no knowledge about causal dependencies (or independencies) of its messages. This is in contrast to the matrix protocol where (with the help of the dependency matrix attached to each message) this knowledge is conveyed to the input buffers and processes. Hence, the matrix protocol can deliver the message sent by P_1 in the program above even if it is not the first message inserted into the buffer.

4.3 Performance Issues

The discussion of the previous subsection shows that the buffer protocol cannot reasonably be used with selective receive statements. This, together with the additional acknowledgement messages, is the price one has to pay for not piggybacking matrices of size $O(n^2)$ on the messages. The acknowledgement message, however, is not that expensive as it seems at first sight. Acknowledgements are short, and they are transmitted between buffer instances only, not between the processes themselves. Often, acknowledgements are used in any case by the underlying transmission protocol and therefore impose little or no additional overhead.

Since in our protocol an output buffer does not transmit any further messages until the acknowledgement for the previous message has been received[4], communication may incur extra delays. However, this should only be relevant when send events directly follow each other or when communication bandwidth is almost saturated. The maximum benefit of our approach would be achieved in a system with very low–latency round–trip acknowledgements, such as may be possible over ATM network hardware. In these settings one would expect that the acknowledgement already arrived before the next message is handed to the output buffer. Recall that an input buffer sends an acknowledgement directly after getting a message, that buffers are almost always ready to insert a message, and that even if the transmission of a message by an output buffer is delayed for some time, the corresponding process can proceed with its computation and perform further send and receive actions. Hence, the extra communication delay should be barely noticeable, and with respect to efficiency the new protocol should compare favorably to the matrix protocol[5]. However, we must admit that we currently lack experimental demonstrations which would support that claim.

[4] If the transport layer does not reorder messages, subsequent messages to the same receiver can be transmitted by an output buffer without waiting for acknowledgements. Hence, the protocol imposes a possible delay only when two successive messages are sent to different receivers.

[5] Note that the matrix protocol does also need buffers and that the piggybacking of matrices also requires time and communication bandwidth.

5 Conclusions

In this paper we proposed a non–blocking low overhead algorithm to implement point–to–point causal order message delivery. Since synchronous communication trivially implements the causal order property, we adopted this principle at the "transport layer" using an acknowledgement-based handshake protocol, but decoupled the processes with the help of buffers. As we have shown, the resulting scheme is usually not applicable to programs with selective receive statements, but otherwise it compares favorably with the earlier matrix protocol and it scales much better: Instead of piggybacking huge matrices on each message, a low-level acknowledgement between buffers at the transportation layer is used. We expect the extra delay caused by messages waiting in an output buffer not to be a major problem. Furthermore, the protocol is simple and easily implementable.

Future work should consider whether the matrix protocol and the buffer protocol can be combined in a reasonable way, how the protocols differ with respect to fault tolerance, and in which way the buffer protocol can possibly be adapted to causal broadcasts.

Acknowledgements. We would like to thank Uli Lemberg for providing us with the nested crowns example, and Ken Birman and André Schiper for general comments on earlier versions of this paper.

References

[AB92] Acharya A., Badrinath B., *Recording Distributed Snapshots Based on Causal Order of Message Delivery*. Information Processing Letters 44, 1992, pp. 317-321

[AV94] Alagar S., Venkatesan S., *An Optimal Algorithm for Distributed Snapshots with Causal Message Ordering*. Information Processing Letters 50, 1994, pp. 311-316

[BJ87] Birman K., Joseph T., *Reliable Communication in the Presence of Failures*, ACM Trans. on Computer Systems 5, 1987, pp. 47–76

[BR94] Birman K., van Renesse R. (eds.), *Reliable Distributed Computing with the Isis Toolkit*, IEEE Computer Society Press, 1994

[BSS91] Birman K., Schiper A., Stephenson P., *Lightweight Causal and Atomic Group Multicast*, ACM Trans. on Computer Systems, 9(3) (August 1991), 272-314

[CMT94] Charron-Bost B., Mattern F., Tel G., *Synchronous, Asynchronous, and Causally Ordered Communication*. Submitted to Distributed Computing, 1994

[Fid88] Fidge C., *Timestamps in Message-Passing Systems that Preserve the Partial Ordering*. Proc. 11th Autralian Computer Science Conf., University of Queensland, 1988, pp. 55-66

[KK89] Kearns P., Koodalattupuram B., *Immediate Ordered Service in Distributed Systems*. Proc. 9th International Conference on Distributed Computing Systems, Newport Beach, California, June 5-9, 1989, pp. 611-618

[Lam78] Lamport L., *Time, Clocks, and the Ordering of Events in a Distributed System.* Comm. of the ACM 21 (7), 1978, pp. 558-565

[Mat88] Mattern F., *Virtual Time and Global States of Distributed Systems.* In: Cosnard M. et al. (eds.): Proc. Workshop on Parallel and Distributed Algorithms, Bonas, France, 1988, pp. 215-226. (Reprinted in: Z. Yang, T.A. Marsland (eds.), *Global States and Time in Distributed Systems*, IEEE, 1994, pp. 123-133)

[RST91] Raynal M., Schiper A., Toueg S., *The Causal Ordering Abstraction and a Simple Way to Implement it.* Information Processing Letters 39, 1991, pp. 343-350

[SES89] Schiper A., Eggli J., Sandoz A., *A New Algorithm to Implement Causal Ordering.* In: J.-C. Bermond, M. Raynal (eds.), *Distributed Algorithms*, Vol. 392 of *Lecture Notes in Computer Science*, Springer-Verlag, 1989, pp. 219–232

[SM94] Schwarz R., Mattern F., *Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail.* Distributed Computing 7 (3), 1994, pp. 149-174

[TM93] Tel G., Mattern F., *The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes*, ACM Trans. on Prog. Lang. Sys. 15 (1), 1993, pp. 1-35