Optimistic Total Order in Wide Area Networks*

António SOUSA , José PEREIRA, Francisco MOURA, Rui OLIVEIRA
Universidade do Minho, Portugal
{als,jop,fsm,rco}@di.uminho.pt

Abstract

Total order multicast greatly simplifies the implementation of fault-tolerant services using the replicated state machine approach. The additional latency of total ordering can be masked by taking advantage of spontaneous ordering observed in LANs: A tentative delivery allows the application to proceed in parallel with the ordering protocol. The effectiveness of the technique rests on the optimistic assumption that a large share of correctly ordered tentative deliveries offsets the cost of undoing the effect of mistakes.

This paper proposes a simple technique which enables the usage of optimistic delivery also in WANs with much larger transmission delays where the optimistic assumption does not normally hold. Our proposal exploits local clocks and the stability of network delays to reduce the mistakes in the ordering of tentative deliveries. An experimental evaluation of a modified sequencer-based protocol is presented, illustrating the usefulness of the approach in fault-tolerant database management.

1. Introduction

Total order multicast greatly simplifies the implementation of fault-tolerant services using the replicated state machine approach [25]. By ensuring that deterministic replicas handle the very same sequence of requests from clients, it is ensured that the state is kept consistent and the interaction with clients is serializable [12]. A particularly interesting application is the database state machine [21] which allows high performance replication of transactional databases.

Implementation of total order multicast is however more costly than other forms of multicast due to the unavoidable additional latency. For instance, in a sequencer based protocol [5, 16] all processes (except the sequencer itself) have to wait for the message to reach the sequencer and for the sequence number to travel back before the message can be delivered.

On the other hand, protocols based on causal history [18, 23, 10] can provide latency proportional to the interarrival delay of each sender and thus lower latency than sequencer based protocols. However, when each sender has a large in-

terarrival time and low latency is desired, this requires the introduction of additional control messages. This is especially unfortunate in large groups and in wide area networks with limited bandwidth links.

In some protocols, such as those based on consensus [7, 4] or on a sequencer [5, 16], the total order decided is the spontaneous ordering of messages as observed by some process. In addition, in local area networks (LANs) it can be observed that the spontaneous order of messages is often the same in all processes. The latency of total order protocols can therefore be masked (not reduced) by tentatively delivering messages based on spontaneous ordering, thus allowing the application to proceed the computation in parallel with the ordering protocol [17]. Later, when the total order is established and if it confirms the optimistic ordering, the application can immediately use the results of the optimistic computation. If not, it must undo the effects of the computation and restart it using the correct ordering.

The effectiveness of the technique rests on the assumption that a large share of correctly ordered tentative deliveries offsets the cost of undoing the effects of mistakes. This is unfortunate as this makes optimistic delivery useful only in LANs where the latency is much less of a problem than in wide area networks (WANs).

This paper proposes a simple protocol which enables optimistic total order to be used in WANs with much larger transmission delays where the optimistic assumption does not normally hold. Our proposal exploits local clocks and the stability of network delays to reduce the mistakes in the ordering of tentative deliveries by compensating the variability of transmission delays. This allows protocols which are based on spontaneous ordering to fulfill the optimistic assumption and thus mask the latency.

An experimental evaluation of the technique is presented using a sequencer-based protocol, illustrating the usefulness of the approach in fault-tolerant database management. When applied to the sequencer based protocol, our technique does not introduce additional messages. The only overhead is that of an additional integer piggybacked on data messages. This compares favorably with both plain sequencer based and causal history based protocols.

The paper is structured as follows. The next section recalls the problems of total order and optimistic total order multicasts, as well as the reasons preventing spontaneous

total order in wide are networks. Section 3 introduces the intuition underlying our proposal and presents a protocol providing optimistic delivery of messages based on a fixed-sequencer total order multicast protocol. In Section 4 we evaluate the performance gains of our approach. In Section 5 we discuss the paper contribution in general settings as well as applied to a specific application. Section 6 concludes the paper.

2. Background

2.1. Totally ordered multicast

Informally, totally ordered multicast (or atomic multicast) ensures that no pair of messages is delivered to distinct destination processes in different order. Totally ordered multicast greatly simplifies the implementation of fault-tolerant services using the replicated state machine (or active replication) approach [25, 12]: By delivering exactly the same messages in the same order to a set of deterministic replicas, their internal state is kept consistent.

More formally, we consider an asynchronous message passing system composed of a finite set of sequential processes communicating over a fully connected reliable point-to-point network [7]. Processes do not have access to shared memory or to a global clock. A process may only fail by crashing and once a process crashes it does not recover. A process that never crashes is said correct. Totally ordered multicast is defined by primitives *to-multicast*(m) and *to-deliver*(m), and satisfies the following properties [14]:

Validity. If a correct process to-multicasts a message m, then it eventually to-delivers m.

Agreement. If a correct process to-delivers a message m, then every correct process eventually to-delivers m.

Integrity. For every message m, every process to-delivers m at most once, and only if m was previously to-multicast.

Total Order. If two correct processes to-deliver two messages m and m', then they do so in the same order.

Total order multicast has been shown to be equivalent to the generic agreement problem of consensus [7]. Therefore we must assume that in our system the consensus problem is solvable [11, 7], requiring that a majority of processes is correct and that failure detection is of class $\diamond S$ [6]. In some protocols, consensus is explicitly invoked to decide the message sequence [7, 4]. In others, consensus is implicit in a group membership service which supports the actual ordering protocol [5, 15, 10].

There is a plethora of total order protocols for asynchronous message passing systems which can be classified according to several criteria [9]. Namely, some order the message while disseminating it [3, 2, 15]. Others take advantage of an existing unordered multicast protocol [5, 16, 7] and work in two stages: First, messages are
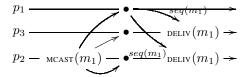

Figure 1. Sequencer based total order protocol.

disseminated using a reliable multicast protocol Then, an ordering protocol is run to decide which is the correct delivery sequence of buffered messages. This results in additional latency, when compared to reliable multicast.

An example of a protocol often used in group communication toolkits is the sequencer [5, 16], which uses consensus implicitly in the view-synchronous reliable multicast protocol used to disseminate messages previously to ordering them. As depicted in Figure 1, a data message is disseminated using unordered reliable multicast. Upon reception (depicted as a solid dot), the message is buffered until a sequence number for it is obtained. A single process (p_1 in the example) is designated as the sequencer: it increments a counter and multicasts its value along with the original message's identification to all receivers as a control message. Data messages are then delivered according to the sequence numbers. A group membership protocol is used to ensure that for any given data message there is exactly one active sequencer.

Besides being a very simple protocol, it offers several advantages, especially in networks with limited bandwidth or in large groups with large and variable message interarrival times: it requires at most a single additional control message for each data message and any message can always be delivered after two successive message transmission delays. The basic protocol is also easily modified to cope with higher message throughput by batching sequence numbers for several messages in a single one [5], reducing the number of control messages at the expense of higher latency.

2.2. Optimistic total order

A reliable multicast protocol can deliver a message after a single transmission delay from the originator to the receiver. This contrasts with the latency of totally ordered multicast[1] which is either twice as large, when using a sequencer based protocol, or proportional to message interarrival delay in protocols using causal history. However:

- Some protocols, such as the sequencer, produce an ordering which is the spontaneous ordering observed by some process.
- In local area networks, it can be observed that the spontaneous ordering of message reception of all processes

---

[1]Except in the degenerate situation where a single process is multicasting and it can assume the sequencer role.

is often very similar, therefore, similar to the final ordering decided by the sequencer.

Nevertheless, delivery incurs always in the additional latency. The optimistic atomic broadcast protocol [22] takes this in consideration to improve average delivery latency of a consensus based total order protocol.

Further latency improvements can be obtained if the application itself can take advantage of a tentatively ordered delivery. This is called optimistic delivery [17, 26] as it rests on the optimistic assumption that reliable multicast spontaneously orders messages. It also implies that eventually an authoritative total order is determined, leading to a confirmation or correction of previously used delivery order. To the interval between the optimistic delivery and the authoritative delivery we call *optimistic window*. It is during this interval that the application can optimistically do some processing in advance.

To define optimistic total order multicast we use two different delivery primitives an optimistic *opt-deliver*(m) that delivers messages in a tentative order and a final *fnl-deliver*(m) that delivers the messages in their final, or authoritative, order. Optimistic total order multicast satisfies the following properties [26]:

Validity. If a correct process to-multicasts a message m, then it eventually fnl-delivers m.

Agreement. If a correct process fnl-delivers a message m, then every correct process eventually fnl-delivers m.

Integrity. For every message m, every process opt-delivers m only if m was previously multicast; and every process fnl-delivers m only once, and only if m was previously multicast.

Local Order. No process opt-delivers a message m after having fnl-delivered m.

Total Order. If two processes fnl-deliver two messages m and m', then they do so in the same order.

An example of such an application is the database state machine [21] which allows high performance replication of transactional databases and works as follows: transactions are executed optimistically by any of the replicas without locking. The resulting read and write sets are then multicast to all replicas which perform a deterministic certification to ensure that the transaction does not conflict with concurrent transactions already committed. Total order multicast is used to ensure that the result of the certification process is identical in all replicas, thus ensuring consistency. If the order of messages is known in advance by optimistic delivery, this can be used to speed up the certification [17].

Notice that if the optimistic ordering turns out to be wrong, the application has to undo the effect of any processing it might have done. Therefore, the net advantage of optimistic delivery depends on the balance between the cost of a mistake and the ratio of correctly ordered optimistic deliveries. In the database state machine, being able to undo the effects of optimistic delivery just means than the transaction cannot be effectively committed until authoritative delivery. When the optimistic delivery is wrong, there is a performance penalty: The processing resources used have been wasted.

The tradeoff is thus similar to the one involved in the design of cache memories. However, the protocol designer has no possibility to reduce the cost of a mistake, as this depends solely on the application. The only option is thus to try to maximize the amount of messages which are delivered early but correctly ordered.

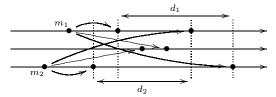## 2.3. Obstacles to spontaneous total order

A high ratio of spontaneously totally ordered messages which results in good performance of optimistic applications is not trivially achieved, especially in wide area networks. One reason for this is loopback optimization in the operating system's network stack. Noticing that the outgoing packet is also to be delivered locally, the operating system may use loopback at higher layers of the protocol stack and immediately queue the message for delivery. This allows it to be delivered in advance of packets from other senders which have reached the network first.
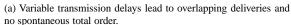
Another reason for out of order delivery lies in the network itself. Although not frequent, there is a possibility that packets are lost by some but not all destinations. A reliable multicast protocol detects the occurrence and issues a retransmission. However, the delay introduced opens up the possibility of other packets being successfully transmitted while retransmission is being performed.
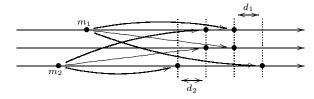
An additional issue is the complexity of the network topology. Different packets can be routed by different paths, being therefore subject to different queuing delays or even to being dropped by congested routers. This is especially noteworthy when there are multiple senders. Receivers which are nearer, in terms of hops, to one of them will receive its messages first. Receivers which are nearer of another will possibly receive messages in the opposite order.

Notice however that bad spontaneous order in wide area networks is not attributable to large delays themselves, but to the fact that the delays to different destinations are likely to be different, often by two orders of magnitude. Consider Figure 2(a). Messages m_1 and m_2 are multicast to three different processes, including the senders themselves. The time taken to transmit each message varies with the recipient, for instance, transmission to the sender itself (typically hundreds of microseconds by loopback) takes less time than transmission to other processes (typically up to tens of milliseconds over a long distance link). The result is that process p_1 spontaneously orders message m_1 first while p_2 and p_3 deliver m_2 first.

Figure 2(b) shows a similar example where message transmission delays are longer but where is it more likely

(a) Variable transmission delays lead to overlapping deliveries and no spontaneous total order.

(b) Comparable transmission delays reduces the probability of overlapping deliveries.

Figure 2. Transmission delays and spontaneous total order.

that messages are delivered by all processes in the same order. What matters is the difference between transmission delays to different processes depicted as d_1 and d_2. Larger values for d_1 and d_2 mean that there is a higher probability of overlapping and thus of different delivery order even with higher delays than the previous example.

3. Delay compensation

3.1. Intuition

A network exhibiting identical transmission delays with low variance among any pair of processes would enable spontaneous total ordering of messages. This observation leads to the intuition underlying our proposal: given the magnitude of the latency introduced by total order protocols it should be possible, by judiciously scheduling the delivery of messages, to reduce the differences among transmission delays and *produce* an optimistic order which is likely to match the authoritative total order. As an example, notice that Figure 2(a) can be transformed in Figure 2(b) simple by delaying some of the deliveries.

What remains to be established is how to determine the correct delays to introduce to each message such that the likelihood of matching the authoritative total order is improved. The challenge is to do this with minimal overhead, both in terms of messages exchanged as well as computational effort. In addition, by introducing delays our technique increases the average latency of optimistic delivery. This must therefore be minimized and compensated by the higher share of correctly ordered optimistic deliveries.

Notice that in a WAN this cannot ever replace a total order algorithm: Transmission delays cannot be precisely estimated, some uncertainty exists and thus it is likely that some messages are delivered out of order [20]. On the other hand, if the only modification to the original sequencer algorithm is the introduction of finite delays, its correctness in an asynchronous system model is unaffected. Therefore by reusing an algorithm known to be correct in the asynchronous system model we ensure the robustness of the solution [19]. Timing assumptions, namely on the stability of transmission delays as measured by a process's local clock are then used only to improve the performance.

3.2. Relatively Equidistant Receivers

As the basis for our protocol, we consider a fixed-sequencer total order multicast algorithm as described in Section 2.1. We assume that the total order of messages is based on the spontaneous ordering of messages as seen by the sequencer.

The different orders seen by a process p between the messages it delivers optimistically and those that it delivers authoritatively reflects the *relative* differences between the communication delays from the senders to p and to the sequencer. We like to think of these communication delays as "distances" between processes (more precisely, as directed distances as the distance from p to q can be different from that of q to p). If, through the introduction of artificial delays, we manage to get each process p and the sequencer as relatively equidistant receivers with respect to all other processes, then the order in which p delivers messages optimistically will be that of the sequencer and therefore will match the authoritative order.

The way to increase the distance between q and p is to delay the optimistic delivery of messages from q at p. This means that when p needs to get q closer either p reduces the delay it might be imposing to the messages from q, or p has to stand back from all other processes by delaying the optimistic delivery of messages from these processes. This is the basic mechanism of our algorithm. It is simple and independently managed at each process, i.e., the adjustment of the distance between p and q is independent from that between q and p.

Two particular cases however require special attention. One is the fact that any process is usually closer to itself than from the sequencer and thus it will have to distance from itself. This case is simple, each process will delay the optimistic delivery of its own messages such that "it distances from itself" as it distances from the sequencer. The other case regards the sequencer itself. While, as any other process, it is closer to itself than the others the distance to the sequencer does not apply here and the order of optimistic delivery trivially matches that of the authoritative's. However, it is required, as happens with the other processes, that the sequencer "distances from itself" by delaying the optimistic delivery of its own messages. The reason for this is that unless the sequencer delays the optimistic de-
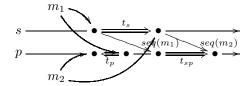
Figure 3. Delays used in adjusting.

livery of its own messages, the optimistic and authoritative delivery of its messages will always occur almost simultaneously. This is true at the sequencer process itself as well as in any other process and, as exemplified in the next section, it would eventually force the same phenomenon in the messages of the other processes. The problem of delaying the optimistic delivery of the sequencer's messages is that it also delays their authoritative delivery.

3.3. Distance calculation

Consider the scenario depicted in Figure 3. Message m_1 is multicast by a process p_1. Message m_2 is multicast by another process p_2. Both the sequencer s and a second process p receive m_1 and m_2 as shown. Upon reception they are ordered by s, which assigns them sequence numbers and delivers them immediately. The authoritative order of the messages becomes m_1, m_2 as this was the spontaneous order seen by s. In contrast, process p can only make an optimistic guess about the final relative order of m_1 and m_2, and in this situation it would have mistakenly predicted the delivery of m_2 before m_1. The final order is known only upon reception of the sequence numbers from s.

As soon as it receives the sequence numbers for both messages, p becomes aware that its relative distances to p_1 and p_2 are different from those of s, because it has received the same messages in the inverse order. If it had delayed the optimistic delivery of m_2 until after the reception of m_1, it would have compensated its relative distance from the senders with respect to that of the sequencer and matched the authoritative order.

Although any sufficiently large delay imposed on m_2 by p would correctly order it relatively to m_1, a correct prediction of the final order by p requires an evaluation of relative distances to senders to s and to p, enabling an optimal delay to be introduced. Notice that the delay should not be so large that it causes m_2 to be misordered with a message m_3 that arrives to all processes after both m_1 and m_2.

Explicit estimation of distances among all processes is not required. A better approach is to directly determine optimal delays to be introduced prior to optimistic delivery by observing that:

- If the relative distance of p and s is the same with respect to senders of m_1 and m_2 and each message is

multicast simultaneously to all destinations, then interarrival times t_s and t_p will be identical.
- If transmission delays of $seq(m1)$ and $seq(m2)$ from s to p are the same, then p can use the value of t_{sp} to locally determine t_s. This avoids assumptions on the drift rate of clocks.

Process p can easily calculate the delay it should have introduced to the optimistic delivery of m_2 to match its relative distance from p_1 and p_2 to that of the sequencer. Specifically, it should have delayed m_2 by $t_{sp} - t_p$.[2] To cope with spurious variations on transmission delays, adjustments are made taking into account an inertia pondering factor.

In the next section we will see in detail how the delays are calculated and which process's messages are delayed. Right now, the reader should keep in mind that delays to a process's messages are only introduced when it is not possible to achieve the same result by reducing the delays inflicted to the other.

The way the sequencer calculates its own messages delays is different. Should it use the same method as the others and it, obviously, would not delay its own messages. To understand the method followed by the sequencer let us first exemplify the consequences of not introducing delays on the sequencer's own messages. Consider three processes, p, q and s. Process s is the sequencer. Process q, for simplicity, is δ equidistant of s and p. The distance from s to p is d_{sp} and the distance of s to itself is d_{ss}.

Having p and s relatively equidistant from q means that $(\delta - d_{ss}) = (\delta - d_{sp})$. To achieve this, since we cannot reduce d_{sp}, we can have 1) p to distance from q, or 2) s to distance from itself, or both. Now suppose that s does not delay its own messages. In this case, p will have to stand back $\Delta = d_{sp} - d_{ss}$ from q. Since d_{ss} (the loopback delay) is usually negligible we can admit that $\Delta \simeq d_{sp}$. This means that when a message multicast by q is optimistically delivered at p it is almost simultaneously delivered authoritatively at p too. Therefore, unless the sequencer delays the optimistic delivery of its own messages the size of the optimistic window at the other processes becomes uninteresting or even vanishes.

Below we will show how the sequencer computes the delay for its own messages. This, contrary to other processes adjustments, is not independent and requires their cooperation. The idea is that the sequencer will stand back from itself what it distances from the farthest process.

3.4. Algorithm

We present in this section the algorithm executed by each process (Figure 4). The algorithm consists of a procedure *TO-multicast(m)* invoked by the client application to multicast a message and a set of four *upon-do* statements, exe-

---

[2]Notice that t_p is negative in Figure 3, indicating that the relative order of m_1, m_2 is reversed.

```
 1: g ← 0 {Global sequence number}              21: upon R_deliver(SEQ(m, s)) do
 2: l ← 0 {Local sequence number}               22:    S ← S ∪ {(m, s, now)}
 3: R ← ∅ {Messages received}
 4: S ← ∅ {Sequence numbers}                    23: upon ∃(m, d, o) ∈ R : (m, l + 1, t) ∈ S ∧ m ∉ F do
 5: O ← ∅ {Messages opt-delivered}              24:    fnl_delivery(m)
 6: F ← ∅ {Messages fnl-delivered}              25:    if ∃(m', d', o') ∈ R : (m', l, t') ∈ S then
 7: delay[1..n] ← 0                             26:       Δ ← (t − t') − (o − o')
 8: r_delay[1..n] ← 0 {Delays requested to the sequencer}  27:       if Δ > 0 then
                                                28:          adjust(m'.sender, m.sender, Δ)
 9: procedure TO_multicast(m) do                29:       else
10:    R_multicast(DATA(m, max(delay[]) − delay[seq]))  30:          adjust(m.sender, m'.sender, |Δ|)
                                                31:    l ← l + 1
11: upon R_deliver(DATA(m, d)) do               32:    F ← F ∪ {m}
12:    R ← R ∪ {(m, d, now + delay[m.sender])}
                                                33: procedure adjust(i, j, d) do
13: upon ∃(m, d, t) ∈ R : now ≥ t ∧ m ∉ O ∧ m ∉ F do  34:    v ← (delay[i] × α) + (delay[i] − d) × (1 − α)
14:    opt_delivery(m)                          35:    if v ≥ 0 then
15:    O ← O ∪ {m}                              36:       delay[i] ← v
16:    if p = seq then                          37:    else
17:       g ← g + 1                             38:       delay[i] ← 0
18:       R_multicast(SEQ(m, g))                39:       delay[j] ← delay[j] + |v|
19:       r_delay[m.sender] ← d
20:       delay[p] ← max(r_delay[])
```

Figure 4. Delay compensation algorithm for process p

cuted atomically, that deal with the optimistic and authoritative delivery of the messages. The actual delivery of the messages to the client application is done through two up-calls *opt-deliver(m)* and *fnl-deliver(m)*. Procedure *adjust* is an auxiliary procedure local to the algorithm.

Each process manages four queues R, O, F and S where it keeps track of the messages received, optimistically and authoritatively delivered to the application, and those for which it has already received a sequence number, respectively. Every message *m* has a special attribute (*m.sender*) identifying its sender. At each process a variable *seq* identifies the sequencer process.

To multicast a totally ordered message, the client application invokes procedure *TO-multicast(m)* (lines 9-10). This, in turn, invokes an underlying primitive providing reliable multicast with a pair $(m, max(delay[]) − delay[seq])$. The value computed by $max(delay[]) − delay[seq]$, as will be discussed below, corresponds to the delay the process suggests the sequencer to inflict to its own messages.

The reception and delivery of messages is done by the four *upon-do* statements. The first two handle the optimistic delivery while the others handle the authoritative delivery.

When a process p receives a message *m* (line 11) it simply adds *m* as a tuple (m, d, d') to the queue of received messages scheduling its delivery for after the delay d' inflicted by p to the sender of *m*. When this timer expires and if *m* was not already optimistically delivered ($m \notin O$) nor authoritatively delivered ($m \notin F$), which corresponds to the condition on line 13, then *m* is optimistically delivered to the application and the fact registered by adding *m* to the O queue. If p happens to be the sequencer it computes a sequence number to give to *m* and reliably multicasts a sequence message composed by *m*'s id and its sequence number. Afterwards, p (if in the role of sequencer) takes parameter d just received with *m* and adjusts the delays it imposes to its own messages.

Upon receiving a sequence message at line 21, each process simply adds the received tuple (message id and sequence number) plus the current time to the queue of sequence numbers S.

Once a message *m* that has already been received ($m \in R$) gets a sequence number in the S queue and its sequence number corresponds to the next message to be authoritatively delivered (the whole condition at line 23), then *m* is authoritatively delivered to the application through *fnl_deliver*. At this point, the algorithm computes the adjustments that might need to be done to the delays inflicted to the sender of *m* or to the sender of the message *m'* delivered just before *m*. To do this we consider the interval between the reception of the sequence number for *m'* and the sequence number for *m* given by $(t − t')$ and the interval between the optimistic reception of *m'* and the optimistic reception of *m* given by $(o − o')$. The difference Δ (line 26) between these intervals represents the relative adjustment that should have been done to the delays imposed to the optimistic delivery of *m'* or *m* to make the interval of the optimistic deliveries of these messages match that of the authoritative deliveries.

If Δ is negative it means that the optimistic order matched the authoritative's. If Δ is positive then either the order was reversed or the interval between optimistic deliveries is smaller than the interval between authoritative deliveries. Depending on this, procedure $adjust$ is called differently. In the first case *adjust* is called to decrease the delay put on messages received from the sender of *m*, otherwise it should decrease the delay inflicted to the sender of *m'*.

Procedure *adjust(p,q,d)* works as follows. Based on the delay d and on a inertia parameter α, we compute the new delay v to give to messages of p. If v becomes negative, then it means that we actually need to anticipate p's messages

which is not possible. Instead, we do not delay the messages of p but start delaying the messages of q by an additional $|v|$.

Finally, we explain how sequencer computes the delays on the optimistic delivery of its own messages. Every process when $R\_multicasts$ a data message (line 10) sends also the value of the greatest delay it is applying locally (this is usually the self delay) minus the delay it is currently inflicting to the sequencer messages. Only the sequencer makes use of this values keeping track of them on vector r_delay. The delay the sequencer inflicts on its own messages is given, at each moment, by the greatest value in r_delay.

4. Performance evaluation

4.1. Evaluation criteria

For an application to benefit from optimistic ordering it is required that the 1) tentative order closely matches the final order; and that 2) the time between optimistic delivery and final delivery is enough to do meaningful processing. Performance evaluation is done with an event-based simulation, which allows us to study the impact of the system and protocol parameters, as well as with an implementation of the protocol within a group communication toolkit, which validates simulation results.

The primary evaluation criteria is thus to compare optimistic and final orders of both the original and the modified sequencer protocol.

It turns out that this criterion is not entirely realistic in the context of the database state machine [21], which is the main motivation of this work. In fact, it has been shown that it is advantageous to certify transactions in batches, which allows them to be reordered in order minimize the number of conflicting transactions that must be aborted [21]. Therefore we also compare the optimistic ordering of batches of messages of size k in a similar fashion: If the first k messages in the final log differ from the next k messages in the optimistic log, a miss is recorded.

A second evaluation criterion is to compute whether the time between optimistic delivery and final delivery — the optimistic window — is enough to do meaningful processing. Finally, we study also the impact of delay compensation in end-to-end latency of final delivery: We want to make sure that the optimistic window is not increased at the expense of larger end-to-end latency. Considering a receiver process p, this is done both by averaging messages from different senders, as well as only for messages from the process p itself. The average latency has direct consequences on the abort rate due to the increased likelyhood of conflicting concurrent transactions.
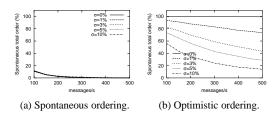

(a) Spontaneous ordering.　(b) Optimistic ordering.

Figure 5. Comparison of spontaneous with optimistic order after delay compensation.

4.2. Simulation model

Using discrete event simulation we study the performance of the protocol in a scenario without overheads for message processing and delivery and without application overheads. We consider a fully connected point-to-point network. The transmission delay in each link is normally distributed with parametrized mean and standard deviation. Message inter-arrival rate is exponentially distributed with equal mean in every participating process.

Although we have experimented with several combinations of transmission delays and number of processes, mimicking several network topologies, we present results in a situation where a long distance link separates two clusters of processes. This is the worst case scenario for spontaneous order: Messages which do not cross the long distance link are much faster and thus deliveries overlap with high probability.

Figure 5 shows the ratio of correctly ordered messages with both protocols. In these we have used an average transmission delay of 20ms within each cluster and 40ms when traversing the long distance link. The delay to the process itself is 0ms. There is no bandwidth limitation or message handling overhead. Each curve presents results for a different configuration of transmission delay variability. As observed in extensive measurements of the Internet [20], the standard deviation of transmission delays is mostly less than 10% for large data packets and often less than 1% for small control packets.

In the experiments we consider the pondering factor α of 95% for process delay adjustments. This is high enough not to degrade the delay estimate even with the maximum 10% standard deviation assumed, and allows for delays to stabilize quickly, typically in less that 100 messages. In networks with less variability, a lower α can be used in order to converge even faster. Each simulation is run for 100s, discarding initial messages needed for stabilization of delays.

With the original protocol (Figure 5(a)), the spontaneous order in the nodes which are more distant from the sequencer across the long distance link is almost inexistent. Nodes close to the sequencer exhibit higher ratios of correctly ordered messages, although not really worth using. When we use delay compensation, as shown in Figure 5(b),
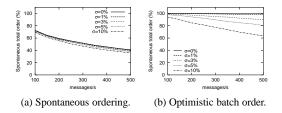
(a) Spontaneous ordering.   (b) Optimistic batch order.

Figure 6. Comparison of spontaneous with optimistic order after delay compensation with batches of size $k = 2$**.**

	messages from:	itself		all	
	compensation:	no	yes	no	yes
Sequencer	latency	0	41.4	28.5	32.3
	opt. window	0	0	0	0
Near	latency	40.1	40.2	48.8	52.6
	opt. window	40.1	21.8	20.3	20.2
Distant	latency	80.6	80.8	69.3	73.0
	opt. window	80.6	27.3	42.0	24.6

Table 1. End-to-end latencies and optimistic window sizes (in ms).

all nodes exhibit higher ratios of correctly ordered optimistic deliveries. In addition, ratios of nodes that are distant from the sequencer are very close to ratios of processes close to the sequencer.

Nevertheless, with high variability of $\sigma = 10\%$ and high message rates, the results are still not perfect. The reason for this is that in this situation the message interarrival interval (e.g. 2.5ms for 400 msg/s) is smaller than the standard deviation (e.g. 4ms for the longest link). Therefore, even after compensating delays it is likely that messages are still out of order. Notice however, that after compensation, instead of messages being out of order by an amount of time comparable to the transmission delay and thus far from their final position, it is likely that messages are just "off by one".

This is confirmed by the results of Figure 6, which shows similar results when spontaneous ordering of batches of size $k = 2$ is considered. Although the advantage of compensating delays is not as dramatic, it is possible to achieve very high rates of correctly ordered messages even when the system is subjected to a very high message throughput.

Therefore, this shows that although delay compensation does not immediately result in a perfect optimistic order, it enables the application to achieve an effective perfect optimistic order even for very high messages rates simply by batching pairs of messages. This is very interesting as with high thoughput the additional delay to work on pairs of messages is comparatively low. For instance, with 400 msg/s, the application has to wait only an additional 2.5ms to take advantage of a much better optimistic ordering.

Table 1 shows end-to-end latency and the size of usable optimistic window as measured by the sequencer, by a process near the sequencer and by a process distant to the sequencer. For each, values considering only the messages from the process itself and from all processes are shown. Notice that the optimistic window at the sequencer is always zero: It never performs optimistic deliveries as it orders messages as soon as received. Notice also that without delay compensation the optimistic window of each process equals delivery latency, as optimistic delivery is performed immediately upon multicast.

The tradeoff for the improved optimistic order is the additional latency caused by delay compensation. In particular,

this shows up in the delivery latency of messages multicast by the sequencer, which grows from zero to 41.4ms. In addition, the optimistic window in other processes is reduced, although mainly for messages from itself. This has to be weighted with the improvement in the ratio of messages correctly ordered.

4.3. Implementation

In order to validate the results obtained with the simulation model we implemented a sequencer protocol using Java based group communication toolkit. Although we have evaluated the performance of the protocol during stable periods when the membership is not changing, the view-synchronous multicast is used to migrate the sequencer role upon membership change and to order remaining messages after the sequencer fails [15].

The underlying reliable multicast protocol transmits messages using point-to-point UDP. The error recovery mechanism is initiated by the receiver when it discovers that messages are missing. Scalable gossip-style algorithms are used for stability tracking and failure detection [13]. Group membership and view-synchrony use a consensus protocol.

Measurements in a wide area network presented below were obtained by running the implementation on top of a simulated network. This setup allows a precise simulation of the components of interest by using a highly accurate timer to measure the duration of the implementation code. This approach has been shown to accurately represent real-time characteristics of the system being simulated while allowing centralized fault injection and omniscient observation [1]. The performance of the implementation code is thus related with that of the host workstation: a dual Pentium III/1 GHz workstation with 1GB memory using IBM Java 1.3. The network simulation used was the Scalable Simulation Framework (SSFNet) [8] which offers realistic routing behavior and allows us to introduce background traffic.

The topology used is that of a backbone network, connecting several leaf networks. The backbone network is composed by two routers connected through a 34Mbps long distance link. Each of these routers connects to the local network main router. The local network backbone are three
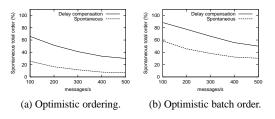
(a) Optimistic ordering. (b) Optimistic batch order.

Figure 7. Spontaneous and optimistic order in the implementation.

routers connected between them for fault tolerance, being two of them connected to the network main router. Departmental routers, to which application hosts are connected, are themselves connected to two of the backbone routers, for fault tolerance.

Experiments were conducted by placing a process in a departmental network of each of the leaf networks. In this setting between every two nodes there are between 8 and 11 communication hops. Notice that the existence of redundant links allows routing of packets to be done by slightly different routes, thus introducing variance in transmission delays. We have also configured realistic background HTTP traffic, by placing 80 clients and 20 servers evenly scattered across the whole network. Notice that protocol mechanisms themselves introduce additional background traffic for failure detection and stability tracking.

These is similar to the scenario described for the discrete event simulation. Message interarrival is also exponentially distributed with parameters from 20 to 100 ms, resulting in an aggregate throughput of 100 to 500 messages per second in the system. We have also used the same value for parameter α.

Figure 7 shows results comparable to those of the previous section. Notice that the experimental values show results similar to simulation results with $\sigma = 10\%$. In short, there is an improvement in the spontaneous total order ranging from less than around 20% to around 70% in workloads of 100 messages per second. Once again considering ordered batches of two messages the results improved.

5. Discussion

To understand the impact of this results in the application we need to evaluate the tradeoff between the benefit of optimistic execution and the penalty incurred by a wrong optimistic delivery.

Consider an application which cannot interrupt the processing of a message after it has started. This means that even if meanwhile there is a final delivery which shows that the optimistic delivery was wrong, it has to wait for it to finish to start the processing of the correct message. On the other hand, we do not consider any penalty for undoing the effect of the computation when required. Let g be the time required to process a message. If the latency of final delivery is l and no optimistic processing is done then latency of the whole computation is $l + g$.

Let w be the size of the optimistic window. If $g \leq w$ and there is no penalty, it is obvious that optimistic delivery is useful, as its latency is never worse than the original computation after delivery of the final message. However, if $g > w$ then the effective latency can be either:

- $l + g - w$, when the optimistic delivery is correct;
- $l + g - w + g$, when the optimistic delivery turns out to be wrong.

If r is the ratio of correct deliveries, the average latency is $r(l+g-w) + (1-r)(l+g-w+g)$. Therefore, optimistic delivery decreases latency if $g < w/(1-r)$.

Consider a distant process with $\sigma = 3\%$ and handling 100 msg/s. This results in hit ratios of 11.2% and 82.5% respectively without and with delay compensation. According to Table 1, respective window sizes are 42.0ms to 24.6ms. Before the optimization we cope with a $g < 47.2ms$. After delay compensation, it is possible to cope with $g < 140.7ms$.

A concern when using an optimistic algorithm is that the performance of the final delivery is not affected, i.e. the optimistic delivery does not increases the algorithm latency. This is not an issue, as we are only delaying optimistic deliveries, and as soon as a sequence number for a message locally available is known it is immediately be delivered, even if its optimistic delivery has been erroneously delayed by an excessive amount of time. Nevertheless, it is up to the application to, as soon as possible, interrupt processing of an optimistic delivery if the final delivery happens to be of a different message. A second concern when implementing our technique is the granularity of operating system timers used to delay messages.

While evaluating the performance of our proposal we have used always messages of similar sizes. However, messages of different sizes result in different transmission delays among the same pair of processes, namely, due to fragmentation. It is thus interesting to consider an extension of our mechanism which takes this into consideration when adjusting delays.

It is also interesting to discuss the application of the proposed technique to algorithms other than the simple fixed sequencer algorithm presented. This requires that messages are disseminated to receivers before suffering the latency of ordering, excluding algorithms which delay dissemination until messages are ordered [15, 3, 2]. It is also required that the decided order is directly derived from the spontaneous ordering at some process, which is not true for causal history algorithms [18, 23, 10]. These requirements are satisfied by consensus based algorithms [7] as long as the coordinator for each instance of consensus is likely to be the same.

6. Conclusions

Although total order multicast is a convenient tool in programming fault-tolerant distributed systems, it exhibits higher latency than simple reliable multicast. This directly translates in increased latency for transactional clients of the database application with which we are concerned.

In this paper we propose a technique to improve the performance of optimistic total order multicast in wide area networks, which allows us to mask the latency of the ordering protocol. Our technique consists in compensating the differences in transmission delays and can easily be applied to an existing sequencer based protocol. This is achieved without additional messages and with minimal additional computational effort. As our proposal works only by introducing finite delays, the correctness of the original protocol developed in an asynchronous system model is not affected.

As far as we know, this is the only total order algorithm which allows an effective end-to-end latency of less than a round-trip delay without restriction of traffic pattern. Similar latency is possible with a moving sequencer when traffic is bursty or with causal history algorithms if the interarrival delay at each sender is less than the transmission delay. We also observe that the variance of latency observed by different clients of the system is reduced. This is interesting in applications such as stock trading [24] in which fair opportunities have to be offered for all clients.

Acknowledgments

References

[1] G. Alvarez and F. Cristian. Applying simulation to the design and performance evaluation of fault-tolerant systems. In *Symp. Reliable Distributed Systems*, 1997.

[2] G. Alvarez, F. Cristian, and S. Mishra. On-demand asynchronous atomic broadcast. In *Proc. the 5th IFIP Working Conf. Dependable Computing and Critical Applications*, Urbana-Champaign, IL, Sept. 1995.

[3] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Trans. Comput. Syst.*, 13(4), Nov. 1995.

[4] E. Anceaume. A lightweight solution to uniform atomic broadcast for asynchronous systems: proofs. TR PI-1066, IRISA, Nov. 1996.

[5] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3), Aug. 1991.

[6] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43, July 1996.

[7] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2), Mar. 1996.

[8] J. Cowie, D. Nicol, and A. Ogielski. Modeling the global Internet. *Comp. in Science and Eng.*, 1(1), Jan./Feb. 1999.

[9] X. Défago, A. Schiper, and P. Urbán. Totally ordered broadcast and multicast algorithms: A comprehensive survey. TR DSC/2000/036, EPFL, Switzerland, Sept. 2000.

[10] P. Ezhilchelvan, R. Macêdo, and S. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *Proc. the 15th Int'l Conf. on Dist. Comp. Syst.*, Los Alamitos, CA, USA, May 30–June 2 1995. IEEE CS Press.

[11] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 1985.

[12] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4), Apr. 1997.

[13] K. Guo. *Scalable Message Stability Detection Protocols*. PhD thesis, Cornell Univ., Computer Science, May 1998.

[14] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. TR TR94-1425, Cornell Univ., CS Dept., May 1994.

[15] J. Hickey, N. Lynch, and R. van Renesse. Specifications and proofs for Ensemble layers. In R. Cleaveland, editor, *5th Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, Berlin, 1999.

[16] M. Kaashoek and A. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proc. the 11th Int'l Conf. on Distributed Computing Systems ICDCS*, Washington, D.C., USA, May 1991. IEEE CS Press.

[17] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Proc. the Int'l Conf. on Dist. Comp. Syst.*, Austin, Texas, June 1999.

[18] L. Lamport. Time, clocks and the ordering of events in distributed systems. *Commun. ACM*, 21(7), 1978.

[19] R. Oliveira, J. Pereira, and A. Schiper. Primary-backup replication: From a time-free protocol to a time-based implementation. In *IEEE Int'l Symp. Reliable Dist. Syst.*, Oct. 2001.

[20] V. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, Univ. of CA, Berkeley, Apr. 1997.

[21] F. Pedone. *The Database State Machine and Group Communication Issues*. PhD thesis, EPFL, Switzerland, 1999.

[22] F. Pedone and A. Schiper. Optimistic atomic broadcast. In *Proc. the 12th Int'l Symp. on Dist. Computing*, Sept. 1998.

[23] L. Peterson, N. Buchholz, and R. Schlichting. Preserving and using context information in interprocess communication. *ACM Trans. Comput. Syst.*, 7(3), Aug. 1989.

[24] R. Piantoni and C. Stancescu. Implementing the Swiss Exchange Trading System. In *Proc. 27th Ann. Int'l Symp. Fault-Tolerant Computing (FTCS'97)*. IEEE, June 1997.

[25] F. Schneider. Replication management using the state-machine approach. In S. Mullender, editor, *Distributed Systems*, chapter 7. Addison Wesley, second edition, 1993.

[26] A. Sousa, F. Pedone, R. Oliveira, and F. Moura. Partial replication in the database state machine. In *IEEE Int'l Symp. Networking Computing and Applications*. IEEE CS, Oct. 2001.