Run-Time Switching Between Total Order Algorithms*

José Mocito and Luís Rodrigues

University of Lisbon
{jmocito,ler}@di.fc.ul.pt

Abstract. Total order broadcast protocols are a fundamental building block in the construction of many fault-tolerant distributed applications. Unfortunately, total order is an intrinsically expensive operation. Moreover, there are certain algorithms that perform better in specific scenarios and given network properties. This paper proposes and evaluates an adaptive protocol that is able to dynamically switch between different total order algorithms. The protocol allows to achieve the best possible performance, by selecting, in each moment, the algorithm that is most appropriate to the present network conditions. Experimental results show that, using our protocol, adaptation can be achieved with negligible interference with the data flow.

1 Introduction

A total order broadcast protocol is a fundamental building block in the construction of many distributed fault-tolerant applications [1]. Informally, the purpose of such a protocol is to provide a communication primitive that allows processes to agree on the set of messages they deliver and, also, on their delivery order. Uniform total order broadcast is particularly useful to implement fault-tolerant services by using software-based replication [2].

Unfortunately, the implementation of such a primitive can be expensive both in terms of communication steps and number of messages exchanged. This problem is exacerbated in large-scale systems, where the performance of the algorithm may be limited by the presence of high-latency links. Several total order protocols have been proposed that use different strategies to offer good performance [3]. There is no protocol that outperforms all others in all scenarios: each protocol offers best results under different load profiles and/or network conditions.

In this paper we describe and evaluate a total order protocol that combines different algorithms and adapts itself to the running environment. The protocol allows a fluid transition between algorithms, never stopping the flow of application messages. Such feature can be very useful in fault-tolerant safety- and mission-critical systems, like air traffic or nuclear plant control, where stoppages and/or significant delays imposed by adaptive mechanisms may be unacceptable.

* This work was partially supported by the IST project GORDA (FP6-IST2-004758).

TO1 - Total order: Let m_1 and m_2 be two messages that are *TO-broadcast*. Let p_i and p_j be any two correct processes that *TO-deliver*(m_1) and *TO-deliver*(m_2). If p_i *TO-delivers*(m_1) before *TO-delivers*(m_2), then p_j *TO-delivers*(m_1) before *TO-delivers*(m_2), and we note $m_1 < m_2$.
TO2 - Agreement: If a correct process in Ω has *TO-delivered*(m), then every correct process in Ω eventually *TO-delivers*(m).
TO3 - Termination: If a correct process *TO-broadcasts*(m), then every correct process in Ω eventually *TO-delivers*(m).
TO4 - Integrity: For any message m, every correct process *TO-delivers*(m) at most once, and only if m was previously *TO-broadcast* by some process $p \in \Omega$.

Table 1. Regular total order properties

We evaluate the performance of an implementation of the protocol and show how it can be optimized to induce a low overhead in resource consumption. Finally, we discuss how the protocol can be configured to operate using different classes of failure detectors.

The rest of the paper is structured as follows. Section 2 clarifies the properties of total order broadcast services. Section 3 describes the adaptive protocol. Performance evaluation results are presented in Section 4. Section 5 discusses optimizations that have been applied in the protocol implementation. Failure detection issues are addressed in Section 6. Section 7 concludes the paper.

2 Total Order Broadcast

Informally, total order broadcast is a group communication primitive that ensures that messages sent to a set of processes are delivered by all those processes in the same order. Such a primitive is useful, for example, in the implementation of fault-tolerant services [1], for instance, using the state machine approach (active replication) [4].

Total order broadcast is defined on a set of processes Ω by the primitives (1) *TO-broadcast*(m) which issues message m to Ω, and (2) *TO-deliver(m)* which is the corresponding delivery of m. When a process p_i executes *TO-broadcast*(m) (resp *TO-deliver(m)*), we say that p_i *"TO-broadcasts m"* (resp *"TO-delivers m"*). The total order primitive characterized by the properties listed in Table 1 is known as regular total order. A stronger version, called uniform total order [3], can also be defined. The difference among these definitions is not relevant for understanding our adaptive protocol, thus we will not delve further in this topic.

Many algorithms exist to implement total order. To give the reader an insight on the possible alternatives, we briefly introduce two of the most used ones, namely the *sequencer-site* [5] and the *symmetric* [6, 7] approach. Both methods have advantages and disadvantages.

In the sequencer-site approach one site is responsible for ordering messages on behalf of the other processes in the system. Sequencer-based protocols are appealing because they are relatively simple and provide good performance when message transit delays are small (they are particularly well suited for local area networks). However, in these protocols, a message sent by a process that is not the sequencer experiences a delivery latency close to $2D$, where D is the message transit delay between two system processes (i.e., the time to disseminate the message plus the time to obtain an order number from the sequencer process). Thus, sequencer-based approaches are inefficient in face of large network delays. Note that it is possible to design solutions where the sequencer role is rotated among processes [8].

In the symmetric approach, ordering is established by all processes in a decentralized way, using information about message stability. This approach usually relies on *logical clocks* [9] or *vector clocks* [10, 6]: messages are delivered according to their partial order and concurrent messages are totally ordered using some deterministic algorithm. Symmetric protocols have the potential for providing low latency in message delivery when all processes are producing messages. In fact, symmetric protocols can exhibit a latency close to $D + t$, where t is the largest inter-message transmission time [11]. Unfortunately, this also means that all (or at least a majority [7]) processes must send messages at a high rate to achieve low protocol latency.

Several other alternatives exist. For a comprehensive survey, the reader is referred to [3]. However, from the two examples above, it should be clear that it is interesting to have a protocol that can dynamically adapt to changes in the operation envelope by switching, in run-time, from one algorithm to another.

3 An Adaptive Protocol

We now present a protocol that is able to switch from a total order algorithm to another total order algorithm in response to changes in the operation envelope (such as changes in the workload, network conditions, number of participants, etc). In this paper we do not focus on the conditions that trigger adaptation, as these are highly application dependent (for a concrete scenario, see [12]). Instead, we are interested in finding a generic switching procedure that can switch from one algorithm to the other with minimum interference in the data flow.

Such protocol can be built from scratch using a monolithic approach where all the functionality of every total order algorithm is embedded in a single unity. A more modular (and generic) way of reaching the same goal is to (re-)utilize independent implementations of total order algorithms and build the adaptive behavior on top of them. In steady-state, the adaptive protocol would simply receive *TO-broadcast*/ *TO-deliver* requests/indications and forward them to the most appropriate algorithms.

To our knowledge, there is little work in the literature on how to efficiently perform this transition. Previous works on dynamic adaptation require messages to be buffered during the reconfiguration [13, 14], the message flow to be stopped

```
1: Initialization:
2:     deliv ← ∅
3:     undeliv ← ∅
4:     curAlg ← TO-A {current algorithm}
5:     newAlg ← ∅ {next alg.}
6:     switching ← false
7:     check[1..n] ← false

8: upon changeAlgorithm(newTO) do
9:     rBroadcast(switch,newTO)

10: upon rDeliver(switch,newTO) do
11:     newAlg ← newTO
12:     switching ← true
13:     TO-broadcast(curAlg,(flag,null,myself))

14: upon TO-deliver(curAlg,(flag,null,sender))
    do
15:     check[sender] ← true

16: upon check[1..n] = true do
17:     endSwitch()

18: upon TO-broadcast(msg) do
19:     TO-broadcast(curAlg,msg)
20:     if switching = true then
21:         TO-broadcast(newAlg,msg)

22: upon TO-deliver(alg,msg) do
23:     if alg = curAlg ∧ msg ∉ deliv then
24:         deliver(msg)
25:         deliv ← deliv ∪ {msg}
26:     else if msg ∉ deliv then
27:         undeliv ← undeliv ∪ {msg}

28: procedure endSwitch()
29:     for all msg ∈ undeliv ∧ msg ∉ deliv do
30:         deliver(msg)
31:         deliv ← deliv ∪ {msg}
32:     undeliv ← ∅
33:     check[i..n] ← false
34:     curAlg ← newAlg
35:     switching ← false
```

Fig. 1. Adaptive Total Order algorithm

in the current protocol [15], or some communication delay to be imposed during the transition between protocols [16]. Here we describe a generic transition protocol that does not require the traffic to be stopped, allowing a smooth adaptation to changes in the underlying network.

To be able to effectively transition from one algorithm to the other, all nodes need to agree on the point in the message flow where they switch. Also, both algorithms must provide FIFO ordering of messages (which is the most common case). The rational behind our proposal is to start broadcasting messages using both total order algorithms, during the switching phase, until a safe point is reached in every process. By using both algorithms simultaneously, no stoppage in the message flow is necessary. The protocol is listed in Figure 1.

Let us assume that the adaptation protocol is using algorithm TO-A to order messages and wants to switch to algorithm TO-B. The transition protocol works as follows. A control message is broadcast to all processes to initiate the reconfiguration (lines 8–9). When a node receives this message (line 10) it starts broadcasting messages using both total order algorithms. Also, the first message it broadcasts using algorithm TO-A is flagged. If no message is to be sent, then a flagged special null message is broadcast using TO-A, to allow faster protocol termination (flagged first message is not represented in the algorithm to preserve clarity). When a process starts receiving messages from both TO algorithms it performs the following steps (lines 22–27): messages received from TO-A are delivered as normally; messages received from TO-B are buffered in order. As soon as a flagged message is received from each and every node (line 15) the transition is concluded using the following "sanity" procedure (lines 28–35). Firstly, all messages received from TO-B that have not yet been delivered by

TO-A are delivered in order. Finally, from this point on, all messages received from TO-A are simply discarded and no further message is sent using TO-A (until a new reconfiguration is needed). The TO-B algorithm is then used to broadcast and receive all the messages to be delivered.

Note that, after the transition is concluded, messages received from TO-B are delivered only if they have not been already received and delivered from TO-A (line 23). This is a necessary safeguard as the two total order algorithms do not necessarily deliver messages in the same order, nor at the same time. So there is a possibility that a message that has already been delivered from TO-A is received after the termination of the reconfiguration procedure from TO-B.

Also, the protocol presented does not allow concurrent adaptations. For one adaptation to happen, the previous (if any) should always have concluded.

4 Performance Evaluation

We evaluate the performance of our adaptive protocol from two different perspectives. First, we evaluate the overhead of the switching procedure. Then, we provide a comparative analysis on how different switching strategies interfere with the traffic flow during the reconfiguration.

4.1 Switching Overhead

To evaluate the switching overhead of our adaptive protocol we compare the performance of a system that always uses the same total order algorithm, with that of a system that is periodically switching between two algorithms. To make the comparison as fair as possible, we made our protocol switch between two instances of the same total order algorithm, which is also used as the non-adaptive protocol. Also, the network topology and working conditions did not change during the tests. In this way, we can isolate the cost of the switching procedure given that all the remaining factors remain unchanged.

The adaptive protocol was implemented in Java using the *Appia* [17] protocol composition and execution framework. The experiments were conducted in the SSFNet [18] network simulator and the scenario consists of a five node cluster, where all nodes are connected to each other by 100Mbps bi-directional links.

Two runs of the same experiment were performed: (A) one using a single total order protocol (non-adaptive), (B) and another using the proposed adaptive total order protocol, which is forced to switch periodically. Each run consists of every node broadcasting 5000 messages of 5KB in total order. The experiment ends when all nodes receive all the broadcast messages. The values presented are averages of the measurements conducted in each node.

Figure 2 presents the overall throughput results when the send rate is made variable. As depicted, both total order algorithms perform the same until they reach approximately 400 msg/s. After this point, the throughput of the non-adaptive protocol continues to grow while its value stabilizes for the adaptive protocol. This behavior is explained by the overhead introduced by the switching

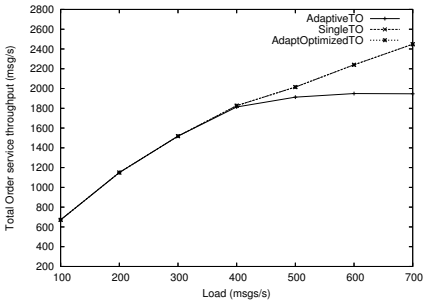

Fig. 2. TO throughput in non-adaptive, adaptive and optimized algorithms

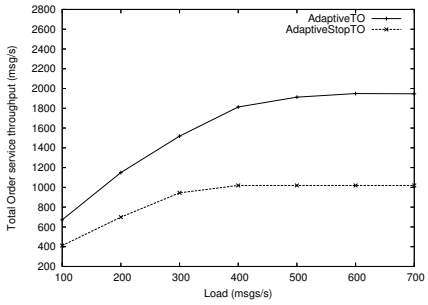

Fig. 3. TO throughput in adaptive and stop algorithms

phase in the adaptive protocol. During this phase, the same set of messages is being broadcast by two total order algorithms at the same time, leading to an increase (approximately double) in the bandwidth usage. If the send rate is too high, the available bandwidth can be exhausted, leading to the stagnation observed in the throughput.

Thus, we can conclude that our switching protocol offers negligible overhead as long as there is enough network bandwidth to support the transmission of data in parallel during the reconfiguration. When the protocol operates close to the available bandwidth, the switching procedure introduces an overhead. This limitation can be addressed at the implementation level, by sending the payload of the messages using just one of the two algorithms. This optimization is described in Section 5 and its switching overhead is also depicted in Figure 2.

4.2 Comparative Analysis

As we noted in Section 1, most switching protocols require the message flow to be stopped in order to terminate the reconfiguration process. By not imposing a gap in the message flow, our protocol provides smooth transitions between algorithms, thus allowing applications that rely in its services to normally execute, even during the switching phase. Therefore, it should offer better overall throughput, as long as enough bandwidth is available to cope with the demand imposed by the transmission of messages using two algorithms at the same time. The same experiment described in 4.1 was conducted using a protocol that stops the message flow. This protocol operates by sending a stop request to all nodes and awaiting for a confirmation from each of these nodes. After confirming the stop request a node does not send further messages until the switch is complete. The performance of such protocol when compared to our proposal can be observed in Figure 3, which clearly shows that our approach always performs better.

Other protocols that try to minimize the cost of switching between algorithms have also been proposed. A previous work [16], proposes a solution that has some

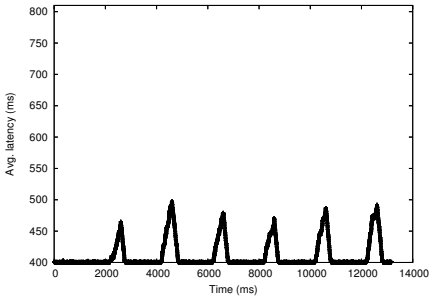

Fig. 4. Latency in Adaptive TO

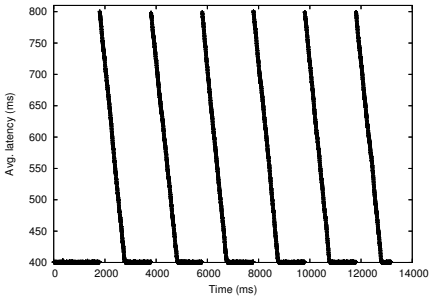

Fig. 5. Latency in RABP

similarities with our protocol, but differs from it by not requiring every node to wait for a "special" (in our algorithm the term is "flagged") message from every other node, and also for not making any assumptions about the failure model where it is executing (see Section 6). In [16], a special reconfiguration message is broadcast in total order. When a node receives such message, it stops the flow in the current algorithm, and re-issues all his undelivered messages in the next algorithm. It then starts using it to broadcast messages in total order. We will refer to this protocol by RABP (*Replacement of the Atomic Broadcast Protocol*).

The RABP strategy has the advantage of requiring less bandwidth during the switching phase. However, some delay is imposed to the message flow during the retransmission of the undelivered messages. To observe this side effect, the experiment was now conducted using our protocol and the RABP protocol. In Figures 4 and 5 we can observe how both compare in terms of latency. The spikes depicted correspond to the switching phases, in the time-line of the experiment. The inter-arrival time of messages was also measured and its evolution is shown in Figures 6 and 7. Finally, the number of messages delivered by a fixed period of time (10 ms) was also observed and the comparative results are depicted in Figures 8 and 9.

This experiment clearly showed that our proposal is able to keep a sustained delivery rate during the switching phase and performs similarly to RABP during the remaining time. By not significantly delaying the message flow, our protocol can best suit environments where application stoppage, due to significant communication delays, is not desirable.

5 Implementation Optimization

When enough bandwidth is available, the (non-optimized) version of our protocol already implements the switching procedure with negligible overhead in the message flow. However, the experimental results provided in Section 4 showed that during the switching phase, when both protocols are being used to broadcast the same set of messages, the available bandwidth can be exhausted when the send rate and/or message payload is too high.

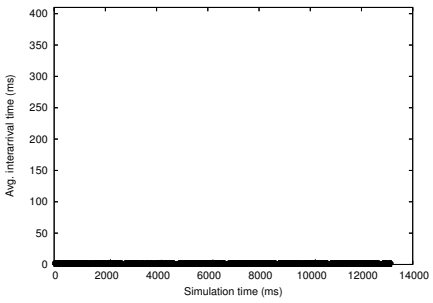

Fig. 6. Inter-arrival time in Adaptive TO

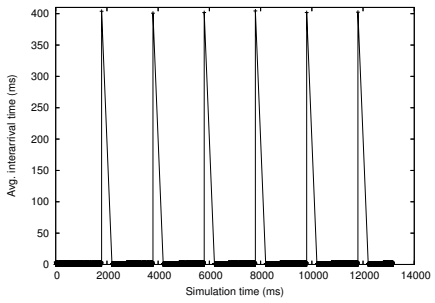

Fig. 7. Inter-arrival time in RABP

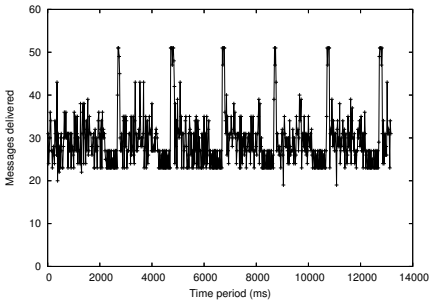

Fig. 8. Delivery rate in Adaptive TO

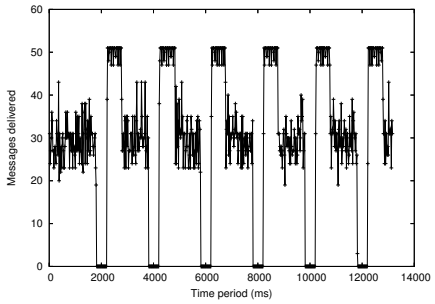

Fig. 9. Delivery rate in RABP

To overcome this problem we now describe an optimization to reduce the amount of data being transmitted by the adaptive protocol during this phase. The optimization consists of broadcasting using the first (and current) algorithm only the identifiers of the messages being transmitted. The messages payload is only transmitted using the second algorithm. In this manner, the amount of redundant information transmitted over the wire is reduced substantially. This optimization has a minor drawback: the protocol cannot deliver a message to the application until it is received by both total order algorithms. However, since both algorithms are executed in parallel, the impact of this feature is negligible.

Figure 2 shows that the optimization allows the protocol to continue increasing its throughput after the point where the non-optimized version stabilizes (approximately 400 msg/s), showing a behavior similar to the non-adaptive protocol (note that the lines for the optimized and the non-adaptive algorithms overlap in the figure).

6 On Failure Detection

To simplify the description of our protocol, in Section 3 we have not addressed the issue of failure detection during the switching protocol. Namely, we have stated that the protocol moves to the sanity step when it receives a flag from

every participant (Figure 1, line 16). Without further changes, the protocol would simply block in the presence of a single failure. We now discuss how our protocol can be adapted to operate in the presence of faults. Our algorithm can operate in asynchronous systems augmented with failure detectors [19].

We start by discussing the operation of the protocol in a system augmented with a *Perfect Failure Detector* ($\mathcal{P}$) [19], i.e., a system where processes fail by crashing and crashes can be accurately detected by all correct processes. In this model, the transition condition should be set to "a flag is received by all *correct* processes". This model is actually used in all of our implementations, where the failure detection is encapsulated by a view-synchronous interface [20].

The protocol can also be modified to operate in an asynchronous system augmented with an unreliable failure detector (such as the $\diamond S$ failure detector proposed in [19]) as long as a majority of processes do not fail (naturally, in this case, the underlying total order algorithms, must also be designed for such a model). In this model, the transition condition should be set to "a flag is received by a majority of processes". However, in this configuration, correct processes that do not belong to the majority may be required to retransmit some messages. It is interesting to observe that the strategy proposed before for the $\mathcal{P}$ detector (perform the switch when a flag is received from *all* correct processes) and the strategy proposed in [16] (perform the switch when the *first* flag is received) can be seen as extreme point of a spectrum. Between these extreme cases, there is a range of alternative switching points, from which the "majority of processes" is the one that ensures less disruption in $\diamond S$ model.

7 Conclusions and Future Work

Several total order protocols exist that use quite different strategies. Such strategies may perform better in specific environments and/or working conditions. We presented an adaptive total order protocol that is able to switch in run-time between different total order algorithms. When the environment is dynamic, this allows the system to use the ordering strategy that is most favorable.

If one is not careful, the procedure to switch between algorithms can disrupt the message flow. Our work tackles this issue by proposing a novel switching strategy that performs the reconfiguration with negligible impact on the observed delivery rate. Evaluation results of an implementation of the protocol showed performance improvements in regard to competing approaches.

Planned future work on this subject will aim at embedding the resulting protocol in a database replication service based on the state machine approach [4].

References

1. Powell, D., ed.: Special Issue on Group Communication. Number 4 in 39. In: Communications of the ACM. ACM (1996) 50–97
2. Guerraoui, R., Schiper, A.: Software-based replication for fault tolerance. IEEE Computer **30**(4) (1997) 68–74

3. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Computing Surveys **36**(4) (2004) 372–421
4. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Computing Surveys **22**(4) (1990) 299–319
5. Kaashoek, M., Tanenbaum, A.: Group communication in the Amoeba distributed operating system. In: Proceedings of the 11th International Conference on Distributed Computing Systems, IEEE (1991) 222–230
6. Peterson, L., Buchholz, N., Schlichting, R.: Preserving and using context information in interprocess communication. ACM Transactions on Computer Systems **7**(3) (1989) 217–146
7. Dolev, D., Kramer, S., Malki, D.: Early delivery totally ordered multicast in asynchronous environments. In: Digest of Papers, The 23th International Symposium on Fault-Tolerant Computing, IEEE (1993) 544–553
8. Chang, J., Maxemchuck, N.: Reliable broadcast protocols. ACM, Transactions on Computer Systems **2**(3) (1984)
9. Lamport, L.: Time, clocks and the ordering of events in a distributed system. Communications of the ACM **21**(7) (1978) 558–565
10. Birman, K., Joseph, T.: Reliable communication in the presence of failures. ACM, Transactions on Computer Systems **5**(1) (1987)
11. Rodrigues, L., Fonseca, H., Veríssimo, P.: Totally ordered multicast in large-scale systems. In: Proceedings of the 16th International Conference on Distributed Computing Systems, Hong Kong, IEEE (1996) 503–510
12. Rodrigues, L., Mocito, J., Carvalho, N.: From spontaneous total order to uniform total order: different degrees of optimistic delivery. In: Proceedings of the 21st ACM symposium on Applied computing (SAC'06) *(to appear)*, ACM Press (2006)
13. Liu, X., van Renesse, R.: Fast protocol transition in a distributed environment. In: Proceedings of the 19th ACM Conference on Principles of Distributed Computing (PODC 2000), Portland, OR (2000) 341
14. Chen, W.K., Hiltunen, M., Schlichting, R.: Constructing adaptive software in distributed systems. In: ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems, Washington, DC, USA, IEEE Computer Society (2001) 635
15. van Renesse, R., Birman, K., Hayden, M., Vaysburd, A., Karr, D.: Building adaptive systems using Ensemble. Software: Practice and Experience **28**(9) (1998) 963–979
16. Rutti, O., Wojciechowski, P., Schiper, A.: Structural and algorithmic issues of dynamic protocol update. In: Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06), IEEE (2006)
17. Miranda, H., Pinto, A., Rodrigues, L.: Appia, a flexible protocol kernel supporting multiple coordinated channels. In: Proceedings of the 21st International Conference on Distributed Computing Systems, Phoenix, Arizona, IEEE (2001) 707–710
18. Nicol, D., Liu, J., Liljenstam, M., Yan, G.: Simulation of large-scale networks using SSF. In: Proceedings of the 2003 Winter Simulation Conference. (2003)
19. Chandra, T., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43**(2) (1996) 225–267
20. Birman, K., Joseph, T.: Exploiting virtual synchrony in distributed systems. Technical Report 87-811, Department of Computer Science, Cornell University, Ithaca, New York (1987)