An Algorithm for an Intermittently Atomic Data Service Based on Group Communication

Roger Khazan[*] and Nancy Lynch[†]
rkh@mit.edu, lynch@lcs.mit.edu

I. Introduction

Group communication provides a convenient mechanism for building applications that maintain a replicated state of some sort [1]. Such applications typically implement certain consistency properties regarding how different application clients perceive the replicated state. One of the well-understood and often preferred consistency properties is *strong-consistency* or *atomicity* [2], [3, Ch. 13], which creates a perception of the replicated state as being non-replicated. However, in large-scale wide-area settings, providing atomicity may result in prohibitive levels of overhead and sometimes, during network partitions, in blocking the applications until partitions repair. Thus, weaker consistency properties may be more appropriate in such settings. A key question is which weaker consistency properties are still coherent and acceptable to the application clients.

In this paper, we propose one way to weaken the atomicity property and illustrate the resulting property, *intermittent atomicity*, with a simple application. We call this application *intermittently-atomic data service* (IADS); the application implements a variant of a data service that allows a dynamic group of clients to access and modify a replicated data object. The IADS application is prototypical of some collaborative computing applications, such as a shared white-board application (e.g., [4], [5]).

Roughly speaking, intermittent atomicity guarantees that, while the underlying network component is stable, clients perceive the data object as *atomic*.[1] During periods of instability, the clients' perceptions of the object may diverge from the atomic one. The non-atomic semantics may persist until after the underlying component becomes stable again. When stability is regained, the atomic semantics is restored within some finite amount of time: The clients comprising the stable component are informed about the current membership of the client group and the new state of the data object. The new state is computed as an application-specified merge of the states of the members' object replicas. From that point on while stability lasts, the clients again perceive the object as an atomic one. A formal definition of intermittent atomicity is presented in [7, Ch. 10] (where intermittent atomicity is called *interim atomicity*).

The IADS application can be conveniently built using standard group communication mechanisms. We demonstrate this by presenting a simple algorithm, IADS, that operates atop a group communication service, GCS, which we assume satisfies the formal design specification of [8], [7]. The algorithm follows the *active replication*/*state-machine* approach [9], [10] and utilizes the state-transfer protocol of Amir, et al. [11]. The Virtual Synchrony semantics provided by GCS allows the application to sometimes avoid state-transfer when views change and also to reduce the number of state messages exchanged during a state-transfer protocol. The set of group members that transitions together from $\mathtt{v}$ to $\mathtt{v}'$ is known as the transitional set $\mathtt{T}$ of $\mathtt{v}$ and $\mathtt{v}'$ [1]. The Virtually Synchronous Delivery property guarantees that every server in $\mathtt{T}$ receives the same set of messages while in view $\mathtt{v}$, before receiving view $\mathtt{v}'$ and set $\mathtt{T}$ from GCS. Thus, if the object replicas of $\mathtt{T}$ were mutually consistent upon entering normal mode in view $\mathtt{v}$, they remain mutually consistent when view $\mathtt{v}'$ is delivered. This leads to two observations: First, it is enough for only one member of $\mathtt{T}$ to communicate the state of its replica during state-transfer protocol. Second, state-transfer is unnecessary in situations when the entire membership of the new view $\mathtt{v}'$ has transitioned together from view $\mathtt{v}$ (i.e., $\mathtt{v.set} = \mathtt{T}$).

II. Application Description

The `Obj` Data Type: The IADS application manages deterministic data objects whose serial behavior is specified by some data type, `Obj`. The `Obj` data type defines possible states of the objects and operators on the objects; it is defined similarly to the variable type of [3]. Formally, the `Obj` type consists of: a) a set S of

[*] MIT Lincoln Laboratory. 244 Wood St., Lexington, MA 02420, USA. This work was partially sponsored by the Department of the Air Force under the Air Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are not necessarily endorsed by the US Government. [†] MIT Laboratory for Computer Science. 200 Technology Square, Cambridge, MA 02139, USA.

[1]Atomic objects are also known as *linearizable* [6], *strongly-consistent, non-replicated,* and *one-copy equivalent*.

object states; b) a distinguished initial state $s_0 \in S$; c) a set R of *response values*; and d) a set O of operations, each of the type $S \to (S \times R)$. Furthermore, we assume an application-defined function `merge`: $\mathtt{SetOf}(\mathtt{Proc} \times S) \to S$. This function is used during state-transfer to compute a new, common, state of the object based on, possibly different, states of the participating object replicas. We assume that the `merge` function has the identity property, i.e., $\mathtt{merge}(\{\langle \mathtt{p}_1, \mathtt{x}\rangle, \langle \mathtt{p}_2, \mathtt{x}\rangle, \ldots, \langle \mathtt{p}_k, \mathtt{x}\rangle\}) = \mathtt{x}$. For simplicity we assume that the application manages a single data object and all the operations requested by clients pertain to this object.

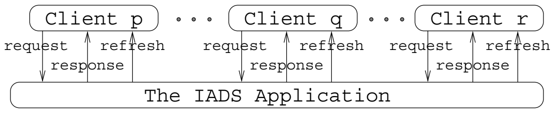

Fig. 1. Interaction of the application with its clients.

Application Interface: The interface between the IADS application and its clients consists of the typical `request`, and `response` actions: The application receives client p's request to process operation $\mathtt{o} \in O$ via input action $\mathtt{request}_\mathtt{p}(\mathtt{o})$, and it eventually responds to the operation via $\mathtt{response}_\mathtt{p}(\mathtt{o}, \mathtt{r})$, where $\mathtt{r} \in R$ is the return value resulting from applying operation o to the underlying data object. In addition to the `request`/`reply` actions, the interface with client p includes special $\mathtt{refresh}_\mathtt{p}(\mathtt{set}, \mathtt{x})$ actions, where $\mathtt{set} \in \mathtt{SetOf}(\mathtt{Proc})$ and $\mathtt{x} \in S$. The application uses these actions to refresh the client's perception of its collaboration group (set) and the state of the underlying data object (x). For simplicity, we do not include the `join` and `leave` actions as part of the interface. Such actions can be processed by the group communication service as requests to join or leave a specified application group.

Application Semantics: Chapter 10 of [7] contains a formal specification of IADS. Among the specified properties, there are some basic ones that are not specific to intermittent atomicity; these include properties such as correspondence between requests and responses, and processing of requests submitted by a given client in gap-free FIFO order. The properties that *are* specific to intermittent atomicity are Stabilization and Interim Atomicity. The Stabilization property is a liveness property that requires IADS to eventually stabilize after a set of clients becomes stable. The Interim Atomicity property is a combination of safety and liveness; it requires IADS to behave as an atomic data service in situations when IADS is stable.

III. GCS-BASED ALGORITHM

The algorithm for the intermittently-atomic data service follows the standard active replication (state-machine) approach [9], [10] and utilizes the state-transfer protocol of Amir et al. [11]. The algorithm is composed of a collection of application end-points, which run the same algorithm. The application end-points operate as clients of the GCS system — as members of the same process group. Figure 2 shows interaction of an application end-point with its client and with the underlying GCS end-point.

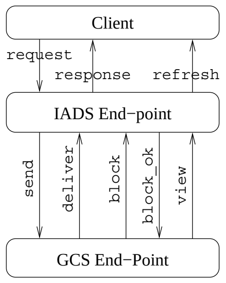

Fig. 2. Application design architecture: An application end-point interacts with its client and the underlying GCS end-point.

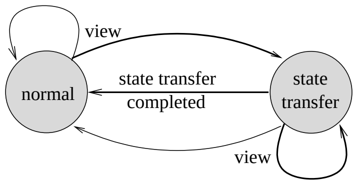

Fig. 3. Application modes of operation: `view`-labeled transitions leading to `normal` mode correspond to circumstances when an application end-point avoids state-transfer by relying on the GCS semantics.

Every application end-point maintains a replica of the data object. The object replicas are modified during normal mode of operation when clients' requests are processed, and as a result of state-transfer when a new state of the object is computed from the merge of the object replicas of different application end-points.

Figure 3 depicts a state-transition diagram that governs transitions between normal and state-transfer modes. Initially, the mode is normal. An application end-point may switch from normal to state-transfer when it receives a new view from GCS; in some situations, discussed below, the application end-point is able to rely on the guarantees provided by GCS to avoid state-transfer and remain in normal mode. When an application end-point completes state-transfer, it switches back to normal mode. If GCS delivers a new view before the application end-point completes state-transfer, the application end-point typically remains in state-transfer mode, but in some situations, discussed below, it may again rely on the guarantees provided by GCS to immediately switch to normal mode.

Figure 4 contains an I/O automaton, IADS_p, modeling the application end-point at process p (see [3] for the definition of I/O automata). The application end-point maintains a replica, obj, of the data object. Requests submitted by the client are placed into a queue, inp, and later multicast using GCS to the application end-points comprising the current view. GCS delivers these requests within the same view and in FIFO order. The application end-points append the requests delivered by GCS into a queue, $\text{ops}[q]$, according to the sender q. The requests stored in the ops queues are processed during normal mode, according to a total order on all requests communicated in a given view; as we explain below, the algorithm establishes this total order by tagging requests with logical timestamps [9] (see also [10] and [3, page 607]).[2] Processing of requests is done by an internal action do and as a result of receiving a view input from GCS. When an application end-point processes an operation request, it applies the operation to its object replica. If the operation request that is being processed was submitted by the application end-point's own client, the application end-point places the operation and the resulting return value into an output queue, out, to be later reported to the client.

Consider application end-points belonging to some stable view. Assume that, at the times when the application end-points start processing requests in the view, the states of their object replicas are the same. In order for their object replicas to stay mutually consistent, the object replicas should undergo the same modifications in the same order at different application end-points; that is, different application end-points should apply the same sequences of operations to their object replicas.

Total order through logical time: The IADS algorithm establishes a total ordering on all requests communicated through GCS in a given view using logical timestamps, as in [9] and [3, Sec. 18.3.3]. We chose this well-known, symmetric approach for the sake of concreteness and simplicity. Many different algorithms for implementing totally ordered multicast exist and can be used here, including those that focus on scalability and efficiency in WANs (see [12]).

Application end-point p maintains an integer $\text{lt}[p]$ which corresponds to p's logical time within the current view; the initial value of $\text{lt}[p]$ is 0 and it is reset to 0 when p receives view inputs. When p starts processing a request by multicasting it to other application end-points, p increments $\text{lt}[p]$ and tags the request with the timestamp. Also, whenever p receives a request tagged with a timestamp ts from some application end-point q, p updates its logical time to $\max(\text{lt}[p], \text{ts}) + 1$. The total order on messages communicated within the same view is defined as their ordering by the "timestamp, application end-point identifier" pair. That is, o_p *totally precedes* o_q if and only if $((\text{ts}_p < \text{ts}_q) \vee ((\text{ts}_p = \text{ts}_q) \wedge (p < q))$, where o_p is an operation sent by p and tagged with timestamp ts_p, and o_q is an operation sent by q in the same view as o_p and tagged with timestamp ts_q. Note that this total order is consistent with both FIFO and causal orderings of requests and responses.

Normal mode: In normal mode, application end-point p processes the operations in its ops queues according to the total order defined above. Internal action $\text{do}_p(q, o)$ models processing of an operation o submitted by client at q. Operation o is processed if

(a) operation o totally precedes all other operations currently in the ops queues; and

(b) p knows that the logical times of all other application end-points in the view exceed the timestamp associated with o.

Condition (b) guarantees that p has received all the operations that precede o, and thus, o is the next one in the totally ordered sequence of operations sent in the current view; see [3, Sec. 18.3.3]. The algorithm implements condition (b) by keeping track of the known logical time of every application end-point q in the current view. The application end-point updates $\text{lt}[q]$ whenever it receives operation requests sent by q. In addition to communicating the values of their logical times through operations, application end-points also let others know of their logical times by periodically sending special heartbeat messages, $\langle \textbf{`lt\_msg'}, \text{ts} \rangle$.

While the current view remains stable, the application end-points process the same sequences of operations, and thus, remain mutually consistent.

When an application end-point receives a new view from GCS, the application end-point processes all of the operations in its ops queues according to the total order, even though condition (b) may not hold for these messages. The sequence of operations processed at this point may diverge from the global sequence because the application end-point may have received only a subset of all of the operations sent in the current view; for example, it may be missing some of the operations sent by disconnected application end-points. However, what is guaranteed by GCS is that members of the transitional set of the new view receive the same set of messages, and hence process the same sequence, if they receive the new view. Thus, after processing the operations in their ops queues, the members of the transitional set have the same states of their object replicas.

[2]Note that we implement total ordering within the application algorithm to make it easier to visualize how the algorithm works. In general, however, total ordering would be implemented as a separate layer, above GCS and below the application.

AUTOMATON IADS$_p$

Type:

```
AppMsg  = (O × Int) ∪ Int ∪ (SetOf(Proc) × S)        // operations, heartbeats, and state-transfer
OutType = (O × R) ∪ (SetOf(Proc) × S)                // operation replies and refresh information
```

Signature:

```
 Input:    request_p(o), O o                         Output:   response_p(o, r), O o, R r
           gcs.deliver_p(q, m), Proc q, m ∈ AppMsg             refresh_p(s, x), SetOf(Proc) s, S x
           gcs.view_p(v, T), View v, SetOf(Proc) T             gcs.send_p(m), m ∈ AppMsg
           gcs.block_p()                                       gcs.block_ok_p()
 Internal: do_p(q, o), Proc q, O o
```

State:

```
S obj, initially s_0                                   View myview, initially v_p
QueueOf(O) inp, initially empty                        Bool mode ∈ {normal, st}
QueueOf(OutType) out, initially empty                  Bool send_state, initially false
(∀ q ∈ Proc) Int lt[q], initially 0                    SetOf(Proc)  SS, initially {p}
(∀ q ∈ Proc) QueueOf(O × Int) ops[q], initially empty  SetOf(Proc × S) StatesV, initially empty
block_status ∈ {unblocked, requested, blocked},        SetOf(Proc) States_Await, initially {}
               initially unblocked
```

Transitions:

```
INPUT  request_p(o)
eff: append o to inp

OUTPUT  gcs.send_p(⟨'op_msg', o, ts⟩)
pre: block_status ≠ blocked
     o = First(inp)  ∧  ts = lt[p] + 1
eff: remove o from inp
     lt[p] ← lt[p]+1
     append ⟨o, ts⟩ to ops[p]

INPUT  gcs.deliver_p(q, ⟨'op_msg', o, ts⟩)
eff: if (q ≠ p) then
         lt[q] ← ts
         lt[p] ← max(lt[p], ts) + 1
         append ⟨o, ts⟩ to ops[q]

INTERNAL  do_p(q, o)
pre: mode = normal
     q = min{t ∈ myview.set : First(ops[t]).ts =
         = (min{First(ops[r]).ts : r ∈ myview.set})}
     (∀ t ∈ myview.set) lt[t] > First(ops[q]).ts
     ⟨o, ts⟩ = First(ops[q])
     let x and r be s.t. ⟨x, r⟩ =  o(obj)
eff: remove ⟨o, ts⟩ from ops[q]
     obj ← x
     if(p = q) then append ⟨o, r⟩ to out

OUTPUT  response_p(o, r)
pre: ⟨o, r⟩ is first on out
eff: remove ⟨o, r⟩ from out

OUTPUT  refresh_p(set, x)
pre: ⟨set, x⟩ is first on out
eff: remove ⟨set, x⟩ from out

OUTPUT  gcs.send_p(⟨'lt_msg', ts⟩)
pre: block_status ≠ blocked  ∧  ts = lt[p]

INPUT  gcs.deliver_p(q, ⟨'lt_msg', ts⟩)
eff: if (q ≠ p) then
         lt[q] ← ts
         lt[p] ← max(lt[p], ts) + 1
```

```
INPUT  gcs.block_p()
eff: block_status ← requested

OUTPUT  gcs.block_ok_p()
pre: block_status = requested
eff: block_status ← blocked

INPUT  gcs.view_p(v, T)
eff: // process all operations in the ops queue
     while (∃ q) such that
     (q = min{t ∈ myview.set : First(ops[t]).ts =
          = (min{First(ops[r]).ts : r∈myview.set})})
        remove first element ⟨o, ts⟩ from ops[q]
        ⟨obj, r⟩ ←  o(obj)
        if(p = q) then append ⟨o, r⟩ to out
     end
     (∀ t ∈ myview.set) lt[t] ← 0
     myview ← v
     block_status ← unblocked
     // state transfer decision:
     (mode = normal ? SS ← T : SS ← SS ∩ T)
     if(v.set = SS) then   // normal mode
         append ⟨v.set, obj⟩ to out
         mode ← normal
     else                  // state-transfer
         State_Await ← v.set; StatesV ← empty
         send_state ← (p = min(SS))
         mode ← st

├──────────────┤ state transfer ├──────────────┤

OUTPUT  gcs.send_p(⟨'st_msg', set, x⟩)
pre: block_status = unblocked
     send_state = true  ∧ ⟨set, x⟩ = ⟨SS, obj⟩
eff: send_state ← false

INPUT  gcs.deliver_p(q, ⟨'st_msg', set, x⟩)
eff: (∀ t ∈ set) add ⟨t, x⟩ to StatesV
     States_Await ← States_Await - set
     if (States_Await = {}) then
       obj ←  merge(StatesV)
       append ⟨myview.set, obj⟩ to out
       mode ← normal
```

Fig. 4.   Application end-point IADS$_p$ of an Interim-Atomic Data Service.

After the operations are processed, application end-point p decides whether or not to enter the state-transfer protocol. Variable SS is used for keeping track of the set of application end-points whose object replicas are synchronized with p; according to [11], SS is computed as the intersection of all the transitional sets delivered since normal mode. If SS is the same as the membership of a new view v, then everyone in the new view is already synchronized and p does not need to participate in a state-transfer protocol for view v; it may resume its normal mode of operation. Otherwise, p enters the state-transfer protocol (see below).

The following two paragraphs explain why computing SS as the intersection of all the transitional sets delivered since normal mode makes sense.

Consider an application end-point p that receives $\texttt{gcs.view}_{\texttt{p}}(\texttt{v}, \texttt{T})$ while in view $\texttt{v}^-$, and assume that p's mode is normal prior to receiving the new view. If the membership v.set of the new view is the same as transitional set T, then all of the members of v enter the new view directly from $\texttt{v}^-$ (provided they do enter v). The Virtual Synchrony semantics guarantees that these members have received the same sets of messages while in view $\texttt{v}^-$, and hence have applied the same operations in the same order to their object replicas. Since the states of the object replicas of the members of T were the same when they began normal mode in view $\texttt{v}^-$, their object replicas are the same after receiving view v from GCS.

As an alternative, consider a situation in which the application end-point p receives $\texttt{gcs.view}_{\texttt{p}}(\texttt{v}, \texttt{T})$ while already engaged in state-transfer in view $\texttt{v}^-$. Even though all the application end-points may be transitioning together from $\texttt{v}^-$ to v, it may be the case that these application end-points had inconsistent object replicas prior to entering view v. Since the state-transfer protocol was interrupted, they did not have a chance to synchronize their object replicas. Thus, it is not sufficient to simply consider transitional set T. The intersection of the current SS set and T yields the set of application end-points that a) were synchronized when they switched from normal mode to state-transfer, and b) have been synchronized since then.

State-transfer protocol: The state-transfer protocol involves each end-point collecting the states of the object replicas of the members of the new view, and then computing a new state for its replica as a merge of the collected states. After the object replica is updated with the result of the merge, the refresh information is placed on the out queue. The refresh information contains the new membership set and the new state of the object.

The GCS semantics allows us to reduce the number of messages and the amount of information communicated during the protocol: Only one end-point among the members of SS needs to send the state of its object replica to others. This is because our algorithm maintains a property that after receiving a view, all members of SS have their object replicas in the same state. The optimization is important because state-transfer messages are typically "costly" due to their large size.

The state-transfer protocol in Figure 4 follows the algorithm of [11]. Boolean variable send_state controls whether end-point p is the one that has to send its object replica's state on behalf of the end-points in set SS. Set StatesV is used for collecting object replicas' states of the view members, and set States_Await is used for keeping track of the list of end-points from whom p has not yet received a state-transfer message.

Note that end-points keep multicasting their clients' operations to one another in the new view, in parallel with the state-transfer protocol. The only part of the algorithm that is blocked during state-transfer is the actual processing of the operations. When state-transfer completes, the end-point may be able to process a whole bunch of operations collected in the ops queues right away, by executing a sequence of do actions.

If the state-transfer protocol is interrupted by a delivery of a new view, the end-point, as before, processes all of the operations in its ops queues according to the total order, and then decides whether to re-start a state-transfer protocol or to switch back to normal mode.

IV. Conclusions

A proof of the algorithm's correctness and two theoretical performance analysis results are presented in [7, Ch. 10]. One of the performance results deals with how quickly the IADS algorithm processes requests and delivers responses to its clients. The algorithm is able to process a given request as soon as it determines the request's position in the totally-ordered sequence of all the requests communicated in the current view and as soon as it receives and processes all the preceding requests. This time depends on the specific algorithm used for totally ordering requests. In general, the performance of state-of-the-art total order algorithms, in situations when the underlying network is well-behaved, is close to a single message latency.

To put this result in a larger context, an alternative approach to using Group Communication for building replicated data services is to use Consensus (e.g., [13], [10]). In this approach, during normal mode of operation, the servers hosting object replicas run Consensus to agree on the order in which to process clients' requests. For the data services that provide weaker consistency semantics, such as the IADS application, using Consensus is an overkill. Optimized solutions based on Consensus require two round-trip messages: the original message to the leader, a “query” message from the leader to the end-points and back, and then the “decision” message from the leader to the end-points [13]. In contrast, the IADS algorithm is able to process requests in the time it takes to totally-order them, which in the case of a leader-based scheme requires only one round-trip time to the leader. In addition, unlike Consensus, Group Communication provides convenient mechanisms to support partitionable semantics and dynamic sets of clients.

The second performance result in [7] expresses how quickly IADS reconfigures and delivers `refresh` inputs when instabilities occur. Roughly speaking, the performance theorem proved in [7] states that, when used in conjunction with a WAN-oriented group communication service described in [8], [7], every application end-point delivers the final refresh event within about two latencies – one detection latency and one message latency – away from the time the final group event occurs, provided the application end-point is able to rely on Virtual Synchrony to avoid participating in a state-transfer protocol. Otherwise, if state-transfer is necessary, the final refresh event occurs within about three latencies – one detection latency and two message latencies – away from the time the final group event occurs.

Acknowledgments: We thank Idit Keidar for her helpful comments on the algorithm and this paper.

REFERENCES

[1] G. V. Chockler and I. Keidar and R. Vitenberg, “Group Communication Specifications: A Comprehensive Study,” *ACM Computing Surveys*, vol. 33, no. 4, pp. 1–43, December 2001.

[2] L. Lamport, “On interprocess communication, Parts I and II,” *Distributed Computing*, vol. 1, no. 2, pp. 77–101, 1986.

[3] N. A. Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers, 1996.

[4] S. McCanne, *A Distributed Whiteboard for Network Conferencing*, UC Berkeley CS Dept., May 1992, Available from `ftp://ftp.ee.lbl.gov/conferencing/wb`.

[5] G. Shamir, “Shared Whiteboard: A Java Application in the Transis Environment,” Lab project, High Availability lab, The Hebrew University of Jerusalem, Jerusalem, Israel, October 1996, Available from: `http://www.cs.huji.ac.il/labs/transis/`.

[6] Maurice P. Herlihy and Jeannette M. Wing, “Linearizability: A correctness condition for concurrent objects,” *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, July 1990.

[7] Roger Khazan, *A One-Round Algorithm for Virtually Synchronous Group Communication in Wide Area Networks*, Ph.D. thesis, MIT Dept. of Electrical Eng. and Computer Science, May 2002.

[8] Idit Keidar and Roger Khazan, “A virtually synchronous group multicast algorithm for WANs: Formal approach,” *SIAM Journal on Computing*, vol. 32, no. 1, pp. 78–130, November 2002, Previous version in ICDCS 2000, pp. 344–355.

[9] Lamport, L., “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 78.

[10] Schneider, F. B., “Implementing fault tolerant services using the state machine approach: A tutorial,” *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, December 1990.

[11] Y. Amir and G. V. Chockler and D. Dolev and R. Vitenberg, “Efficient state transfer in partitionable environments,” in *2nd European Research Seminar on Advances in Distributed Systems (ERSADS’97)*, pp., 183–192, Full version: TR CS98-12, Institute of Computer Science, The Hebrew University, Jerusalem, Israel.

[12] Xavier Défago and André Schiper and Péter Urbán, “Totally ordered broadcast and multicast algorithms: A comprehensive survey,” Tech. Rep.DSC/2000/036, Swiss Federal Institute of Technology, Lausanne, Switzerland, September 2000.

[13] B. Lampson, “How to build a highly available system using consensus,” in *Distributed Algorithms*, LNCS 1151, 1996.