A Fault-Tolerant Sequentially Consistent DSM with a Compositional Correctness Proof

Niklas Ekström(✉) and Seif Haridi

KTH Royal Institute of Technology, Stockholm, Sweden
{neks,haridi}@kth.se

Abstract. We present the SC-ABD algorithm that implements sequentially consistent distributed shared memory (DSM). The algorithm tolerates that less than half of the processes are faulty (crash-stop). Compared to the multi-writer ABD algorithm, SC-ABD requires one instead of two round-trips of communication to perform a write operation, and an equal number of round-trips (two) to perform a read operation. Although sequential consistency is not a compositional consistency condition, the provided correctness proof is compositional.

1 Introduction

Using fault-tolerant distributed shared memory (DSM) as a building block in the design of a distributed system can simplify the design, as individual process failures are masked through replication. To characterize an implementation of distributed shared memory, we consider the following criteria:

- Consistency: a stronger consistency condition may be easier to program against, but may provide worse performance, and vice versa.
- Multiple writers: an implementation may allow a single process, or multiple processes, to update registers.
- Latency: the number of round-trips of communication required to execute an operation.
- Resilience: the number of processes that can be tolerated to be faulty in an execution, f, in relation to the total number of processes in the system, n.

In this paper, we consider the problem of implementing distributed shared memory that is sequentially consistent, allow multiple writers, can complete a write operation after one round of communication and a read operation after two rounds of communication, and that tolerates $f < n/2$ faulty processes. We present the SC-ABD algorithm as a solution to this problem. In Table 1 in the conclusion section, we present a comparison of SC-ABD to two other DSM algorithms along the mentioned criteria.

This work was supported by the Swedish Foundation for Strategic Research (SSF).

P.A. Abdulla and C. Delporte-Gallet (Eds.): NETYS 2016, LNCS 9944, pp. 183–192, 2016.
DOI: 10.1007/978-3-319-46140-3_14

Proving that a distributed shared memory implementation satisfies sequential consistency can be a difficult task. Unlike some other consistency conditions, sequential consistency is not a *compositional* consistency condition. Never the less, the proof given for the correctness of SC-ABD is compositional, and we therefore present this proof technique as a contribution in itself.

2 Model and Definitions

We consider an asynchronous distributed system composed of n processes, denoted $p_1, \ldots, p_n$, and a communication network with reliable links. We denote by $\Pi = \{1, \ldots, n\}$ the set of process identifiers. In any given system execution, a process is said to be correct if the process never crashes, and otherwise it is said to be faulty. A process that crashes stops taking steps and can never recover. We assume that at most f processes are faulty in any given execution, where $f < n/2$.

2.1 Shared Memory

A distributed shared memory is a distributed implementation of shared memory. We consider a shared memory consisting of read/write registers. Each register holds an integer value, initially zero. The shared memory defines a set of primitive operations, that provide the only means to manipulate the registers. In our case, the operations provided are *read* and *write*. A process invokes an operation and receives a response when the execution of the operation is complete. We will refer to an *operation execution* as an operation, if the distinction is clear from the context. Each process is allowed to have at most one outstanding operation, meaning that a process may not invoke another operation before the process has received the response for the previously invoked operation. Let o refer to a particular operation execution, invoked by process p_i. We denote by $inv(o)$ the *invocation event* that occurs when p_i invokes o, and denote by $res(o)$ the *response event* that occurs when the execution of o completes.

We model an execution using a *history*, which is a sequence of invocation and response events, ordered by the real times when the events occurred. History H is *sequential* if the first event is an invocation event, and every invocation event (except possibly the last) is immediately followed by the matching response event. By $H|p_i$ we denote the subsequence of H where every event occurs in process p_i; we refer to $H|p_i$ as a *process subhistory.* Similarly, by $H|x$ we denote the subsequence of H containing only events related to operations that target register x, and refer to $H|x$ as a *register subhistory.* A history is *well-formed* if each process subhistory is a sequential history, and in the following we only consider well-formed histories. Two histories H and H' are *equivalent*, denoted $H \simeq H'$, if and only if, for each process p_i, $H|p_i = H'|p_i$. For events e_1 and e_2 in history H we write $e_1 <_H e_2$ to denote that e_1 precedes e_2 in H. We say that "operation o is in history H" if $inv(o)$ is in H. For operations o_1 and o_2 in H we write $o_1 <_H o_2$ to denote that $res(o_1) <_H inv(o_2)$.

Operation o is *pending* in history H if the invocation event for o is in H but not the response event. History H is complete if H does not contain any pending operations. For presentational simplicity, we consider only complete histories in the rest of this paper.

The shared memory has a *sequential specification*, which is a set containing all sequential histories such that each read operation of some register returns the value written by the last write to that register (the write closest preceding the read in the sequential history), or the default value if no such write exists. A sequential history is *legal* if it is in the shared memory's sequential specification.

Sequential consistency is a consistency condition that was described by Lamport [7]. We define what it means for a history to be sequentially consistent:

Definition 1. *History H is* sequentially consistent*, denoted* $\mathsf{SC}(H)$*, if and only if there exists a legal sequential history S such that $S \simeq H$.*

The correctness conditions that we require of an algorithm implementing sequentially consistent distributed shared memory are:

- **Termination:** If a correct process invokes an operation, then the operation eventually completes.
- **Sequential Consistency:** Each history corresponding to an execution of the algorithm must be sequentially consistent.

2.2 Causality and Logical Clocks

Causality and logical clocks were described in a paper by Lamport [6]. Event e_1 is said to *causally precede* event e_2, denoted $e_1 \rightarrow e_2$, if at least one of the following conditions hold: (1) e_1 and e_2 both occur in the same process and e_1 occurs before e_2, (2) e_1 is the sending of message m and e_2 is the receipt of m, (3) there exists an event e' such that $e_1 \rightarrow e'$ and $e' \rightarrow e_2$.

A logical clock is a device that assigns integers to events in a manner consistent with the causally precedes relation. More precisely, by letting $lt(e)$ denote the logical time assigned to event e, we require that: $e_1 \rightarrow e_2 \Rightarrow lt(e_1) < lt(e_2)$.

3 Algorithm

In this section we present the SC-ABD algorithm, whose pseudo-code is contained in Algorithm 1. The algorithm is given as a set of reactive *handlers*. Each handler has an associated condition that describes when that handler is eligible for execution, e.g., when an operation is invoked, or a message is received.

For each process, the algorithm contains a variable lt that implements a logical clock. Whenever a handler is executed in response to a local condition (i.e., an operation is invoked) the logical clock is incremented by one. When a message is sent from process p_i to process p_j, the current logical time of p_i is included in the message, and when the message is received by p_j and the corresponding handler is executed, p_j's logical clock is updated to a logical time that is one greater than the maximum of p_j's previous logical time and the logical time included in the message.

Each process stores the values that have been written to the registers. In order to determine which value is more recent, a timestamp is associated with each value. A value and its associated timestamp are stored together as a *timestamp-value pair*. The algorithm has a local variable, *tvps*, that maps register identifiers to timestamp-value pairs.

Communication in the algorithm proceeds in *phases*. A phase consists of a round of communication, where the process executing the phase, p_i, sends a request to all processes and waits for responses from a majority of the processes before the phase ends.

A write operation has one phase: the update phase. The process executing the write operation, p_i, creates a timestamp as the pair with p_i's current logical time and p_i's process identifier, i. It then pairs this timestamp together with the value to be written into a timestamp-value pair. p_i sends an update request containing the register identifier and the timestamp-value pair to all processes (lines 16–20 in Algorithm 1). When process p_j receives the update request it updates its *tvps* with the supplied timestamp-value pair if the timestamp is greater than the timestamp of the timestamp-value pair that was previously stored, and then sends an ack response (lines 21–23). After p_i receives acks from a majority of processes, p_i returns OK (lines 24–30).

A read operation has two phases: the query phase and the update phase. The process executing the read operation, p_i, sends a query request to all processes containing the register identifier for the register that is being read (lines 1–5). When process p_j receives the query request, p_j retrieves the timestamp-value pair stored in *tvps* for the register identifier, and sends this timestamp-value pair in a response message to p_i. This timestamp-value pair is the maximal timestamp-value pair that p_j has received so far in an update request, or the initial timestamp-value pair, $((0, 0), 0)$, if no update request had been received previously (lines 6–7). When p_i has received response messages from a majority of processes, p_i chooses the timestamp-value pair, (ts, v), with the maximum timestamp out of the timestamp-value pairs received. Before returning value v, p_i performs an update phase using the (ts, v) timestamp-value pair, in order to guarantee that a majority of the processes have stored the timestamp-value pair before the read completes (lines 8–15 and 21–30).

4 Correctness Proof

We first prove that SC-ABD satisfies the termination property.

Lemma 1. *Algorithm SC-ABD satisfies the termination property.*

Proof. As links are reliable and a majority of processes are correct according to the assumptions in our model, each communication phase executed by a correct process is guaranteed to eventually complete, and every operation executed by a correct process is therefore guaranteed to complete. □

In the rest of this section we prove that the algorithm satisfies sequential consistency.

4.1 Linearizability

Linearizability is a consistency condition described by Herlihy and Wing [5].

Algorithm 1. SC-ABD – code for p_i.

```
Local variables:
$lt$ – logical time; initially 0
$rid$ – current request identifier; initially 0
$tvps$ – map from register ids to timestamp-value pairs; initially maps to $((0,0),0)$
$responses$ – tracking responses/acks; initially $\{\}$
$reading$ – indicating whether currently reading ($true$) or writing ($false$)
$rreg, rval$ – temporary storage for register identifier and return value during reads

Note: bcast $\langle m \rangle$ is an abbreviation for: for $j \in \Pi$ do send $\langle m \rangle$ to $p_j$

When READ($r$) is invoked:
$lt \leftarrow lt + 1$
$reading \leftarrow true$
$rreg \leftarrow r$
$rid \leftarrow rid + 1$
bcast $\langle$"query"$, lt, rid, r\rangle$

When $\langle$"query"$, lt', rid', r\rangle$ is
    received from $p_j$:
$lt \leftarrow \max(lt, lt') + 1$
send $\langle$"response"$, lt, rid', tvps[r]\rangle$ to $p_j$

When $\langle$"response"$, lt', rid', tsv'\rangle$ is
    received from $p_j$ with $rid = rid'$:
$lt \leftarrow \max(lt, lt') + 1$
$responses \leftarrow responses \cup \{(tsv', j)\}$
if $|responses| = \lfloor |\Pi|/2 \rfloor + 1$ then
    $(tsv, \_) \leftarrow \max(responses)$
    $(ts, rval) \leftarrow tsv$
    $responses \leftarrow \{\}$
    $rid \leftarrow rid + 1$
    bcast $\langle$"update"$, lt, rid, rreg, tsv\rangle$

When WRITE($r, v$) is invoked:
$lt \leftarrow lt + 1$
$reading \leftarrow false$
$tsv \leftarrow ((lt, i), v)$
$rid \leftarrow rid + 1$
bcast $\langle$"update"$, lt, rid, r, tsv\rangle$

When $\langle$"update"$, lt', rid', r, tsv'\rangle$ is
    received from $p_j$:
$lt \leftarrow \max(lt, lt') + 1$
$tvps[r] \leftarrow \max(tvps[r], tsv')$
send $\langle$"ack"$, lt, rid'\rangle$ to $p_j$

When $\langle$"ack"$, lt', rid'\rangle$ is
    received from $p_j$ with $rid = rid'$:
$lt \leftarrow \max(lt, lt') + 1$
$responses \leftarrow responses \cup \{j\}$
if $|responses| = \lfloor |\Pi|/2 \rfloor + 1$ then
    $responses \leftarrow \{\}$
    $rid \leftarrow rid + 1$
    if $reading$ then RETURN $rval$
    else RETURN OK
```

Definition 2. *History H is* linearizable*, denoted $\mathsf{LIN}(H)$, iff there exists a legal sequential history S such that $S \simeq H$, and $\forall o_1, o_2 \in H : o_1 <_H o_2 \Rightarrow o_1 <_S o_2$.*

Linearizability is *compositional*, in the sense that history H is linearizable if and only if each register subhistory $H|x$ is linearizable:

$$\mathsf{LIN}(H) \Leftrightarrow \forall x : \mathsf{LIN}(H|x) \tag{1}$$

From the definition of sequential consistency and the definition of linearizability, it follows that linearizability is stronger than sequential consistency:

$$\mathsf{LIN}(H) \Rightarrow \mathsf{SC}(H) \tag{2}$$

4.2 Logical-Time History

We define the *logical-time history* corresponding to history H, denoted H^{lt}, to be the sequence containing the same events as H, but reordered according to the logical times when the events occurred, using the process identifiers of the processes where the events occurred to break ties.

For each process p_i, the relative ordering of events in $H|p_i$ is preserved in $H^{lt}|p_i$, as the logical times of events in $H|p_i$ are monotonically increasing. It follows that the (real-time) history H and its corresponding logical-time history H^{lt} are equivalent, $H \simeq H^{lt}$. Together with the definition of sequentially consistent histories it follows that:

$$\mathsf{SC}(H) \Leftrightarrow \mathsf{SC}(H^{lt}) \tag{3}$$

4.3 Compositional Reasoning

Combining (1), (2), and (3), we have:

$$\big(\forall x : \mathsf{LIN}(H^{lt}|x)\big) \Rightarrow \mathsf{LIN}(H^{lt}) \Rightarrow \mathsf{SC}(H^{lt}) \Rightarrow \mathsf{SC}(H) \tag{4}$$

Equation (4) allows us to reason compositionally, i.e., to reason about, for each register x, the register subhistory $H^{lt}|x$ in isolation.

4.4 Reasoning About the Algorithm

We state a couple of definitions regarding the algorithm:

- The logical time of a handler execution is the value assigned to the lt variable on the handler's first line in the algorithm text.
- The timestamp of operation o, denoted $ts(o)$, is the timestamp used in the operation's update phase.

From the definition of logical-time history H^{lt}, it follows that:

$$o_1 <_{H^{lt}} o_2 \Rightarrow lt(res(o_1)) \leq lt(inv(o_2)) \tag{5}$$

We state and prove the following proposition:

Proposition 1. *Let o_1 and o_2 be operations in $H^{lt}|x$ such that o_1 contains an update phase and o_2 contains a query phase. If $o_1 <_{H^{lt}|x} o_2$ then $ts(o_1) \leq ts(o_2)$.*

Proof. Let p_i be the process that executes the update phase in o_1, and p_j be the process that executes the query phase in o_2. At the time when p_i's update phase completes, p_i will have received response messages from a majority of processes. Let M_u refer to this majority set of processes. Similarly, let M_q refer to the majority set of processes from which p_j received responses before the query phase in operation o_2 completed. As any two majority sets intersect, there must be one process, p_k, that is both in M_u and in M_q.

Let e_1 be the event when p_k processes o_1's update request, and e_2 the event when p_k processes o_2's query request. By causality we have $lt(e_1) < lt(res(o_1))$ and $lt(inv(o_2)) < lt(e_2)$, and together with (5) we get $lt(e_1) < lt(e_2)$. Since e_1 and e_2 are in the same process, this implies that e_1 occurs before e_2.

Since p_k returns the timestamp-value pair with the maximal timestamp that it has received in all previous update requests, the timestamp in the response to o_2's query request is guaranteed to be greater than or equal to the timestamp in o_1's update request. As p_j picks the timestamp-value pair with the maximal timestamp on line 11 of the algorithm, and uses it in its update phase, it follows that $ts(o_1) \leq ts(o_2)$. □

Lemma 2. *Algorithm SC-ABD satisfies the sequential consistency property.*

Proof. By using Eq. (4), we prove that the algorithm satisfies sequential consistency, by showing, for each execution, and for each register x, that $\mathsf{LIN}(H^{lt}|x)$ holds. From the definition of linearizability, we see that in order to prove that $\mathsf{LIN}(H^{lt}|x)$ holds we are required to show that there exists a legal sequential history S such that $S \simeq H^{lt}|x$, and, for all operations o_1 and o_2 in $H^{lt}|x$, if o_1 precedes o_2 in $H^{lt}|x$ then o_1 also precedes o_2 in S. We proceed by creating a total order on the operations in $H^{lt}|x$ as follows:

1. Order write operations according to their timestamps. Any two write operations have unique timestamps by construction, so this is a total order.
2. Then order each read operation immediately after the write operation that wrote the value that the read operation returned. If there are more than one read operations with the same timestamp then they are internally ordered based on the logical times when they were invoked (breaking ties using process identifiers).

Let S be the sequential history obtained from this total order. As each read operation in S returns the value written by the closest preceding write operation, it follows that S is legal.

We show that $o_1 <_{H^{lt}|x} o_2 \Rightarrow o_1 <_S o_2$ using the following case analysis:

– o_1 is a write, o_2 is a write: By causality we have $lt(inv(o_1)) < lt(res(o_1))$, which together with (5) gives us $lt(inv(o_1)) < lt(inv(o_2))$. Because of how the algorithm constructs timestamps (line 18), this implies that $ts(o_1) < ts(o_2)$, from which $o_1 <_S o_2$ follows.

- o_1 is a read, o_2 is a write: There exists a write w_0 such that $ts(w_0) = ts(o_1)$. Since the invocation event of w_0 causally precedes the response event of o_1, we have $lt(inv(w_0)) < lt(res(o_1))$, and, using (5), we have $lt(inv(w_0)) < lt(inv(o_2))$. From the analysis of the previous case we have $ts(o_1) = ts(w_0) < ts(o_2)$, from which $o_1 <_S o_2$ follows.
- o_1 is a write, o_2 is a read: By the assumption and Proposition 1 it follows that $ts(o_1) \leq ts(o_2)$, from which $o_1 <_S o_2$ immediately follows.
- o_1 is a read, o_2 is a read: Again, by the assumption and Proposition 1 it follows that $ts(o_1) \leq ts(o_2)$. If $ts(o_1) < ts(o_2)$ we directly have $o_1 <_S o_2$. Otherwise, we have $ts(o_1) = ts(o_2)$. By causality and (5) we have $lt(inv(o_1)) < lt(inv(o_2))$, and $o_1 <_S o_2$ follows from the definition of S.

Finally we must show that $S \simeq H^{lt}|x$. For any process p_i, consider the history $(H^{lt}|x)|p_i$, which is sequential. For any pair of operations o_1 and o_2 in $(H^{lt}|x)|p_i$, either $o_1 <_{(H^{lt}|x)|p_i} o_2$ or $o_2 <_{(H^{lt}|x)|p_i} o_1$. The same ordering will be preserved in $S|p_i$, according to the case analysis above. As S and $H^{lt}|x$ contain the same events, we have $S \simeq H^{lt}|x$. □

Theorem 1. *Algorithm SC-ABD is a correct implementation of sequentially consistent distributed shared memory.*

Proof. Follows directly from Lemmas 1 and 2. □

5 Related Work

Research about shared memory has a long history in distributed computing.

5.1 Consistency Conditions

Lamport described sequential consistency [6]. In multiprocessor systems, sequential consistency is widely regarded as the "gold standard", but most multiprocessor systems provide weaker consistency by default, and require that programs use memory fences to achieve sequentially consistent behavior.

Proving that a shared memory implementation satisfies sequential consistency is a well-researched problem. Alur, McMillan, and Peled proved that, in general, the sequential consistency verification problem is undecidable [1].

Bingham, Condon, and Hu suggested that the original formulation of sequential consistency, which is not prefix-closed, may be a reason why the verification problem is hard, and suggested two alternative variants to sequential consistency, Decisive Sequential Consistency (DSC) and Past-Time Sequential Consistency (PTSC) that are prefix-closed [4].

Plakal, Sorin, Condon, and Hill use logical (Lamport) clocks as a tool to reason about correctness of their distributed shared memory protocol [9].

Linearizability was described by Herlihy and Wing [5]. Linearizability has the pleasant property that it is a compositional consistency condition.

The cost of sequential consistency vs. linearizability was analyzed by Attiya and Welch [3]. They proved that the cost of sequential consistency is lower than the cost of linearizability under reasonable assumptions.

5.2 Fault-Tolerant Shared Memory

The ABD algorithm was described by Attiya et al. [2]. ABD was the first algorithm that showed it to be possible to implement fault-tolerant linearizable shared memory in a message passing system, but allowed only a single process to write to the memory. Write operations complete after a single round of communication and read operations complete after two rounds.

The multi-writer ABD (MW-ABD) algorithm was described by Lynch and Shvartsman [8]. MW-ABD extended the ABD algorithm by allowing multiple processes to write to the memory, and in order to do so added a second round of communication to write operations.

6 Conclusion

We presented the SC-ABD algorithm that implements fault-tolerant, sequentially consistent, distributed shared memory, and proved it to be correct using a compositional proof structure.

Table 1 contains a comparison between SC-ABD, ABD, and MW-ABD along the criteria mentioned in the introduction: consistency condition (linearizability (LIN) or sequential consistency (SC)); multiple writers allowed; number of rounds of communication required to complete a write (W)/read (R) operation; and how many faulty processes, f, that the algorithm tolerates.

Table 1. Comparison between three fault-tolerant DSM algorithms.

	ABD	MW-ABD	SC-ABD
Consistency	LIN	LIN	SC
Multiple writers	No	Yes	Yes
Latency	W:1, R:2	W:2, R:2	W:1, R:2
Resilience	$f < n/2$	$f < n/2$	$f < n/2$

In a situation where an application, running on top of distributed shared memory, would satisfy its correctness conditions if the distributed shared memory provides sequential consistency, and the application would benefit from having a lower latency for write operations, we think that SC-ABD is a good choice.

Finally, we showed that, although sequential consistency is not a compositional consistency condition, it was still possible to reason compositionally about the correctness of the algorithm.

Acknowledgements. We would like to thank the Swedish Foundation for Strategic Research for funding this work, and Jingna Zeng for helpful discussions.

References

1. Alur, R., McMillan, K., Peled, D.: Model-checking of correctness conditions for concurrent objects. In: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science, LICS 1996, p. 219. IEEE Computer Society, Washington, DC (1996)
2. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. J. ACM **42**(1), 124–142 (1995)
3. Attiya, H., Welch, J.L.: Sequential consistency versus linearizability. ACM Trans. Comput. Syst. **12**(2), 91–122 (1994)
4. Bingham, J.D., Condon, A., Hu, A.J.: Toward a decidable notion of sequential consistency. In: Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 2003, pp. 304–313. ACM, New York (2003)
5. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
6. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
7. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **28**(9), 690–691 (1979)
8. Lynch, N.A., Shvartsman, A.A.: Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In: Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS 1997), p. 272. IEEE Computer Society, Washington, DC (1997)
9. Plakal, M., Sorin, D.J., Condon, A.E., Hill, M.D.: Lamport clocks: verifying a directory cache-coherence protocol. In: Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 1998, pp. 67–76. ACM, New York (1998)