1 Introduction

In the last lecture, we discussed the ABD algorithm to implement atomic storage. That algorithm is from the early 1990s, and did not use quorums. Today, we’ll present a generalization of the ABD algorithm, using quorums.

2 Summary of generalized ABD algorithm

This isn’t a formal exposition of the algorithm; see the previous lecture for that. Instead, here’s a summary, so we can discuss and then prove correctness of implementation through quorums.

Algorithm 1 Client code summary for ABD algorithm

1: $\text{inv}(\text{ABD, write}(v))$
2: Phase 1: time synchronization phase
3: receive timestamp from $Q \in RQ$
4: Phase 2: write phase
5: send($\max((timestamp) + 1, v)$) to $Q \in WQ$
6: receive ack
7: $\text{inv}(\text{ABD, read()})$
8: Phase 1: read phase
9: receive values, timestamps from $Q \in RQ$
10: Phase 2: writeback phase
11: Send write, maxval, maxts
12: Receive ack from $Q \in WQ$
13: return maxval

Algorithm 1 is a sketch of the ABD client code. The fault tolerance idea is to send a timestamp request to all processors, but only to wait until one hears from all members of some Read Quorum. Similarly, one sends writes to all processors, but considers the task completed after receiving acks from any Write Quorum. That way, as long as there is at least one full Read Quorum and Write Quorum of correct processors, all other processors may fail. It can be formally proved that any implementation of atomic storage requires both a read phase and a write phase as part of its atomic read and write invocations. So there’s no way to skip one of the phases in the server code.
Algorithm 2 Code summary for server $s_i$ of generalized ABD algorithm

**Initialization:** $\text{tag}_i[ts, cid] = (0, 0)$; $val \leftarrow \bot$

- **Read request**
  - receive($\text{read}, cnt$) from client $c$
  - send($\text{readack}, val_i, \text{tag}_i, cnt$) to $c$

- **Write request**
  - receive($\text{write}, val', \langle ts', cid' \rangle, cnt$) from client $c$
  - if $ts' > \text{tag}_i \cdot ts$ OR ($ts' = \text{tag}_i \cdot ts$ AND $cid' > \text{tag}_i \cdot cid$) then
    - $\text{tag}_i \leftarrow (ts', cid')$
    - $val_i \leftarrow val'$
  - send($\text{writeack}, cnt$) to $c$

2.1 Proof of correctness

The ABD algorithm is wait-free as long as there is one correct Read Quorum and one correct Write Quorum. This is a generalization of majority-correct fault tolerance. It’s pretty easy to see that wait-freedom holds. Showing that atomicity holds is harder.

Recall the definition of atomicity. $O_H$ is the set of operations in history $H$.

**Definition 1** A partial execution satisfies atomicity if for history $H'$ of that execution there exists a history $H$ that completes $H'$ and a sequence $\sigma$ of operations $O_H$ such that

1. **(A1)** If $op_1 \prec_H op_2$ then $op_1 \prec_\sigma op_2$.
2. **(A2)** The value returned by each read is the value of the last preceding write in $\prec_\sigma$.

Essentially, we obtain atomicity because timestamp synchronization ensures multiple writers are synchronized. A writer reads the highest timestamp in writes preceding the current write. Similarly, the writeback phase for readers ensures atomicity among pairs of readers. In particular, if a reader that reads a value from a single server who has failed and has not let any other processors know about its updated timestamp, that reader will not commit to the new timestamp until hearing from an entire quorum. That ensures global correctness of the timestamp. We formalize this intuition in the following lemma.

**Lemma 1** The ABD algorithm satisfies atomicity.

**Proof:** Given history $H_{ex}'$ of execution $ex$, we define $H_{ex}$ that completes $H_{ex}'$ as follows.

1. **(a)** For any incomplete write wr in $H_{ex}'$ in which the write completes the Timestamp Synchronization Phase, we append the response step for wr to the end of $H_{ex}'$.
2. **(b)** For any other incomplete operation, we remove the invocation.
We prove atomicity by establishing the partial order \( \prec \) among all complete operations in \( H_{ex} \). Given any execution, we assign to each operation \( op \) a tag \( \text{tag}_{op} = (ts_{op}, cid_{op}) \). This mirrors the tags used in the actual algorithm.

If \( op \) is a WRITE, then \( ts_{op} \) equals the timestamp that it uses, which is \( maxts + 1 \), where \( maxts \) is computed by the writer in the Timestamp Synchronization Phase. Also, \( cid_{op} \) is the ID of the writer.

If \( op \) is a READ, then \( \langle ts_{op}, cid_{op} \rangle \) equals \( \langle maxts, maxcid \rangle \) computed by the reader in its read phase. This is the \( ts \) and \( cid \) of the writer whose value the reader chooses to return.

Let \( wr_0 \) be the WRITE in \( ex \) with the earliest realtime start of invocation. We set \( \text{tag}_{wr_0} = \langle 0, 0 \rangle \). We order operation tags lexicographically, i.e., \( \text{tag}_{op_1} < \text{tag}_{op_2} \) iff \( ts_{op_1} < ts_{op_2} \) OR \( ts_{op_1} = ts_{op_2} \) and \( cid_{op_1} < cid_{op_2} \). We now define the sequence \( \sigma \) and its partial ordering \( \prec \).

We say \( op_1 \prec \sigma op_2 \) if either:

- \( op_1 \) is a WRITE and \( op_2 \) is a READ and \( \text{tag}_{op_1} < \text{tag}_{op_2} \).
- \( op_1 \) and \( op_2 \) are both WRITES or both READS; or \( op_1 \) is a READ and \( op_2 \) is a WRITE, and \( \text{tag}_{op_1} < \text{tag}_{op_2} \).

To prove condition (A1) of atomicity, we show that if \( op_1 \prec H_{ex} op_2 \) then \( op_2 \not\prec \sigma op_1 \).

Since \( op_1 \) precedes \( op_2 \) in \( H_{ex} \), we know that \( op_1 \) is complete, i.e., \( op_1 \) completes its write or writeback. Hence, it propagates its tag to a write quorum before \( op_2 \) is invoked. When \( op_2 \) is invoked, every server \( s_i \in Q_w \) has tag \( \langle ts_i, cid_i \rangle \geq \text{tag}_{op_1} \).

**Case 1: \( op_2 \) is a write.** Since \( op_1 \) precedes \( op_2 \), we agree \( \text{tag}_{op_1} < \text{tag}_{op_2} \). Hence \( op_2 \in H_{ex} \), \( op_2 \) completes Time Stamp Synchronization, and obtains tags from all servers in some Read Quorum \( Q_r \). Since \( Q_r \cap Q_w \neq \emptyset \), we have \( maxts \geq ts_{op_1} \). Since \( ts_{op_2} = maxts + 1 \), we have \( ts_{op_2} > ts_{op_1} \), implying \( \text{tag}_{op_1} < \text{tag}_{op_2} \).

**Case 2: \( op_2 \) is a read.** We can show this with the same general argument we used for Case 1. 

\[ \]