Wait Free Atomic Semantics and Writebacks – Preliminary Version

Amitanand S. Aiyer[1], Lorenzo Alvisi[1], and Rida A. Bazzi[2]

[1] Department of Computer Sciences,
The University of Texas at Austin
{anand,lorenzo}@cs.utexas.edu
[2] Computer Science and Engineering Department,
Arizona State University
bazzi@asu.edu

Revised 26^{th} Feb 2007

1 Abstract

In the presence of Byzantine faults no protocol can achieve wait-free atomic semantics without having the reader perform a write back. All existing protocols for wait-free atomic semantics write back the entire value to the servers. We show the first protocol that does not writeback the value, but still achieves wait-free atomic semantics by writing back the timestamp. Further, we also show that wait-free atomic semantics can be achieved even when the readers are only allowed to change just *1-bit of information* at the servers.

2 Introduction

It can be shown that write-backs from the readers are necessary to achieve wait-free atomic semantics in a (pure shared memory) scenario where the base objects cannot communicate with each other directly. This limitation holds true, even if we use cryptography, self-verifying data, and need to tolerate just one crash failure among a million nodes.

In this work we show that despite the need to perform a write back for atomic wait free semantics, the readers need not write back the value as such. Specifically, we show that writing back just *One bit of information* is sufficient to achieve atomic wait free semantics.

We present our algorithm in two stages.

- First, we show a protocol **WriteBackOnlyTS** that achieves atomic wait-free semantics by only writing back the timestamp (without having to write back the value).
- Later, we present a subroutein to simulate the former protocol over a base object, where *only one-bit of information* is allowed to be updated by the reader. This subroutein when used as a primitive in **WriteBackOnlyTS** achieves wait-free atomic semantics, writing-back *only one-bit of information.*

For simplicity, we present a single-writer multiple-reader version of the protocol.

3 Model/Assumptions

The system consists of a set of n replicas (servers), a writer and a set of readers. Readers and writers are collectively referred to as clients.

Clients execute protocols that specify how *read* and *write* operations are implemented. We assume that clients do not start a new operation before finishing a previous operation. We assume that up to f servers may be Byzantine faulty and may deviate arbitrarily from the specified protocol. The remaining $(n - f)$ servers are correct and follow the specified protocol. We require that the total number of servers n be at least $4f + 1$.

We assume authenticated FIFO point-to-point asynchronous channels between clients and servers. Servers do not communicate with other servers.

Clients can fail by crashing and follow their protocol before they crash. Up to f Byzantine servers may behave arbitrarily. The remaining $n - f$ are correct and do not crash.

4 WriteBackOnlyTS protocol

This section presents the **WriteBackOnlyTS** protocol, that provides atomic wait free semantics when the reader only writes back the timestamp but not the value.

4.1 Protocol Description

The servers maintain 4 registers, **RVal** , **R1** , **RValPrev** and **R2** . Registers **RVal** and **RValPrev** store the value that is written by the writer, registers **R1** and **R2** store the timestamps for the value.

Register **R1** stores the timestamp for the value stored in register **RVal** , and is incremented whenever the register **RVal** is updated. Register **RValPrev** stores the previously written value, corresponding to the timestamp (**R1** - 1).

The register **R2** stores the timestamp of the oldest value that any further read may return.

Insight: To achieve atomic semantics, we just need to ensure that whenever a writer completes the write operation for a value with timestamp t, or a reader returns a value with timestamp t, no further "read" from register **R2** *will return a timestamp older than t.*

The servers follow the "listeners" pattern of communication as in [1]. They maintain a list of active readers and push updates to any of the registers to all active readers.

```
write( val ) {
      ++ts;
      // phase 1
      send (WRITE1, v = val, ts1 = ts) to all;
      wait for n-f acks;
      // phase 2
      send (WRITE2, ts2 = ts) to all;
      wait for n-f acks;
}
```

Fig. 1. Writer's protocol

Write protocol The protocol for the (single) writer is shown in Figure 4.1.

To perform a write operation,

1. The writer increases the existing timestamp by 1 and sends a message to all the servers requesting them to update register **RVal** with the new value and register **R1** with the new timestamp.
2. The writer then waits for $(n - f)$ acknowledgements before it now sends a message to all the servers to update the value of register **R2** with the new timestamp.
3. When $(n - f)$ servers acknowledge the receipt of this message, the write completes.

Read protocol To perform a read operation, a reader sends the read request to all the servers. After gathering at least $n - f$ responses, the reader calculates *min_ts2* as the $2f + 1$-st smallest timestamp among all the received values for **R2** .

The reader then waits and collects responses, until it receives enough matching value-timestamp pairs for some timestamp that is at least as fresh as *min_ts2*.

If the value timestamp of the value that is being decided upon is same as *min_ts2*, $f + 1$ identical responses are sufficient. Otherwise, the reader waits to collect $n - f$ identical responses.

After collecting enough identical responses to choose a value, the reader writes back the timestamp of the value chosen to register **R2** . The read operation completes after the reader receives $(n - f)$ acknowledgements for the write-back.

Server's protocol The servers follow the "listeners" pattern of communication with the readers. They maintain a list of active readers – readers that are currently reading – and forward the values to the readers until they complete the read.

When a server recieves a write or a write-back message, it updates the corresponding register and acknowledges the client.

```
read( ) {
    send (START_READ) to all;
    do {
        receive (RESPONSE, v, ts1, vprev, ts2) from server s;
        value[s][ts1] := v,
        value[s][ts1-1] := vprev,
        if timestamp2[s] == NULL:
            timestamp2[s] := ts2;

        if received >= (n − f) responses:
            min_ts2 := (2f + 1)-th smallest among timestamp2[];
            prev_ts2 := (f + 1)-th largest among timestamp2[];

        if min_ts2 != null and ∃ ts, val, such that:
            (either
                ts ≥ min_ts2 and
                ts ≤ prev_ts2 and
                there are f + 1 servers, s, with
                value[s][ts] == val
            or
                ts ≥ min_ts2 and
                ts > prev_ts2 and
                there are n − f servers, s, with
                value[s][ts] == val)
            begin
                selected_ts = ts;
                selected_value = val;
                break;
            end
    } while(true)

    // write-back phase
    send (WRITE_BACK, ts2 = selected_ts) to all;
    wait for (n − f) acks;

    return selected_value;
}
```

Fig. 2. Reader's protocol

4.2 Protocol Correctness

First, we show the safety condition: that our protocol achieves atomicity assuming it is live. We will later show that our protocols always terminate establishing liveness.

Atomicity is shown by proving that

W-R atomicity Once a writer completes a write operation for a value with timestamp t, no further read will return an older value.

R-R atomicity Once a read operation returns a value with timestamp t, no future read will return an older value.

Lemma 1 (W-R atomicity). *Once a writer completes a write operation for a value with timestamp t, no further read will return an older value.*

Proof: To complete a write operation, the writer sends (WRITE2, ts2 = ts) to all and waits for acknowledgements from at least $n - f$ servers. Thus, when a write operation completes at least $n - 2f$ correct servers have set the value of

```
server( ) {
      while(true) {
          receive msg from client;

          if client == WRITER:
              if msg == (WRITE1, v = val, ts1 = ts) and ts > R1:
                  waitFor(R2 >= ts − 1);
                  RV_PREV = RV1;
                  R1 = ts;
                  RV = val;

                  // send updates to all active readers;
                  send (RESPONSE, RV, R1, RV_PREV, R2) to Readers;

              if msg == (WRITE2, ts2 = ts):
                  waitFor(R1 == ts);
                  if (ts > R2)
                      R2 = ts;
                      // send updates to all active readers;
                      send (RESPONSE, RV, R1, RV_PREV, R2) to Readers;

              acknowledge the client;

          if client == READER:
              if msg == (START_READ):
                  // add client to the set of active readers;
                  Readers = Readers ∪ { client }
                  send (RESPONSE, RV, R1, RV_PREV, R2) to client;

              if msg == (WRITE_BACK, ts2 = ts):
                  if ts > R2:
                      waitFor(R2 >= ts − 1);
                      R2 = ts;
                  // remove client from active readers
                  Readers = Readers \ { client }
                  acknowledge the client;
      }
}
```

Fig. 3. Server's protocol, involving writeback of timestamp

register $\mathbf{R2} = t$. Since the value of register **R2** only increases, any further read will only receive values $\geq t$ from these correct servers.

Hence, the value of *min_ts2* computed by the reader will be $\geq t$. So, the read will only return a value $\geq t$. □

Lemma 2 (R-R atomicity). *Once a read operation returns a value with timestamp t, no future read will return an older value. for a value with timestamp t, no further read will return an older value.*

Proof: Similar to Lemma 1 □

Lemma 3 (Correctness). *A read only returns a value that is written by a writer.*

Proof: A value is set as the selected_value only if there are either $f+1$ or $n-f$ servers responding with the same value and timestamp. Since, in either of the case, at least one of the servers has to be correct and clients are non-malicious, selected_value will only be set to the value written by the writer. □

Lemma 4 (Write Liveness). *A write operation always terminates and is wait-free.*

Proof: A write operation only waits for $n - f$ responses at any stage. Since at most f servers are faulty, it follows that a write operation always terminates and is wait-free. □

Lemma 5 (Read Liveness). *A read operation always terminates and is wait-free.*

Proof: If a reader will eventually receive responses from all the $n - f$ correct servers.

Let t be the $f+1$-th largest timestamp2 entry received from a correct server. Since $n \geq 4f + 1$, we would have

$$t \geq min\_ts2$$

$$t \leq prev\_ts2$$

The correct server must have set register **R2** to t, either because it received a (WRITE2, ts2 $= t$) message from the writer, or it received a (WRITE_BACK, ts2 $= t$) message from another reader [3]

We will now argue that in either case, the reader will receive enough identical responses to set selected_value and selected_ts. If the reader sets selected_value to a non-null value, then termination follows since the reader only waits for $n-f$ responses to the write-back message.

Consider the following two cases:

1. The writer sends at least one (WRITE2, ts2 $= t$) message:
   In this case, the writer would have sent (WRITE1, value = val, ts1 $= t$) to all the servers. Eventually all the correct servers among the $2f + 1$ servers with timestamp $\leq t$ will recieve this message and send the updates to the reader. When all these updates reach the reader, the reader will have $\geq f+1$ identical responses for the value with timestamp t, and will be select a value if the reader has not already selected.
2. If the writer never sent a (WRITE2, ts2 $= t$) message, then some reader(s) must have sent a (WRITE_BACK, ts2 $= t$) message. Consider the first reader to have sent such a message. No client would have written t to register **R2** earlier, and the *prev_ts2* evaluated during that read would have to be $< t$. Thus, in order to have selected t to perform a write-back, the reader must have received at least $n - f \geq 2f + 1$ identical responses with timestamp t. At least $f + 1$ of these are correct and there value will not be overwritten by any further write. Thus when the responses from these servers reach the reader, the reader will be able to decide on the value.

□

[3] The message could also have been sent by the same reader during a previous read operation. The argument in this case is similar to the argument when the read was by another reader.

5 Protocol with only One write-backable bit

We now show how to achieve atomic wait free semantics if only one bit of information at the server may be updated by a reader with a write-back. We will show how to infer the same values of registers **R1** and **R2** using just the register **R1** and a *write-backable* bit b.

As in the protocol presented earlier, register **R1** stores the timestamp of the value stored in register **RVal** and is updated by the writer during phase 1 of the write. The value of the register **R2** is calculated using the formula

$$R2 = R1 \pm b$$

where R1 is value in register **R1** , R2 is the value in register **R2** and b is the rewritable bit, that may be updated by both the reader and the writer.

On recieving WRITE1 message, the server updates the registers **RVal** , **R1** and sets the equation for evaluating **R2** as

$$R2 = R1 - b$$

while setting b to 0 or 1 accordingly to satisfy the equation. If this write message is not significantly delayed, then the value of b would be 1. However, if before the write message has reached the server a reader has written back the timestamp to **R2** then b would be 0.

On recieving WRITE2 message, the server sets the equation for evaluating **R2** as

$$R2 = R1 + b$$

while setting b to 0 or 1 accordingly to satisfy the equation. Typically, at this point, b would be 0 unless this message has been so much delayed that the writer has written the next value, a reader read it and has performed a write-back.

The protocol for the server using only 1-write backable bit is shown in Figure 5.

The protocols for the readers and the writer are unchanged.

Correctness [**Sketch**] Inspite of having only 1 writebackable-bit at the server this protocol provides the same interface and guarantees as the protocol in Figure 4.1.

Specifically, for any execution, a message sent by a (correct) client to a server running either of the protocols always receives identical responses.

Thus this protocol also achieves the same guarantees as the previous protocol, namely, wait-free atomic semantics.

References

1. Martin, J.P., Alvisi, L., Dahlin, M.: Minimal byzantine storage. In: DISC '02, London, UK, Springer-Verlag (2002) 311–325

```
server( ) {
    while(true) {
        receive msg from client;

        if client == WRITER:
            if msg == (WRITE1, v = val, ts1 = ts) and ts > R1:
                // waitFor(R2 >= ts − 1);
                waitFor(R1 == ts-1 and EQN == "R2 = R1 + b")
                int tmpR2 = computeR2();
                RV_PREV = RV1;
                R1 = ts;
                RV = val;
                EQN == "R2 = R1 - b";
                b = 1 - b; // same as b = R1 - R2
                    // = ts - tmpR2
                    // = ts - (ts-1 + b)
                    // = 1 - b
                assert(computeR2() == tmpR2);

                // send updates to all active readers;
                send (RESPONSE, RV, R1, RV_PREV, computeR2()) to Readers;

            if msg == (WRITE2, ts2 = ts):
                // waitFor(R1 == ts);
                waitFor(R1 == ts);
                assert(EQN == "R2 = R1 - b");

                EQN == "R2 = R1 + b";
                b = 0;
                assert(computeR2() == ts);

                if (tmpR2 != computeR2())
                    send (RESPONSE, RV, R1, RV_PREV, computeR2()) to Readers;

            acknowledge the client;

        if client == READER:
            if msg == (START_READ):
                // add client to the set of active readers;
                Readers = Readers ∪ { client }
                send (RESPONSE, RV, R1, RV_PREV, computeR2()) to client;

            if msg == (WRITE_BACK, ts2 = ts):
                if ts > computeR2():
                    // waitFor(R2 >= ts − 1);
                    waitFor((R1 == ts - 1 and EQN == "R2 = R1 + b")
                        or R1 > ts − 1);
                    // computeR2() = ts;
                    if (R1 == ts and EQN == "R2 = R1 - b"):
                        b = 0;
                    else if (R1 == ts - 1 and EQN == "R2 = R1 + b")
                        b = 1;
                    else if (R1 == ts and EQN == "R2 = R1 + b")
                        // do nothing
                        ;
                    else if (R1 >= ts + 1)
                        // do nothing
                        ;

                // remove client from active readers
                Readers = Readers \ { client }
                acknowledge the client;

        assertInvariants();
    }
}
```

Fig. 4. Server's protocol, involving writeback of only 1-bit