In this addendum, we seek to clarify the assumptions made by DAM, as well as prove several correctness properties of the system which are agnostic to the simulator(s) built on top of the DAM framework. Since the code *within* each context is entirely user-defined, this focuses on proving properties about communication and timing information between contexts.

0.1 Viewing Time

A context may *view* the time of another context, obtaining a handle. Using the view, the viewing context may perform the following operations:

1. view.tick_lower_bound() -> Time: The viewing context reads the viewed context's time, obtaining a lower bound on the time of the viewed context.
2. view. wait_until (Time) -> Time: The viewing context blocks until the viewed context has reached up to a particular time point T^S_{wait}, provided that the awaited point is not in the future relative to the viewing context; $T^S_{wait} \leq T^S_{viewing}$. This operation yields the updated timestamp $T^S_{viewed} \geq T^S_{wait}$.

Both operations form Release-Acquire pairs between the viewed context's timestamp update (Release) and the operation (Acquire).

0.2 Assumptions

Assumption 1 (Context-Monotonicity)**.** Simulated time for each context T^S is monotonically non-decreasing w.r.t. real time.

Assumption 2 (Channel-Monotonicity)**.** The timestamps written to each real and simulated channel is monotonically non-decreasing w.r.t. both real and simulated time.

Assumption 3 (Future-Horizon)**.** A context C writes to a simulated channel c with timestamp at least $T^S_C + l_c$, where $l_c \geq 1$ is a static lower bound on the channel latency.

Assumption 4 (Channel-Behavior)**.** Both real and simulated channels are single-producer, single-consumer FIFOs with non-zero channel depth.

Assumption 5 (Acq-Rel)**.** Context time updates occur with *Release* ordering, context time reads occur with *Acquire* ordering, and this ordering is *transitive* w.r.t. other operations.

Assumption 6 (Access-Type)**.** Real and simulated channel enqueue/dequeue operations are both reads and writes, and empty/full checks are read operations.

1 Channel Behaviors

In this section, we seek to prove that DAM's simulated channel operations return the correct timestamps as well as the correct values when applicable. For the purposes of this section, consider a pair of contexts A, B, where B enqueues to a channel c and A dequeues from the same channel.

1.1 Dequeue Behavior

Theorem 1 (Dequeue-Empty)**.** If A views B and obtains a lower bound with timestamp T^S_B, and then peeks c and c is empty, then A has dequeued all enqueues by B to c with timestamps prior to $T^S_B + l_c$.

Proof by contradiction. Suppose there exists some enqueued data d with timestamp $T^S_d < T^S_B + l_c$ that was not dequeued by A. By assumption 3, d must have been enqueued prior to B advancing to T^S_B, both of which are write operations by assumption 6. By transitivity (assumption 5), reads which see the updated timestamp T^S_B *must* also observe all prior writes, including the enqueue of d. This contradicts our original assumptions on d. □

Theorem 2 (Dequeue-Ordering)**.** If A peeks the channel c and observes data d with timestamp T^S_d, then A has dequeued all prior enqueued data d' with timestamps $T^S_{d'} < T^S_d$.

Proof by contradiction. Suppose there exists some enqueued data d' with timestamp $T^S_{d'} < T^S_d$, but has not yet been dequeued. By assumption 2, d' must have been enqueued to c before d. Then by assumption 4, since the channels have FIFO behavior, previously enqueued elements must be dequeued before later elements. Therefore d' must have already be dequeued before d becomes the front of the channel, resulting in a contradiction. □

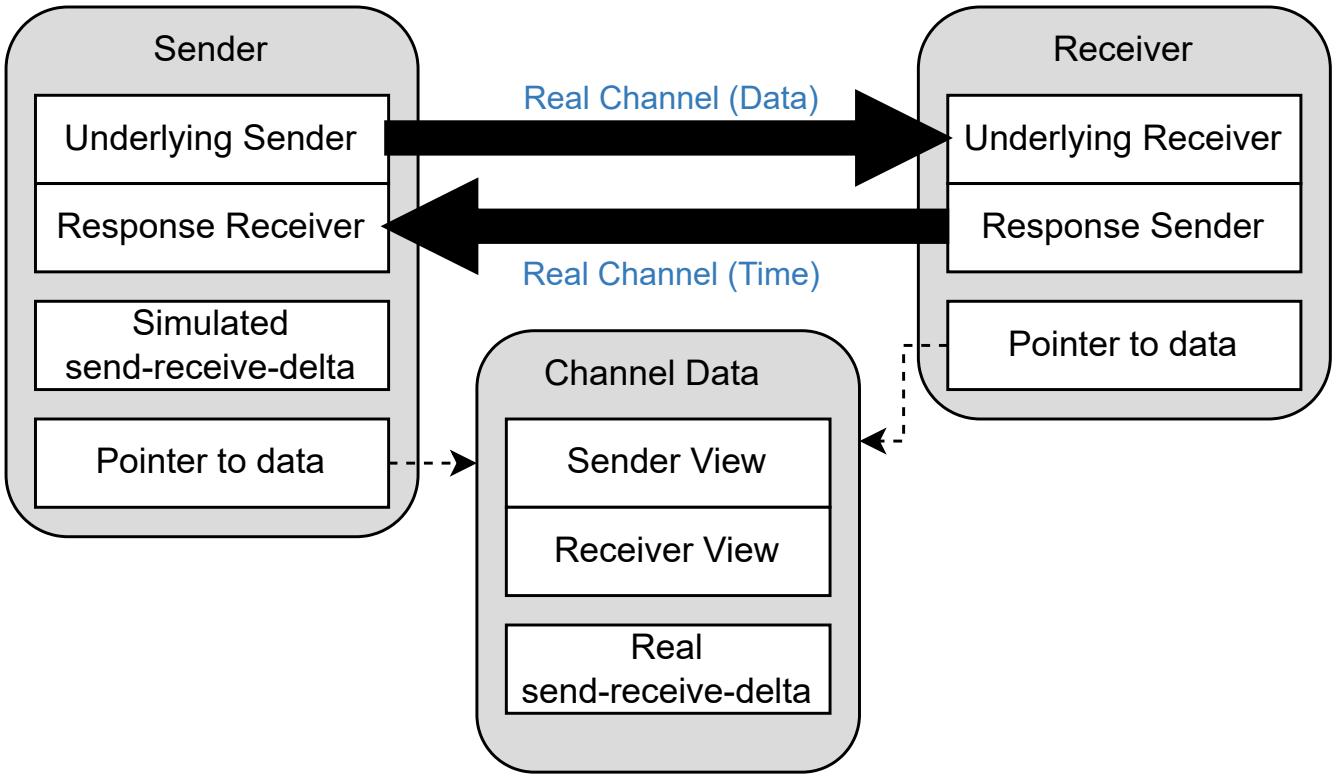

Figure 1: An illustration of the real sender/receiver data structures and their connections. Simulated channels are backed by a pair of real channels – one which carries data from the sender to the receiver, and another which notifies the sender of the simulated time at which elements were removed from the channel. (Replicated Fig 2. from the original paper)

1.2 Enqueue Behavior

DAM uses a local send-receive-delta on the sending side, which maintains a conservative approximation of the difference between the number of sent messages and the number of dequeued messages. This number is incremented each time a message is sent, and decremented when a response with a past or present timestamp is received. If a future timestamp is received, it is left in the channel for later processing. Below, we have included a simplified procedure for checking if a channel is full.

```
fn is_full() -> bool {
    if send_receive_delta < capacity {
        return false;
    }
    let reader_time = receiver_view.tick_lower_bound();
    match response.peek() {
        case Some(timestamp) if timestamp <= sender_time {
            response.recv(); // Pop the timestamp from the response channel
            send_receive_delta -= 1;
            return false;
        }
        case Some(timestamp) if timestamp > sender_time {
            // If the next timestamp is in the future, assumption 4's FIFO assumption
            // dictates that no intervening prior timestamp can arrive, and therefore
            // the channel must be full even if the check was speculative.
            return true;
        }
        case None => {}
    }
    if reader_time >= sender_time {
        // If the reader time was in the future or present
        // then the check was non-speculative and the channel is full.
        return true;
```

```
    }
    // Otherwise, the check was speculative, so a second check is needed
    // after waiting for the receiver to catch up.
    receiver_view.wait_until(sender_time);
    match response.peek() {
        case Some(timestamp) if timestamp <= sender_time {
            response.recv();  // Pop the timestamp from the response channel
            send_receive_delta -= 1;
            return false;
        }
        case Some(timestamp) if timestamp > sender_time {
            return true;
        }
        case None => {
            // Since this check is non-speculative, an empty channel indicates
            // that no dequeues occurred and the channel is full.
            return true;
        }
    }
}
```

Theorem 3 (Backpressure-Start)**.** If at time T_*^S, channel c is full, then B will observe that the channel is full at time $T_B^S = T_*^S$.

Proof. The send-receive-delta is incremented on each enqueue, and only decremented when receiving a corresponding response from the receiver which occurred at or prior to the current time. As a result, the local send-receive-delta is a conservative overestimate of the simulated channel occupancy at T_B^S.

□

Theorem 4 (Backpressure-End)**.** If at time T_*^S, channel c is not full, then B will observe that the channel is not full at time $T_B^S = T_*^S$.

Proof. If the SRD is equal to the channel capacity, then DAM speculatively checks the response channel for a dequeue time.

If no timestamp is present, then waits until the receiver to catch up to some time $T_A^S \geq T_B^S$. Depending on the relative timestamps, this either triggers SvA or SvP. Due to assumption 6, this second check is guaranteed to be non-speculative.

Due to assumption 4, if a timestamp is present, then it must be the lowest timestamp that has not yet been dequeued. If this timestamp is in the past, then DAM dequeues the timestamp and decrements the SRD. In this case, the DAM protocol correctly reports that the channel is not full. Alternatively, if the response channel has a timestamp in the future, then by theorem 2 it is guaranteed that no intervening timestamp is possible, and therefore the channel must be full.

If the response channel is empty during both checks, then by assumption 5 and assumption 6, the next response must have timestamp greater than T_A^S. As a result, B can correctly conclude that c is full at T_B^S. □