Running Consistent Applications Closer to Users with Radical for Lower Latency

1 Consistency Proof

1.1 Definitions

Here we show that Radical provides linearizability guarantees to its users. Linearizability is a local consistency model, thus to prove that a system is linearizable, it is sufficient to show that operations on each individual key are linearizable [1]. Without loss of generality, consider key k. Radical supports two operations: reads ($r(k_v)$ denoting that a read to key k returns its value version v) and writes ($w(k_v)$ denoting a write to k of value version v). Linearizability requires that there exists a total order of operations on k which we construct as follows:

1. Each write is ordered by its version number: $w(k_v) \xrightarrow{exe} w(k_{v+i}), i > 0$.
2. Each read is ordered after the write that it observes: $w(k_v) \xrightarrow{exe} r(k_v)$
3. All reads that observe the same write are ordered by invocation time: $r_1(k_v) \xrightarrow{exe} ... \xrightarrow{exe} r_n(k_v)$ if $r_1.inv < ... < r_n.inv$

For the system to be linearizable, this total order must also obey real-time ordering constraints.

1.2 Proof

Thus to show that Radical is linearizable, we must show that Radical respects the above-constructed total order under real-time constraints. First, let all operation be illustrated as a directed graph where the operations are nodes that are connected by real-time edges. Then we can say that there exists a total real-time order if and only if the directed graph is acyclic (operations do not circularly affect each other), meaning that the following invariant holds: $\forall op_i, op_j, (op_i \xrightarrow{rto} op_j) \implies \neg(op_j \xrightarrow{rto} op_i)$.

Note that there exists a real-time edge $op_1 \xrightarrow{rto} op_2$ if op_2 sees the result of op_1 and op_2 starts after op_1 ends ($op_1.resp < op_2.inv$). Note that this implies that operations are transitive. That is if $op_1 \xrightarrow{rto} op_2$ and $op_2 \xrightarrow{rto} op_3$, then $op_1 \xrightarrow{rto} op_3$. Two operations have a real-time edge in one of two cases:

1. op_1 and op_2 are performed sequentially by the same function execution where op_1 precedes op_2
2. op_1 and op_2 are performed by two different executions, and op_2 sees the result of op_1 via Radical's design

We prove that Radical's total order is a real-time order by contradiction. More specifically, we consider pairs of operations (w, w), (r, r), (r, w), (w, r).

1. (w, w)**:** let there be two writes such that $w_1 = w(k_v) \xrightarrow{exe} w_2 = w(k_{v'}), v' > v$. By the contradiction we assume we also have $w_2 \xrightarrow{rto} w_1$. Because w_2 is real-time ordered before another operation, we know w_2 must have completed. There are two cases:
   a. w_2 completed at the edge. Then it must have acquired a write lock as part of its successful consistency check (lines XX–YY). This lock precludes any other operation from acquiring a write lock (lines XX–YY) until it is released. There are three subcases depending on how w_1 executes:
      i. w_1 completes at the edge. Then it must also acquire a write lock as part of its successful consistency check. Since $w_2 \xrightarrow{rto} w_1$, then $w_2.write\_lock < w_2.resp < w_1.inv < w_1.write\_lock$, so we know that w_1 must acquire its lock after w_2 releases its lock. Version numbers only increase for subsequent writes (lines XX–YY), so v' (written by w_2) $< v$ (written by w_1). Contradiction.
      ii. w_1 completes in the datacenter. Then it must also acquire a write lock as part of its failed consistency check. w_2 must have released the locks before w_1 could start running at the datacenter, so if $w_2 \xrightarrow{rto} w_1$, then $w_2.write\_lock < w_2.resp < w_1.inv < w_1.write\_lock$, similarly to the case when w_1 completes in the edge. Version numbers only increase for subsequent writes, so $v' < v$. Contradiction.
      iii. w_2 completes at the datacenter or at the edge as part of a timeout. If w_2 started running at the edge, then it must have acquired a write lock as part of its consistency check. If the datacenter times out on the follow-up from the edge, the write lock is still held, and the execution is rerun in the datacenter. Thus, depending on whether the follow-up from the edge finally arrives or the function on the datacenter finishes executing first, the same logic from cases (i) and (ii) applies. If the function finished executing first, the later stale follow-up is discarded, and if the follow-up arrives during datacenter execution, the result of the datacenter execution is ignored.
   b. w_2 completes in the datacenter. Then it must have acquired a write lock as part of its failed consistency check (lines XX-YY). No other operations, whether in the datacenter or at the edge, could be performing operations on k until w_2 completes. There are two

subcases depending on where w_1 executes (we omit the timeout case as the reasoning is similar to what is described in 1(a)iii):

i. w_1 completes at the edge. Then it must have been the case that w_1 acquired the write lock as part of its successful consistency check. This necessitates that w_2 released its lock first. If $w_2 \xrightarrow{rto} w_1$, then $w_2.write\_lock < w_2.resp < w_1.inv < w_1.write\_lock$. Version numbers only increase for subsequent writes, so $v' < v$. Contradiction.

ii. w_1 completes at the datacenter. Then it must have waited for w_2 to release its lock before acquiring the lock as part of its failed consistency check. Thus, w_2 and w_1 execute at the datacenter sequentially, in that order. As in the above case, $w_2 \xrightarrow{rto} w_1 \implies w_2.write\_lock < w_2.resp < w_1.inv < w_1.write\_lock$, and since version numbers only increase for subsequent writes, so $v' < v$. Contradiction.

2. (r, r)**:** let there be two reads such that $r_1 \xrightarrow{exe} r_2$. By the contradiction we assume we also have $r_2 \xrightarrow{rto} r_1$. Because r_2 is real-time ordered before another operation, we know r_2 must have completed. There are two cases:

a. If $r_1 = r(k_v)$ and $r_2 = r(k_v)$ return the same values, then the two reads are ordered by invocation time. Since $r_1.inv < r_2.inv$, there cannot be a real-time edge from r_2 to r_1. Contradiction.

b. If $r_1 = r(k_v)$ and $r_2 = r(k_{v'})$ where $v' > v$ and $r_1 \xrightarrow{exe} r_2$, then there must exist w_1 that r_1 sees and w_2 that r_2 sees, such that w_2 is ordered after w_1. In other words, it must be the case that $w_1 \xrightarrow{exe} r_1$, $w_2 \xrightarrow{exe} r_2$, and $w_1 \xrightarrow{exe} w_2$. Assuming there exists the edge $r_2 \xrightarrow{rto} r_1$, then we know that r_2 must have completed. There are two cases:

i. If r_2 completed at the edge, it must be the case that the read lock on k was acquired and the consistency check was successful. Thus, there could have been no writes to k between $r_2.inv$ and $r_2.read\_lock$. There are two subcases depending on where r_1 executes:

A. r_1 also completes at the edge. Then it must have also acquired a read lock on k to ensure all pending writes were complete. However, r_1 would have acquired the read lock after w_1 executed. Since w_1 should have acquired a write lock before r_1, it is necessary that w_1 executed between $r_2.read\_unlock$ and $r_1.read\_lock$. In other words, the following must be true: $r_2.inv < r_2.read\_lock < w_1.write\_lock < r_1.read\_lock$. Since $r_2.resp < r_1.inv$ by assumption ($r_2 \xrightarrow{rto} r_1$) and $w_2.write\_lock < r_2.read\_lock$ ($w_2 \xrightarrow{exe} r_2$), then transitively, it must be true that $w_2 \xrightarrow{rto} r_1$. Since reads must be ordered after the writes they observe, w_1 must occur between w_2 and r_1. However, writes are ordered in increasing version number in the total order, and since $w_2 = w(k_{v'})$, $w_1 = w(k_v), v' > v$, this is a contradiction.

B. r_1 completes at the datacenter. Then it must have acquired a read lock on k to avoid reading stale data. Similarly, w_1 must have acquired the write lock at some point before r_1 acquired its read lock. Similar to the above, we necessarily expect that $r_2.inv < r_2.read\_lock < w_1.write\_lock < r_1.read\_lock$ and $w_2.write\_lock < r_2.read\_lock$. Thus $w_2 \xrightarrow{rto} r_1 \implies w_2 \xrightarrow{exe} w_1 \xrightarrow{exe} r_1$ which violates legal ordering of writes by version number. Contradiction.

ii. If r_2 completed at the datacenter, then the read lock on k was acquired as part of a failed consistency check. No executions that write to k are possible (whether at datacenter or on edge) once the read lock is acquired. As described above, regardless of where r_1 executes, w_1 must have obtained a write lock beforehand such that $w_2.write\_lock < r_2.read\_lock < w_1.write\_lock < r_1.read\_lock$. Consequently, $w_2 \xrightarrow{rto} r_1$, $w_2 \xrightarrow{exe} w_1 \xrightarrow{exe} r_1$ which is out of order writes. Contradiction.

3. (r, w)**:** let there be a read and a write such that $r = r(k_v) \xrightarrow{exe} w' = w(k_{v'}), v' > v$. By the contradiction we also have $w' \xrightarrow{rto} r$. Because w' is real-time ordered before another operation, we know w' must have completed. There are two cases:

a. w' completes at the edge. Then it must have acquired a write lock on k before r executed. Whether r was executed on edge or in datacenter, it held the read lock and sent a response back to the user before w' took its write lock. In other words, since $w' \xrightarrow{rto} r$, then $w'.write\_lock < w'.resp < r.inv < r.read\_lock$. However, r would return the value of k of version v, not v'. Thus there must exist a write $w = w(k_v)$ that is ordered between w' and r such that $w' \xrightarrow{exe} w \xrightarrow{r}$, which are out-of-order writes. Contradiction.

b. w' completed at the datacenter. Then it must have acquired a write lock as part of its failed consistency check. Then w, by the same logic as above, either executes in datacenter or on edge, after w' released its lock, so if $w' \xrightarrow{rto} r$, the operations must be ordered as $w' \xrightarrow{exe} w \xrightarrow{r}$, guaranteed by $w'.write\_lock < w.write\_lock < r.read\_lock$, which are out-of-order writes. Contradiction.

4. (w, r)**:** let there be a write and a read that observes that write such that $w = w(k_v) \xrightarrow{exe} r = r(k_v)$. By the contradiction we also have $r \xrightarrow{rto} w$. Because r is real-time ordered before another operation, we know r must have completed. There are two cases:
   a. r completes at the edge. Then it must have acquired a read lock on k. There are then two subcases on where w is executed:
      i. If w is executed on the edge, then it acquired the write lock and necessarily after r released its read lock. Thus r must have read some data before w executed. Since r read k_v before k_v was written by w, the read is not ordered by the write it observes. Contradiction.
      ii. If w executed in the datacenter, then similarly $r.inv < r.read\_lock < r.resp < w.inv < w.write\_lock$ which means that the read is not ordered after the write it observes. Contradiction.
   b. r completes at the datacenter as a result of a failed consistency check. It does so after acquiring a read lock such that all pending writes complete first. Consider where w is executed afterwards:
      i. If w is executed at the edge, then w acquired a write lock as part of a successful consistency check, thereby requiring that $r.read\_lock < w.write\_lock$. The write lock is only released upon central datastore update as a result of the follow up from edge. Thus if $r \xrightarrow{rto} w$, then the read would observe a write that did not yet occur. Contradiction.
      ii. If and when w is executed at the datacenter, then r must have already released its read lock. In other words, r and w are executed sequentially at the datacenter, so the read would not be ordered after the write it observes. Contradiction.

Thus, the ordering obeys the real-time order for all pairs of operations on each key k. Because our given ordering is a legal total order that obeys real-time constraints, Radical provides linearizability.

References

[1] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.