Proofs of Linearizability and Regular Semantics for SimpleStores

1 Appendix

1.1 Linearizability of Chain SimpleStore

Chain SimpleStore (Figure **??**) provides linearizability. To show this, we prove that any history of operations produced by Chain SimpleStore is linearizable. We take an arbitrary history produced by Chain SimpleStore: $H =< h_1, h_2, ..., h_n >$, where h_i is a {read, write} {request, response} (generically called an operation). We want to find a linearization S of H, *i.e.*, a re-ordering of all operations in H that fulfills three conditions: (1) S is a sequential ordering; (2) S is legal (*i.e.*, reads return in H values that are legal to be returned in S, as well); and (3) if a response precedes a request in H, this ordering is preserved in S.

First, a bit of notation. For any request k, let $\bar{k}$ be the response associated with it. In H, some of the requests have their responses also included in H (such requests are called *finished*), while others do not (*unfinished* requests).

We now construct S. We first arrange all finished operrations in S. We place finished requests (reads and writes) and their responses at consecutive positions in S, in the order in which they were dequeued by SerialDB for handling (using the *commit*, respectively *read* actions). All unfinished requests in H go to the end of S.

We now prove that the resulting S is a linearization of H (*i.e.*, prove each of the three properties). First, all operations in H are executed sequentially in S, thanks to the construction.

Second, clients see the same results of their operations in both H and S. In other words, read responses must return the same value in S as in H. Let $\bar{r}(x)$ be a read response in H, where x is its value. We show that the same read in S returns the same value. To generate the read response, SerialDB must have executed a *read(r)* action, and must have responded with the value of the disk at that time. Let w be the last write that SerialDB committed before dequeuing r for handling it. If w does not exist, then no write can exist before r in S, either. So, in both S and H r should return the non-value (initial value of the disk). So we are left with w exists. Then, when executing r, SerialDB must have returned the value of w. Since SerialDB commits all writes it dequeues, then w must also be the last write that SerialDB dequeued before dequeuing r. So, in S, $(w,\bar{w})$ is the last write before $(r,\bar{r})$. Therefore, r returns the value of w in S, as well.

Third, let $\bar{h}$ be a response and k a request, such that $\bar{h}$ precedes k in H. We must show that $\bar{h}$ precedes k in S, as well. If k is unfinished in H, then it is clear that $\bar{h}$ is before k in S, as well, since all unfinished requests are placed at the end of S. We are left with the case where k is finished in H. Since $\bar{h}$ precedes k in H, then $\bar{h}$ precedes $\bar{k}$ in H. So, SerialDB must have dequeued h before it dequeued k for handling. So, according to S' construction, h and its response must be placed before k and its reponse in S, so $\bar{h}$ precedes k in S. (*q.e.d.*)

1.2 Linearizability of Niobe SimpleStore

Niobe SimpleStore (Figure **??**) provides linearizability. Let H be an arbitrary history produced by Niobe SimpleStore. We need to construct linearization S. We first arrange the finished operations in S. We place finished *write* requests and their responses at consecutive positions in S, in the order in which the requests were inserted into the pending_wrreq ordered channel. A finished *read* request r and its response $\bar{r}$ are placed at consecutive positions in S, right after the response of the last write that was committed by SerialDB (using the *commit* action) prior to SerialDB executing the read r (the *read* action). If no such a write exists, then we place r, $\bar{r}$ at the beginning of S. Ties between reads are broken according to the order in which they were executed by SerialDB. Finally, place all unfinished requests in H in any order at the end of S.

We now prove that the resulting S is a linearization of H. First, all operations in H are executed sequentially in S (no overlapping operations exist, except for the unfinished ones). This property is clearly ensured by the construction of S.

Second, clients see the same results of their operations in both H and S. In other words, read responses must return the same value in S as in H. Let $\bar{r}(x)$ be

a read response in H, where x is its value. We show that the same read in S returns the same value. To generate the read response, SerialDB must have executed a *read* action, and must have responded with the value of the disk at that time. So, x must be that value. We have two cases.

Case 1. X is the non-value, meaning that no writes were committed prior to read r being executed (thus prior to $\bar{r}$). Therefore, according to S' construction, it must be that r is not preceded by any writes in S (since we place all reads with no committed writes before them at the beginning of S). Thus, in S, read r should return the non-value, as well.

Case 2. X is a value, meaning that there must have existed a write w which was committed by SerialDB before it executed r. So, in H, there must exist a $\bar{w}$ before $\bar{r}$. Let the *last* write that was committed by SerialDB before executing r be w. Since reads are executed by SerialDB by responding with the current disk value and the only action that modifies the disk value is *commit*, the read r must have returned the value of w in H. In S, the read r must have been placed right after w (ignoring other reads that might be in between, because they do not modify the disk). Thus, the value of w is the legal value to be returned by r in S.

Third, let $\bar{h}$ be a response and k a request, such that $\bar{h}$ precedes k in H. We must show that $\bar{h}$ precedes k in S, as well. Let's eliminate the simplest cases first. If k does not finish in H, we must have placed k in S after any response to any request (since unfinished requests are placed at the very end). Therefore, $\bar{h}$ precedes k in S. If k has a response in H, we have four cases.

Case 1. h is a read and k is a read. Since h finished in H before k was even issued, it must be that $\bar{h}$ precedes $\bar{k}$ in H. In other words, SerialStore executed h before it executed k. Thus, in S, we must have placed (h, $\bar{h}$) before (k, $\bar{k}$). So, $\bar{h}$ precedes k in S.

Case 2. h is a read and k is a write. In S, h and its response are placed right after the last committed previous write w, with no other writes in between. Obviously, if no such w exists, it means that in S, h and its response must have been placed at the beginning of S, before any writes. So, $\bar{h}$ precedes k in S. So now the case where w exists. We now show that it must be the case that w precedes k in S. Since $\bar{w}$ precedes $\bar{h}$ in H, and $\bar{h}$ precedes k in H, then $\bar{w}$ precedes k in H. So, at the time w was committed, k was not even issued, so it must have been the case that k was inserted in the pending_wrreq queue later than w. So, since writes in S are ordered according to their insertion order in pending_wrreq, w and its response precede k in S. Since there cannot be any write in between $\bar{w}$ and (h,$\bar{h}$) in S, it follows that $\bar{h}$ precedes k in S.

Case 3. h and k are writes. Since h finished before k was even issued, it must have been the case that h was inserted into pending_wrreq before k was, and so h and its response are ordered before k in S. So, $\bar{h}$ precedes k in S.

Case 4. h is a write and k is a read. Let w be the last committed write prior to k's execution by SerialDB (so prior to $\bar{k}$ in H). Such a write must exist, as follows. The only actions that could have responded to h are *commit* and *respond*. But *respond* for h must have been preceded by a *commit* action that moved h to pending_wrresp. So, in any case, there must have existed a committed write prior or equal to $\bar{h}$ in H, and so prior to k and $\bar{k}$ in H. Therefore, our w exists.

We show that $\bar{h}$ precedes or is equal to w in S; given that we know w precedes k in S, it would follow then that $\bar{h}$ precedes k in S. It suffices to show that h precedes or is equal to w in H, since writes and their responses are ordered according to their issuance order in S. Suppose not. Then, h request comes later than the w request in H. So, since w was indeed committed, SerialDB must have committed w before it either committed or moved h. Since responses to committed values are sent right away (without delay), it must be that the response to w is sent before response to h. So, $\bar{w}$ precedes $\bar{h}$ in H. Since $\bar{h}$ precedes k in H, it must be that $\bar{h}$ is in between $\bar{w}$ and k. This cannot be the case unless h was not committed, but rather moved by the commit for another later write (call it w'). Clearly, w' fullfills the following properties: (1) it is committed, (2) it comes later than w in H, and (3) $\bar{w'}$ precedes $\bar{k}$ in H (w' must have been responded to earlier than read k, because it must have been responded to earlier than write h – committed writes are not delayed). At this point we reached contradiction, because $\bar{w}$ is the last committed write before $\bar{k}$ in H, but the above three points indicate that there must exist another write (w') that was committed in between $\bar{w}$ and $\bar{k}$. (*q.e.d.*)

1.3 GFS SimpleStore offers Regular Semantics

GFS SimpleStore (Section **??**) provides regular register semantics. An implementation of a regular register ensures that a read returns either one of the values written by most recent writes, or one of the overlapping writes (if any) [**?**]. By the most recent writes we mean all those writes that were overlapping with the most recent write in response order (last responsed write). See Figure **??** for an example that illustrates

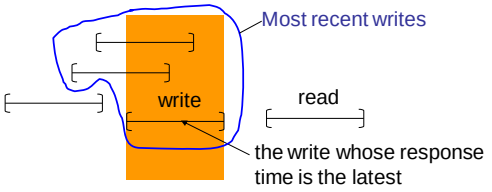

Figure 1: **Most recent writes notion in regular registers.** (Borrowed from a course material [**?**].) The Figure demonstrates the notion of the most recent writes in regular register semantics. It shows a read that does not overlap any writes and a set of writes. The read will return the value of one of the most recent writes (circled writes). Those are all the writes that overlap the most recent write in response order prior to the read request.

the most recent writes and the last responsed write before a read.

To show this fact, we pick an arbitrary history H produced by GFS SimpleStore and show that any read complies the regular register semantic. There are obviously two cases, one for reads non-overlapping any writes in H and the other for reads that do overlap some writes.

Case 1. Let r be a read that does not overlap any write in H. In GFS SimpleStore, all pending writes at a certain moment of time are in one of the following channels: pending_wrreq, pending_wrresp, or failed_wr. These are all writes that have not yet finished. Since r does not overlap any writes in H, then from the moment r was introduced in pending_rdreq until the moment it was processed and responded to by SerialDB, there were no pending writes, *i.e.*all the write channels were empty. Therefore, SerialDB must have responded to r with the value of its disk. Let w be the last write *committed* by SerialDB prior to dequeuing r (so prior to $\bar{r}$ in H).

Case 1.1 If no such w exists, no write has been committed prior to r's response, and so no write could have been committed prior to r's request, either (since r does not overlap any write). So, the value of the disk was clearly the non-value at the time r was requested, and the same value must have been preserved till r was executed. So, r returns non-value, which is indeed the value of the last written writes.

Case 1.2. W exists. Since the only writes that touch the disk are the committed ones and these writes are responded to without delay, then the disk must have contained the value of w at the time r was dequeued by SerialDB for handling. Therefore, r must have returned the value of w. We now need to show that w is among the most recent writes prior to r request. Call mrw the set of most recent writes and lrw the last responded write before r. Suppose $w \notin mrw$. Then, w cannot overlap lrw. So, w must be before lrw, because if it were in between it and $\bar{r}$ it would have been the last responsed write. It follows that w must have been requested and finished before lrw was requested. But since lwr was responded to, it must have been the case that between lwr and its response $\bar{lrw}$, a write w' was committed by SerialDB (this is the only chance for a write to get responded to). So, w' was committed after lrw was requested but before lrw was responded to (or $w' = lrw$). In any case, it follows that w' was committed after w was committed, but r was responded to. This leads to contradiction, because we now have w' committed after the last committed write before $\bar{r}$.

Case 2. Let r be a read that overlaps a set of writes, sw, in H. The interesting case is if at the time of r's response ($\bar{r}$ time), either of the writes channels is non-empty. The other case can be treated in a similar manner as Case 1.2 (nowhere did we use the fact that r cannot overlap any write). So, we are left with the case when at the time of $\bar{r}$, either of the write channels is non-empty. Then, SerialDB would respond to r with one of the values in the pending_wrreq, pending_wrresp, or failed_wr, or with the value of the last committed write before $\bar{r}$. The last case can be proven to adhere regular semantics similarly to Case 1.2. So, let us consider the case where r returns a value in one of the channels. As mentioned before, all these values are pending writes, thus writes that overlap read r. Thus, once more SerialDB replies to the read with a "regular" value. (*q.e.d.*)