Proof of the FAW Protocols

In this document, we demonstrate the correctness of the FAW protocol (§1), its variant for two-phase commit (2PC) (§2) and the RainTx transaction protocol (§3). Also note two *errata* of our original paper (§4).

1 FAW Write Procedure

The form of the three-phase update procedure results from safety constrains. We first establish a *generic* write procedure that models *all possible* write paths. Then we enforce the safety constrains on the generic model and show that the FAW write procedure is necessary and optimal.

1.1 Generic Write Model

We consider a stripe of n chunks using a k-of-n fault tolerance scheme. A write request modifies u chunks (including the m redundant chunks). A traditional logging method has to duplicate all u chunks, while our procedure allows δ chunks to be updated in-place, saving δ chunk writes.

After logging and updating $u - \delta$ chunks, the remaining steps of the write procedure can be viewed as a repeat of a step that in-place updates some of the δ chunks. We assume, in the ith step, ϵ_i chunks are updated in-place in parallel, and before that step, d_i chunks have been updated in-place by previous steps ($d_1 = 0$ denotes the first step). Therefore, $d_{i+1} = d_i + \epsilon_i$, for $i \in \mathbb{N}^+$. If $d_i + \epsilon_i < \delta$, that means not all δ chunks can be updated in the ith step, so we have to do a next step; otherwise, the write procedure ends. Various write paths result from choosing a different ϵ in every step. A write path consists of a series of steps $(0, e_1), (d_2, \epsilon_2), ..., (\delta, 0)$: in the first step, we update e_1 chunks while no chunks have been updated; in the second step, we update ϵ_2 chunks while d_2 chunks (which equals to ϵ_1 in this step) have been updated; so on and so forth; finally, we have updated all δ chunks and thus we should update no chunks in the end step. We visualize the steps in an ϵ-dimensional space (see Figure 1). A point in the figure represents a step in a write path.

1.2 Optimization and Constrains

We formalize the problem of finding the optimal and safe write procedure of FAW as an optimization problem. In this optimization problem, the *objective* is to maximize δ, while the *constraints* are to guarantee the safety of data.

Constraints. To express the constraints, we have to count the chunks of different versions, as summarized in Table 1. Note that a chunk is regarded as holding both new and old versions ("new/old") when it is logged and updated; also, if it does not change at all in the write path ("not updated"), it is regarded as holding both versions. A chunk holds none ("–") if it is being updated in-place.

As described in our original paper, a distribution of f is a tuple (f_0, f_1, f_2), where f_0 failures destroy chunks holding both versions, f_1 failures destroy old-version only chunks, and f_2 destroy new-version only chunks. Then, the invariant is equivalent to satisfying one of the following two constraints.

C1 – the number of old-version chunks is greater than or equal to k: The number of old-version chunks is $(n-u) + (u-\delta) + (\delta - d_i - \epsilon_i)$, as Table 1 lists. Among all f failures, $f_0 + f_1$ destroy old-version chunks. So, we must have $(n-u) + (u-\delta) + (\delta - d_i - \epsilon_i) - (f_0 + f_1) \geq k$ to recover using *roll-back*.

C2 – the number of new-version chunks is greater than or equal to k: Similarly, summing up all new-version chunks as in Table 1 and considering $f_0 + f_2$ failures that destroy new-version chunks, we must have $(n-u) + (u-\delta) + d_i - (f_0 + f_2) \geq k$ to recover using *roll-forward*.

We construct the worst-case failure distribution by assuming an *adversary* who attempts to make best use of the f failures to maximize damaged data.

State	# of chunks	Version
not to be updated	$n-u$	new/old
logged and updated	$u-\delta$	new/old
updated in-place	d_i	new
being-updated in-place	ϵ_i	–
not-yet-updated in-place	$\delta - d_i - \epsilon_i$	old

Table 1: Chunk states. A chunk holds the old or/and the new version of data.

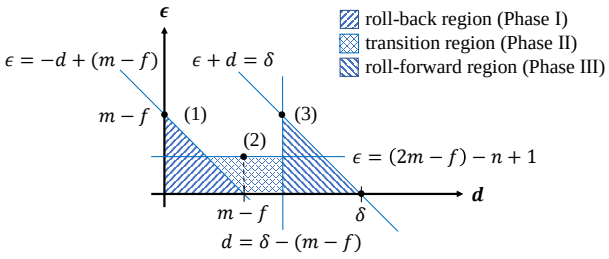

Figure 1: FAW steps in a $d-\epsilon$ space. A point (d, ϵ) means one step to update ϵ chunks when d chunks have been updated. Constraints produce feasible regions (e.g., the left side of the line $\epsilon = -d+(m-f)$ satisfies $\epsilon \leq -d + (m - f)$, which is Inequality 1). Points (1) – (3) represent three steps of a write path.

Our protocol and the adversary are *gaming* against each other.

The first priority of the adversary must be to increase f_0 because one such failure can cause damage to *both* versions anyway. Consequently, if the number of chunks with both versions is larger than f, the adversary must set $f_0 = f$, and $f_1 = f_2 = 0$. As a result, C1 becomes:

$$\epsilon_i \leq -d_i + (m - f). \tag{1}$$

And C2 becomes:

$$d_i \geq \delta - (m - f). \tag{2}$$

Otherwise (the number of chunks with both versions is less than f), the adversary would maximize f_0 by setting it to the number of chunks with both versions, i.e., $f_0 = (n - u) + (u - \delta)$. Since f_0 is determined, C1 becomes $f_1 \leq m - (n - \delta) - d_i - \epsilon_i$. In addition, substituting $f - f_0 - f_1$ for f_2, C2 becomes $f_1 \geq \delta - (m - f) - d_i$. The adversary must seek f_1 within the scope $(m-(n-\delta)-d_i-\epsilon_i, \delta-(m-f)-d_i)$ so that neither C1 nor C2 is satisfiable. Therefore, we have to ensure the scope is empty. As the value of f_1 is discrete, that requires $m-(n-\delta)-d_i-\epsilon_i \geq \delta - (m - f) - d_i - 1$, which is:

$$\epsilon_i \leq (2m - f) - n + 1. \tag{3}$$

In Figure 1, we color all feasible (d_i, ϵ_i) points that satisfy the constraints. The left triangle region always supports roll-back in *any distribution* of failures, because any point/step therein can survive the most adversarial case which leads to Inequality 1. Similarly, any point/step in the right triangle region is constantly recoverable by roll-forward. Between them may exist a *transition region* where we cannot say for sure whether recovery will be via rolling forward or backward. One or the other will surely work, and the exact choice depends on the failure distribution (particularly, the choice of f_1 as discussed for Inequality 3).

1.3 Procedure and Proof

Having the model and constraints, we now solve the optimization problem: find the maximum δ so that there is a safe write procedure. A write procedure is safe as long as every step is recoverable, i.e., located in the feasible regions in Figure 1. As long as $\epsilon = (2m-f)-n+1 \geq 1$, i.e., $n \leq 2m-f$, all three regions connect. That means any step can find a feasible ϵ until $d = \delta$ which is the end of a write path. If so, there is no safety restriction on the value of δ, which can be up to u.

Otherwise, when $n > 2m-f$, the transition region is empty. But if the roll-back region and the roll-forward region connect, a write path always exists. Thus, we need $\delta-(m-f) \leq m-f$, i.e., $\delta \leq 2(m-f)$. All in all, the maximal δ is as follows.

$$\delta_{\max} = \begin{cases} u, & \text{if } n \leq 2m - f; \\ \min\{2(m - f), u\}, & \text{otherwise.} \end{cases} \tag{4}$$

Next, we determine the optimal write procedure. As Figure 1 shows, when $d = 0$, we take the first step. To reduce the overall latency, we update as many chunks as possible in parallel. As a result, this step will reach from $d = 0$ to $d = m - f$, bypassing the entire roll-back region. This step corresponds to Phase I of FAW. Hence, the number of chunks to update in this phase is $s_1 = m - f$. Then, we go through the transition region where $\epsilon = (2m - f) - n + 1$. Different from a triangle-shaped region, it is possible we need more than one step to get across the region. These steps make Phase II. Finally, we can go through the roll-forward region in a single step with $s_3 = m - f$ in Phase III. In conclusion, FAW produces a safe write procedure whose every step is recoverable, with maximum δ as well as minimal latency (steps).

2 FAW-2PC Protocol

We aim to prove that, after the logged Phase II, a stripe is recoverable by both roll-back and roll-forward.

If Phase II does not exist, i.e., $\delta \leq 2(m - f)$, the roll-back region and the roll-forward region in Figure 1 overlap. The state of the chunks after Phase I is $(m - f, 0)$. Note that a point with $\epsilon = 0$ denotes a state (or a "step" without updating any chunk). The

point falls in both the roll-back region and the roll-forward region, so the state is recoverable by both roll-back and roll-forward.

If $\delta > 2(m-f)$, Phase II updates $s_2 = \delta - 2(m-f)$ chunks in-place. FAW-2PC requires the s_2 chunks to be logged as well, so only the rest $2(m-f)$ chunks avoid logging. Accordingly, the state after Phase II is as Table 2 lists. We examine the state with C1 and C2. Both C1 and C2 translate to $n-(m-f) \geq k$. In the worst case, all f failures destroy the old-version/new-version chunks for C1/C2, so C1/C2 becomes $n-(m-f)-f \geq k$, i.e., $n-m \geq k$. Since $n-m=k$ by definition, the inequality is constantly true. Therefore, both C1 for roll-back and C2 for roll-forward are meet.

Overall, after a logged Phase II (no matter whether it is empty), the chunks can always roll back or roll forward, if 2PC requires either.

State	# of chunks	Version
not to be updated	$n-u$	new/old
logged and updated	$u-2(m-f)$	new/old
updated in-place	$m-f$	new
being-updated in-place	0	–
not-yet-updated in-place	$m-f$	old

Table 2: Chunk states after a logged Phase II.

3 RainTx Protocol

The serialization point of a RainTx transaction is the same as in FaRM, because the integration with RAIN does not change the concurrency control behavior. Particularly, the serialization point is when all the write locks are acquired for a read-write transaction, and when the last read is performed for a read-only transaction. The reasoning behind this statement can be found in §4-Correctness of the original FaRM paper [1].

To ensure serializability across failures, a transaction is regarded as *committed* in FaRM after Step ③, i.e., all have been prepared (see Figure 5(a) in our original paper). The logs ensure the commit decision to be able to survive failures. Similarly, the commit point of RainTx is after its Step ③, i.e., after Phase I and logged Phase II. As demonstrated in §2, RAIN guarantees the commit decision is safe.

Note that, to survive a failure of the coordinator, region IDs of the transaction have to be included in the logs of FaRM or the version records of RainTx. Upon recovery, the region IDs are used to find proper logs or version records to judge if all regions have been prepared in FaRM or passed Phase II in RainTx, respectively. If so, the transaction must be committed by replaying the logs in FaRM or by rolling forward in RainTx.

Finally, our implementation of FaRM's protocol is based on two-sided RDMA, so we do not need the special treatment for reserving log spaces as described in the original FaRM paper.

4 Errata

- Figure 5(b) of our original paper should be replaced with Figure 2. The original figure does not match our description of Step ⑤ in the text, though it is also a correct protocol – it shows an optimization we planned but did not get enough time to implement before the paper submission. The optimization is to unlock primaries as early as possible without waiting for the coordinator's notifications. Note that our evaluation of RainTx follows Figure 2 without the optimization. Its Step ⑤ is similar to Step ⑤ of FaRM.

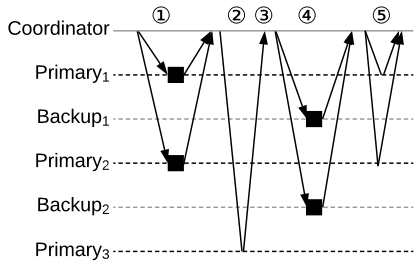

Figure 2: The commit protocol of RainTx.

- In Figure 8(a) of our original paper, "FM-thr" and "RX-thr" should be swapped. The throughput of RainTx (RX-thr) is higher than that of FaRM (FM-thr).

References

[1] Dragojević, A., Narayanan, D., Nightingale, E. B., Renzelmann, M., Shamis, A., Badam, A., and Castro, M. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP '15, pp. 54–70.