A Correctness

In this Appendix, we discuss the correctness of Cabinet in §A.1 and leader properties §A.2. Since Cabinet is implemented based on Raft, we show the added weight mechanism does not affect correctness; i.e., Cabinet maintains Raft's safety and liveness properties.

A.1 Correctness arguments

Cabinet inherits Raft's INTEGRITY property, where every correct node decides at most one value. The imposed weight mechanism does not change the rule of nodes accepting values in the `AppendEntries RPCs`. In addition, Cabinet maintains the validity in the consensus process.

Theorem A.3 (Validity). *If some correct node decides a value v, then v is the initial value of some node.*

Proof. Cabinet's weighted consensus requires a minimal size of cabinet members of 2 where $t=1$ (the lower bound). Thus, at least one cabinet member must be correct under any choice of t. If a value v is committed, at least one correct cabinet member has agreed on v. Therefore, v must have been proposed by some node. □

Cabinet adopts Raft's leader election mechanism with a modification of the election quorum size to $n-t$ (see §4.1.3). This modification is in accordance of Cabinet's customized failure threshold. Cabinet does not impose further changes to Raft's original specifications and maintains the correctness properties of Raft's leader election mechanism.

We first show that the modified quorum size still enforces the election of the most up-to-date nodes as a new leader.

Lemma A.3. *By requiring an election quorum of size $n-t$, an elected leader in Cabinet must have the most up-to-date log.*

Proof. We partition the n nodes into two groups: $|G_1|=t+1$ and $|G_2|=n-t-1$, where $|G_1|+|G_2|=n$. Consider a consensus instance replicating a value v. In Cabinet, the minimum quorum size for weighted consensus is $t+1$, which corresponds to all cabinet members (represented by G_1). Nodes in G_1 possess the most up-to-date logs, while those in G_2 lag behind.

In Raft, a node only votes for a candidate that is as least as up-to-date as itself. Thus, a lag-behind candidate can collect at most $n-t-1$ votes (i.e., from nodes in G_2). Since Cabinet's election quorum size is $n-t$, any candidate that does not possess the up-to-date log cannot be elected as a new leader.

When the number of nodes that replicate v is greater than $t+1$, then stall nodes are less than $n-t-1$. Consequently, a stall candidate will receive fewer votes than $n-t-1$, which is a worse case than the aforementioned case. The stall candidate cannot be elected. Therefore, Cabinet ensures that an elected leader must have the most up-to-date log. □

Lemma A.4. *Cabinet maintains Raft's election safety; that is, at most one leader can be elected in a given term.*

Proof. In Raft, each node only votes once in a given term and votes for a candidate that is as least as up-to-date as itself. Thus, there can only be one elected candidate that has the most up-to-date log. When votes are split among eligible candidates, Raft simply waits for new timeouts that increase the current term and repeats the election process until a new leader is elected.

Cabinet adopts all the above policies without any changes. Since Cabinet's failure threshold t is always less than f (i.e., $1 \le t \le f = \lfloor \frac{n-1}{2} \rfloor$; see §4.1.1), at most one candidate can collect $n-t$ votes in a term. Thus, Cabinet cannot elect more than one leader in a given term, thereby inheriting Raft's election safety. □

In §3, we have shown the unique properties of fast agreement and fault tolerance of weighted consensus in one consensus execution. Now we sketch the proof for Cabinet's safety.

Theorem A.4 (Safety). *No two correct nodes decide differently.*

Proof. (Sketch) From Lemma A.3 and A.4, Cabinet inherits the correctness of Raft's leader election mechanism; thus, the *election safety* and *leadership completeness* introduced in Raft are maintained. Once a leader is elected, the replication process commences with nodes assigned distinct weights in each consensus execution.

Raft's leader election mechanism guarantees that a new leader comes from the most up-to-date nodes in the system. As weight clocks monotonically increase in Cabinet's AppendEntries RPCs, a new leader must have the highest weight clock of the latest system-wide committed value. Consequently, the new leader will not overwrite previous weight clocks, ensuring that weight distinctness is maintained throughout the consensus process.

Given Lemma 3.1, in each weight clock, non-cabinet members cannot decide on a different value when cabinet members have decided. Therefore, Cabinet prohibits correct nodes from deciding differently. □

Theorem A.5 (Liveness). *A correct node eventually decides.*

Proof. We show that Cabinet's liveness property is equivalent to Raft's liveness property. We discuss the equivalence in leadership changes and replication.

By Lemma A.4, Cabinet is able to reuse Raft's leader election mechanism. Thus, the liveness in leadership changes in Cabinet is equivalent to that in Raft. In addition, by Theorem 3.2, in each consensus execution, a Cabinet system with n nodes can at least tolerate t failures ($t \le f = \lfloor \frac{n-1}{2} \rfloor$, where f is the failure threshold in Raft). Thus, the combination of leader election and replication in Cabinet can at least tolerate t failures. Consequently, Cabinet upholds the same liveness property as Raft. □

Theorem A.5 shows the lower bound liveness of Cabinet. As discussed in §4.2, Cabinet may exhibit stronger liveness than Raft by operating under more than $f = \lfloor \frac{n-1}{2} \rfloor$ node failures as long as the total weights of remaining nodes exceed its consensus threshold.

A.2 Leader properties

Expanding on the earlier discussion regarding the properties of new leaders (previously addressed in §4.1.3), Cabinet's weight reassignment strategy prioritizes log responsiveness. In each consensus instance, the nodes that have the most up-to-date logs are the cabinet members; thus, a leader will be elected from the cabinet members of the latest consensus instance of replication.

nodes	1	2	3	4	5	w_i
n_1	x	x	x	x	x	w_1
n_2	x	x	x	x	x	w_2
n_3	x	x	x			w_3
n_4	x	x	x			w_4
n_5	x	x				w_5

Figure 18. A possible log snapshot of Cabinet cluster of 5 nodes with $t = 2$. n_1 and n_2 are cabinet members in Instance 5.

For example, consider a Cabinet system with $n = 5$ and $t = 2$, illustrated by the snapshot of node logs depicted in Figure 18, where 'x' denotes nodes that have replicated the entry coordinated by the leader. Assume node n_1 is the leader and fails. With a failure tolerance of $t = 2$, the system can tolerate two failures, so a leader election will take place.

According to Raft's leader election mechanism, only n_2 can be elected as the new leader, given that it has the most up-to-date log and can receive votes from all the remaining nodes (i.e., $n - t$, including itself). Prior to the election, the most up-to-date log must be replicated by the cabinet members, which in this case are nodes n_1 and n_2. Therefore, the elected leader must emerge from the cabinet members during the latest replication phase before the election occurs.

In heterogeneous systems, it is common for a strong node to possess the most up-to-date log and be selected as a new leader. While this outcome is often expected, it cannot be guaranteed. For instance, in complex networks, strong nodes may encounter high delays, rendering them less responsive. In such scenarios, during an ongoing consensus instance, strong nodes might be assigned lower weights. Consequently, during an election at such moments, a weak node, despite having the most up-to-date log, could potentially be elected as the new leader.

Nevertheless, in typical scenarios where strong nodes are more responsive due to their processing capability, Cabinet can increase the likelihood of a new leader emerging from a pool of strong nodes. This emphasis on selecting leaders from nodes with the most up-to-date logs contributes to the overall robustness and performance of the system.

B Reconfiguration for failure thresholds

In Cabinet, the failure threshold (t) plays a pivotal role in determining the weight scheme (WS) and consensus threshold (CT) used for consensus operations. Given the customizable nature of the failure threshold, applications may find it necessary to adjust t to adapt to evolving operating conditions. For instance, there may be periods where applications prioritize higher fault tolerance, while in others, they prioritize higher performance, anticipating fewer failures.

To cater to such dynamic needs, Cabinet introduces a *lightweight reconfiguration* mechanism for changing failure thresholds. Unlike Raft's reconfiguration method, which focuses on changing the cluster size [45] (i.e., changing n), Cabinet's lightweight reconfiguration is geared towards changing the failure threshold t while keeping n unchanged. This lightweight reconfiguration process is executed by the leader utilizing `ReconfigRPCs`.

```
struct ReconfigRPC:
    ...    // ReconfigRPC keeps the parameters
    ...    // of Raft's AppendEntriesRPC but
    nilEntry // removes Entires

    wclock   // weight clock
    newT    // the new failure threshold
```

Compared to Raft's `AppendEntriesRPCs`, `ReconfigRPC` keeps the original parameters but does not contain entries for replication (i.e., `Entries = nil`). `ReconfigRPC` is specifically designed for reconfiguring the failure threshold and does not involve replicating entries. The reconfiguration process works as follows.

```
// System is in WS_old. Given a failure
// threshold, newT, the leader generates
// a new WS, WS_new.
WS_new = InitWS(n, t)
wclock++ // increment wclock
Broadcast ReconfigRPCs to followers
If consensus is reached under WS_old:
    set weight scheme to WS_new
    start to operate under WS_new
```

First, the leader initializes a new weight scheme (WS_{new}) based on the given n and the new t, following the same procedure outlined in Line 4 of Algorithm 1. Then, the leader begins coordinating consensus under the old weight configuration. The objective of this consensus instance is to synchronize with followers regarding the new consensus threshold. Upon successfully reaching consensus, the leader transitions its weight scheme from WS_{old} to WS_{new}. Following this update, replication resumes under WS_{new}, with the leader transmitting each follower w_i contained in WS_{new}.

The lightweight reconfiguration is specifically designed for changing the failure threshold, not for changing the cluster size. In scenarios where both the cluster size and failure thresholds need to be changed, the system can execute Raft's reconfiguration method to change the cluster size first. Then, a lightweight reconfiguration can be performed to change the failure threshold under the new cluster configuration. This sequential approach ensures smooth transitions while maintaining system stability.