Orcaella: Hybrid Fault Tolerance with Client-Selectable Finality Latency

Lefteris Kokoris Kogias, Alberto Sonnino
Mysten Labs
{lefteris, alberto}@mystenlabs.com

***Abstract*—Classical partially synchronous state machine replication, as in PBFT [1], tolerates f Byzantine replicas among $n \geq 3f+1$ using three communication steps per request. Recent protocols such as Minimmit [2] achieve *two-message-delay* decisions under stronger size assumptions, notably $n \geq 5f+1$ when any silent replica must be counted as a potential equivocator. Hydrangea [3] and Kudzu [4] treat mixed Byzantine and crash faults, focusing on providing a fast-path under optimistic conditions while maintaining a fallback commitment path similar to PBFT. In this paper, we also consider a mixed model, but focus on studying the fault tolerance of the 2-message-delay commit. For this, we prove a tight bound of $n \geq 5f+3c+1$. Extending this result, we also show that there exists a more resilient commit path that allows an extra $f_{abc} < n-3f-2c$ alive-but-corrupt [5] faults at 4-message-delays. Core liveness is claimed in executions with at most f equivocators; if this regime is violated (e.g., AbC-induced forks), the protocol enters synchronous recovery, where only the resilient-path safety guarantee is preserved. As a result, for $f=16$, $c=6$, and $n=99$, we obtain a commit path that tolerates 22% of replicas failing for liveness, 16% equivocating for 1-RTT safety, and 54% equivocating for 2-RTT safety.**

I. Introduction

State machine replication (SMR) protocols are the foundational building blocks of modern decentralized systems [6]. Under partial synchrony [7], classical solutions like PBFT [1] tolerate up to f Byzantine replicas among $n \geq 3f+1$, but require three all-to-all communication delays to reach consensus. As decentralized applications demand ever-lower latency, recent research has introduced *two-message-delay* protocols (e.g., Minimmit [2]). However, achieving this optimal fast-path latency traditionally requires a much larger committee size of $n \geq 5f+1$, as every silent replica must be conservatively treated as a potential Byzantine equivocator.

The challenge. To mitigate the severe $5f+1$ requirement, a natural approach is to distinguish between fully Byzantine faults (who may equivocate) and crash faults (who merely go silent). Recent work like Hydrangea [3] and Kudzu [4] adopt this mixed fault model. However, existing hybrid systems focus on providing a fast-path under optimistic conditions, while eagerly falling back to a slower, PBFT-style 3-round path when faults increase.

This leaves an open challenge: *How can we maximize the fault tolerance of the optimal 2-message-delay commit itself when relaxing the fault model?* Furthermore, if a client does not care about this lower latency, *can we enhance the mixed fault model to provide even stronger, FlexibleBFT-style [5] safety guarantees?*

Our solution. In this paper, we answer these questions by formally defining the tight quorum intersections required to maximize 2-message-delay commits under separate Byzantine (f) and crash-faulty (c) caps. We show that such a system necessitates:

$$n \;\geq\; 5f+3c+1$$

with an optimal fast-path quorum of $q = n-f-c$, i.e., $4f+2c+1$ at the minimal committee size. At $c=0$, this recovers the familiar $5f+1$ regime, but when crashes are separated, it allows for configurations with better liveness guarantees.

Building on this foundational result, we introduce Orcaella, a hybrid protocol that exposes two explicit finality paths for clients. These paths are built using *Quorum Certificates* (QCs)—cryptographic proofs consisting of q matching signed messages from distinct replicas:

- **Optimal Fast-Path (2-delay):** Clients can finalize quickly after collecting a single quorum of votes (VoteQC), providing safety against f Byzantine faults.
- **Resilient Path (4-delay):** Clients willing to wait for two additional rounds (chaining a CheckpointQC and FinalityQC) gain resilience against an additional f_{abc} *alive-but-corrupt* [5] replicas. These are replicas that participate but may equivocate.

Crucially, the resilient path provides safety when the core execution assumptions are violated and forks occur. In executions with at most f equivocators, the protocol retains its normal liveness behavior. If this regime is violated (e.g., AbC-induced forks), liveness is temporarily lost, replicas enter synchronous recovery, and resilient-path safety remains preserved throughout.

Concrete trade-offs. This dual-path architecture allows operators to explicitly navigate the latency-safety-liveness trade-off. Depending on the expected environment, for a deployment of $n \approx 100$ replicas, an operator can tune their configuration (Figure 1):

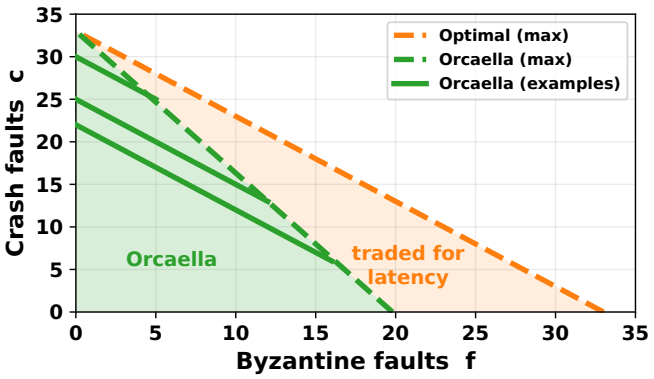

Fig. 1: Fault-tolerance design space at $n = 100$. Typical $3f + 1$ protocols treat every fault as Byzantine, covering the region $f+c \leq 33$ (orange) at the cost requiring 3 message delays to commit. Orcaella instead covers the region $5f + 3c + 1 \leq n$ (green), trading some fault tolerance for its 2-message-delay commit. Each solid green line is one example config's runtime frontier: a cap (f, c) tolerates any mix down to $(0, f + c)$.

- **Balanced ($f = 12, c = 13$):** Tolerates up to 25 offline replicas for liveness, ensures 1-RTT safety against 12 Byzantine faults, and 2-RTT safety against a maximum of 49 total equivocators (strictly below the intersection bound $T = 50$).
- **Byzantine-heavy ($f = 16, c = 6$):** Optimizing for more malicious conditions slightly reduces liveness tolerance (22 offline replicas) but boosts 1-RTT safety to 16 and 2-RTT safety to a maximum of 54 total equivocators ($< T = 55$).
- **Crash-heavy ($f = 5, c = 25$):** Assuming a more benign but flaky network maximizes liveness (tolerating 30 offline replicas) while still offering 2-RTT safety against a maximum of 40 total equivocators ($< T = 41$).

This flexibility allows deployments to adapt to their exact threat models without sacrificing the optimal 2-message-delay commit. Although not the focus of this work, Orcaella can be configured with $f = 0$ and $n = 2c + 1$ to provide a CFT variant without any code changes other than configuring the quorums. Appendix D shows an evaluation of this variant for completeness.

Contributions. We make the following contributions:

1) A characterization of *vote-counting* protocols with tight quorum inequalities ($n \geq 5f+3c+1$) that maximize the fault tolerance of 2-delay BFT consensus (Theorem 1).
2) Orcaella, a protocol that exposes the 2-delay and 4-delay paths, including a fully specified view change and fork recovery mechanism.
3) FlexibleBFT-style client analyses formalizing the safety vs. liveness trade-offs and proving the 4-delay path remains safe against an additional $f_{abc} < n - 3f - 2c$ alive-but-corrupt faults.
4) OrcDAG, a DAG-based instantiation of Orcaella over an uncertified-DAG fabric, and its evaluation on realistic deployment quantifying the latency gained by trading fault tolerance.

TABLE I: Notation

Symbol	Meaning
n	Total replicas
f	Byzantine (equivocating) cap
c	Crash-faulty cap
f_{abc}	Alive-but-Corrupt cap (Resilient Path client analysis)
q	Quorum threshold, $q = n - f - c$
k	View-Change accept threshold, $k = 2f + c + 1$
σ	State root after applying a slot ($\sigma_s = H(\sigma_{s-1} \| h)$)
Δ	Post-GST bound on message delay

II. Network and Fault Model

We use standard message-passing state machine replication [8] among n *replicas* (the *leader* or *primary* proposes each slot) under partial synchrony [7]. Channels are authenticated; digital signatures identify senders. Each replica is in exactly one of four disjoint classes:

- *Byzantine* (at most f) replicas, which may crash or send arbitrary messages including *equivocations* (inconsistent signed statements to different recipients) on any protocol message;
- *Crash-faulty* (at most c) replicas, which follow the protocol while active and may then permanently stop sending messages without equivocating;
- *Alive-but-Corrupt* (AbC, at most f_{abc}) replicas, which represent a *separate client-side threat model* used only for the Resilient Path analysis, where they may equivocate on any protocol message—votes and checkpoint proposals alike—and can thus cause VoteQC forks, whereas in the core Fast-Path execution model and liveness analysis, AbC behavior is not activated and only up to f replicas may equivocate;
- *Correct* replicas, which remain honest and live throughout the execution under consideration, assuming $n - f - c$ correct replicas for Fast-Path safety/liveness (core protocol) and $n - f - c - f_{abc}$ correct replicas for Resilient Path client-side safety under AbC assumptions.

The standard partial synchrony network model [7] assumes that after an unknown Global Stabilization Time (GST), message delays between correct replicas are bounded by a known constant Δ. The adversary controls all non-honest replicas up to the misbehavior they are allowed to perform and the network scheduling (subject to this bound after GST). It cannot break cryptography.

III. The Orcaella Protocol

This section instantiates the fault model and Fast-Path/Resilient Path split from Section II into a message-level protocol. It uses the thresholds derived in Theorem 1: $n \geq 5f + 3c + 1$ and $q = n - f - c$ (equal to $4f + 2c + 1$ at the minimal n). Replicas communicate on authenticated channels under partial synchrony. The protocol proceeds

in views $v \in \mathbb{N}$, each with a designated leader $\textsc{Leader}(v)$, and assigns proposals to sequence numbers $s \in \mathbb{N}$.

A. The protocol

Algorithm 1 specifies the protocol, covering both normal operation and leader recovery. Normal operation is divided into a Fast-Path (Steps 0–1, lines 2 to 4) that guarantees safety against f Byzantine faults in two message delays, and a Resilient Path (Steps 2–4, lines 8 to 14) that provides extended safety against alive-but-corrupt (AbC) faults at the cost of two additional message delays.

Client finalization paths. Clients finalize transactions by observing the certificates produced by the protocol. The latency bounds stated in this paper (e.g., two or four message delays) assume that clients are collocated on the replicas participating in the consensus protocol. If clients are external, their perceived finality latency naturally increases by the round-trip time (RTT) to communicate with the replica set, as is standard in all BFT protocols.

- **Fast-Path (2-delay):** A client finalizes (v, s, h) immediately upon collecting a VoteQC. This provides optimal latency and safety against f Byzantine faults.
- **Resilient Path (4-delay):** A client finalizes (s, h, σ) upon obtaining a FinalityQC (output by a replica in Step 4, line 14) together with its underlying CheckpointQC. This trades higher latency for extended safety against an additional f_{abc} alive-but-corrupt faults, without interfering with the core protocol's liveness [5].

Protocol description. The leader of view v proposes a batch of transactions. Replicas validate the proposal and cast an accept vote, but only if the proposal is consistent with the branch $\mathcal{O}$ adopted from the latest NewView (line 4); this guard is what carries Fast-Path commits across view changes (Lemma 4). Correct replicas send at most one vote per slot (v, s). If a client gathers q such accept votes, it forms a VoteQC and can immediately safely execute the payload (Fast-Path). If the leader is faulty [9] or the network is asynchronous, replicas eventually timeout and broadcast ViewChange messages, which collectively serve as an implicit abandon for uncommitted slots.

To support the Resilient Path, replicas that observe a VoteQC compute the resulting deterministic state root σ and broadcast a checkpoint proposal. We assume state roots bind the execution history: σ_s commits to the previous root and the applied digest (e.g., $\sigma_s = H(\sigma_{s-1} \,\|\, h)$), so a root at height s determines the checkpointed content at every height below it. **Safety is strictly guarded by Step 2** (line 8): to prevent conflicting checkpoints, an honest replica broadcasts at most one ChkProp per slot height. For this reason, checkpoint artifacts deliberately carry no view number: a slot's VoteQC may form in view v at one replica and in a later view v' at another (with the same digest, by Fast-Path safety), and view-tagged ChkProps would split each replica's single permitted proposal across views, permanently blocking the checkpoint. While this strict locking rule means *liveness* can be lost if honest replicas split their ChkProp messages across different branches (a trade-off we explicitly accept), it provides an ironclad safety guarantee. Once q replicas agree on this checkpoint (forming a CheckpointQC), they lock it by broadcasting a ChkWitness message. Note that ChkWitness messages attest to the checkpoint *content* (s, h, σ) rather than to one specific certificate: two CheckpointQCs over the same content but different signer sets may circulate, and binding witnesses to a particular certificate would needlessly split them. To safely support a checkpoint, an honest replica will broadcast a ChkWitness as long as the checkpoint's state σ matches its own deterministic execution of the payload ($\sigma = \textsc{Apply}(h)$), even if its single permitted ChkProp for that height was already spent on a different, unfinalized branch. Once q such witness messages are collected, a replica outputs a FinalityQC (Step 4, line 14) as its Resilient Path finality decision for that slot.

Notice that with f_{abc} alive-but-corrupt faults, there can be VoteQC forks across different views. However, because honest replicas only broadcast a single ChkProp per height, the double-signing bound (Lemma 7) guarantees that at most one valid CheckpointQC can ever be formed per height. Any resulting loss of liveness on the Resilient Path is cleanly resolved by the Fork Recovery protocol (Algorithm 2) during a synchronous epoch. Steps 2–4 (lines 8 to 14) ensure that this single CheckpointQC is witnessed by the network and the resulting FinalityQC is propagated before resilient finalization. Notice that every protocol artifact (VoteQC, CheckpointQC, FinalityQC) uses the exact same quorum threshold q.

View-Change details. The view change protocol is structurally similar to both PBFT and modern protocols like Streamlet [10] or HotStuff [11], [12]. If $\textsc{Leader}(v+1)$ fails to assemble a valid NewView before a timeout, replicas re-issue ViewChange messages for view $v+2$ with doubled timeouts, following standard view synchronization [1]. Assuming the core limit of f Byzantine faults holds (i.e., there are no VoteQC forks), any two sets of q ViewChange messages intersect at an honest replica ($2q - n > f$). This guarantees that any VoteQC used by a client to finalize a Fast-Path commit is reported to the new leader and preserved in the new view's starting state $\mathcal{O}$. Since replicas may re-vote for the same digest across views, the leader counts the reported votes per digest rather than per view (Lemma 4). By operating over a large $5f + 1$ style quorum, the protocol acts as a generalized 2-chain Streamlet, enabling rapid 2-message-delay finality instead of the standard 3-chain required under $3f + 1$. If the f limit is exceeded (e.g., by AbC faults double-voting) and a VoteQC fork occurs, Fast-Path safety is broken and preserving it is no longer required; instead, replicas halt and rely on Fork Recovery to salvage the Resilient Path. Note that for simplicity, Algorithm 1 requires replicas to send all

Algorithm 1: Orcaella: Normal Operation & Leader Recovery for replica r

Definitions & State Variables:
Thresholds: $q = n - f - c$ and $k = 2f + c + 1$.
VoteQC: q matching VOTE messages for (v, s, h) with accept.
CheckpointQC: q matching CHKPROP messages for (s, h, σ).
FinalityQC: q matching CHKWITNESS messages for (s, h, σ).
current_view $\leftarrow 0$
$\mathcal{O} \leftarrow \emptyset$ ▷ Branch adopted from the latest NEWVIEW ($\perp$ = unconstrained)

(step 0) upon $r = \text{LEADER}(v)$ **and** $v =$ current_view **and** new payload $B_{v,s}$ **do**
  broadcast $\langle \text{PROPOSE}, v, s, B_{v,s} \rangle_r$

(step 1) upon receiving $\langle \text{PROPOSE}, v, s, B \rangle_L$ from $L = \text{LEADER}(v)$ **do**
  if $v =$ current_view **and** r has not yet voted for slot s in view v **then**
    if $\mathcal{O}[s] \in \{\perp, H(B)\}$ **then** ▷ Vote only consistent with the adopted branch
      broadcast $\langle \text{VOTE}, v, s, H(B), \text{accept} \rangle_r$

(step 2) upon observing a valid VoteQC Q for (v, s, h) **do**
  if no CHKPROP has been broadcast for slot s **then**
    $\sigma_s \leftarrow \text{APPLY}(h)$ ▷ Deterministic state transition
    broadcast $\langle \text{CHKPROP}, s, h, \sigma_s \rangle_r$

(step 3) upon observing a CheckpointQC C for (s, h, σ) **where** $\sigma = \text{APPLY}(h)$ **do**
  broadcast $\langle \text{CHKWITNESS}, s, h, \sigma \rangle_r$

(step 4) upon collecting q matching $\langle \text{CHKWITNESS}, s, h, \sigma \rangle$ messages **do**
  output $\langle \text{FINALITYQC}, s, h, \sigma \rangle$ ▷ Resilient Path finality decision for slot s

upon timeout in view v **or** timeout awaiting $\langle \text{NEWVIEW}, v \rangle$ **do** ▷ Timeouts double per attempt
  current_view $\leftarrow \perp$ ▷ Halt normal operation
  $\mathcal{P}^r \leftarrow \{$highest-view $\langle \text{VOTE}, v', s, h, \text{accept} \rangle$ cast by r for each $s\}$
  broadcast $\langle \text{VIEWCHANGE}, v + 1, \mathcal{P}^r \rangle_r$

upon $r = \text{LEADER}(v)$ and receiving q valid $\langle \text{VIEWCHANGE}, v, \mathcal{P}_i \rangle$ messages $\mathcal{V}$ **do**
  for each slot s represented in $\mathcal{V}$ **do**
    $\mathcal{K} \leftarrow \{h \mid h$ is the highest-view vote for s in at least k reports in $\mathcal{V}\}$ ▷ Aggregated per digest, across views
    if $\mathcal{K} \neq \emptyset$ **then**
      $\mathcal{O}[s] \leftarrow \min \mathcal{K}$ ▷ Singleton if slot s has a VoteQC (Lemma 4); ties broken deterministically
    else
      $\mathcal{O}[s] \leftarrow \perp$ ▷ No locked value; safe to propose arbitrary valid block
  broadcast $\langle \text{NEWVIEW}, v, \mathcal{V}, \mathcal{O} \rangle_r$

upon receiving valid $\langle \text{NEWVIEW}, v, \mathcal{V}, \mathcal{O} \rangle_L$ from $L = \text{LEADER}(v)$ **do**
  verify $\mathcal{O}$ against $\mathcal{V}$ ▷ Deterministically recompute $\mathcal{K}$ and $\mathcal{O}$
  current_view $\leftarrow v$ ▷ Enter new view
  adopt branch $\mathcal{O}$ and resume normal operation

their highest-view votes. In a practical implementation, replicas would use CheckpointQCs to prove a globally agreed upon state and safely truncate the required vote history, significantly reducing the size of VIEWCHANGE messages [13].

B. Fork recovery

Algorithm 2 specifies the procedure for detecting and recovering from forks (when the core f Byzantine limit is exceeded but clients rely on Resilient Path safety).

Synchronous network model. Unlike normal operation which relies on partial synchrony, this recovery procedure operates under a strictly synchronous network model. We assume that message delays between correct replicas are guaranteed to be bounded by a known, pessimistic constant Δ_{sync}. Replicas anchor the epoch's lockstep round structure at the delivery of the first valid ALARM (which every correct replica re-broadcasts), so the round boundaries of correct replicas are offset by at most Δ_{sync}—a bounded skew that synchronous broadcast protocols tolerate by standard techniques [14].

Fork recovery details. While the core protocol naturally resists up to f fully Byzantine faults, Resilient Path clients may additionally assume f_{abc} alive-but-corrupt replicas. These replicas behave correctly during normal operation but might double-sign conflicting CHKPROP messages to intentionally cause forks.

However, because honest replicas only broadcast a single CHKPROP per height, the double-signing bound (Lemma 7) explicitly prevents the f_{abc} alive-but-corrupt replicas from successfully forming two conflicting CheckpointQCs. Thus, there can be at most one valid CheckpointQC per height.

If a VoteQC fork occurs (exceeding the core f Byzantine limit), correct replicas immediately detect it via cryptographic evidence, halt normal operation, and force a global repair by broadcasting an ALARM. Upon entering this synchronous recovery epoch, every replica initiates its own Byzantine Broadcast instance (such as Dolev-Strong [15]), n in parallel, to reliably exchange all locally

Algorithm 2: Fork Detection and Repair for replica r

upon observing conflicting VoteQCs or a valid $\langle$ALARM, evidence$\rangle$ **do**
    halt normal operation
    broadcast $\langle$ALARM, evidence$\rangle_r$
    initiate a Byzantine Broadcast protocol (e.g., Dolev-Strong)
    to reliably exchange all locally known CheckpointQCs and VoteQCs

upon broadcast protocol completes **do**
    $\mathcal{C} \leftarrow$ all CheckpointQCs delivered by the broadcast
    if $\mathcal{C} \neq \emptyset$ **then**
        $C_{max} \leftarrow$ the $C \in \mathcal{C}$ with the highest slot ▷ Mathematically guaranteed unique per height
        $\mathcal{O}_{canon} \leftarrow$ branch of C_{max} extended with arbitrary valid VoteQCs
    else
        $\mathcal{O}_{canon} \leftarrow$ arbitrary valid branch from VoteQCs
    resume normal operation from $\mathcal{O}_{canon}$

known CheckpointQCs and VoteQCs; correct replicas then operate on the union of the delivered sets. Because Dolev-Strong does not rely on an honest majority, it guarantees that all correct replicas will output the exact same set of valid certificates, even if the total number of faulty replicas ($f + f_{abc}$) constitutes a dishonest majority ($> n/2$). However, if the deployment guarantees that $f + f_{abc}$ remains a strict minority ($< n/2$), this recovery step can be significantly optimized by deploying an honest-majority synchronous broadcast protocol that terminates much faster than Dolev-Strong's $(f + f_{abc}) + 1$ rounds (e.g., expected constant-round protocols [16] or practical synchronous SMR implementations like Sync HotStuff [14]). When the broadcast protocol completes, replicas collect all delivered CheckpointQCs across all slots. Because there is at most one per height and checkpoints are chain-consistent across heights (Lemmas 7 and 8), they form a single, non-conflicting history. Replicas simply identify the highest CheckpointQC and adopt its branch as canonical, extending it with any arbitrary valid VoteQCs for subsequent slots. If no CheckpointQC was ever formed, they can safely choose either valid VoteQC branch. This guarantees that Resilient Path safety remains unbroken.

IV. Safety and Liveness Proofs

We show that Orcaella satisfies safety and liveness under the network and fault model defined in Section II. Throughout, the quorum threshold is $q = n - f - c$ and the view-change accept threshold is $k = 2f + c + 1$; at the minimal committee size $n = 5f + 3c + 1$, $q = 4f + 2c + 1$. We derive both thresholds directly through these proofs, ensuring a rigorous foundation for the optimal $n \geq 5f + 3c + 1$ bound.

We say a protocol is *vote-counting* if it (i) finalizes a digest on the Fast-Path upon q matching first-round votes, and (ii) recovers from leader failure through a view change in which the new leader re-proposes a digest if and only if it appears in at least k of the highest-view vote reports embedded in q ViewChange messages $\mathcal{V}$.

Theorem 1 (Tight Bound Necessity)**.** *A vote-counting protocol that is live under f Byzantine and c crash faults, and whose Fast-Path commits are safe across view changes, requires $n \geq 5f + 3c + 1$. At $n = 5f + 3c + 1$, necessarily $q = 4f + 2c + 1 = n - f - c$ and $k = 2f + c + 1$.*

Proof. We derive three counting constraints; each is necessary against an explicit adversarial strategy.

Liveness (L): $q \leq n - f - c$. Byzantine replicas may remain silent and crash-faulty replicas may stop at any time, so only $n - f - c$ replicas are guaranteed to respond. Both VoteQC formation and the collection of q ViewChange messages must complete without faulty participation, hence $q \leq n - f - c$.

Reachability (R): $k \leq 2q - n - f$. Suppose a client finalizes h for slot s via a VoteQC with vote quorum Q_{vote}, $|Q_{vote}| = q$. The adversary crashes the (up to c) crash-faulty members of Q_{vote} before the view change and schedules delivery so that $\mathcal{V}$ contains as few members of Q_{vote} as possible. Since $|\mathcal{V}| = q$ and only $n - q$ replicas lie outside Q_{vote}, $\mathcal{V}$ contains at least $2q - n$ members of Q_{vote}, of which up to f are Byzantine and may misreport; only $2q - n - f$ reports for h are thus guaranteed. If $k > 2q - n - f$, the adversary makes slot s miss the threshold, so even an honest new leader sets $\mathcal{O}[s] = \perp$ and proposes a fresh block B' with $H(B') \neq h$. Non-equivocating replicas, who have not yet voted in the new view, vote for B', forming a conflicting VoteQC and violating safety. Hence $k \leq 2q - n - f$.

Exclusivity (E): $k \geq n - q + f + 1$. While a VoteQC for h exists, at least $q - f$ non-equivocating replicas voted for h. Since each ViewChange message carries a single highest-view vote per slot, each non-equivocating replica supports at most one digest for s: at most $(n - f) - (q - f) = n - q$ non-equivocating replicas can report a conflicting digest h', joined by up to f Byzantine misreports, so any $\mathcal{V}$ contains at most $n - q + f$ reports for h'. This bound is realizable: a Byzantine leader of view v equivocates between h and h'; $q - f$ non-equivocating replicas vote for h (completing the VoteQC together with the f Byzantine), the remaining $n - q$ vote for h', and the Byzantine replicas report h' during the view change. By (R) the digest h always reaches k supporting reports; if additionally $k \leq n - q + f$, the conflicting digest h' also reaches k, and a Byzantine new leader may adopt h', overwriting the finalized h. Hence $k \geq n - q + f + 1$.

Combining. (R) and (E) give $n - q + f + 1 \leq k \leq 2q - n - f$,

i.e., $3q \geq 2n + 2f + 1$. Substituting (L), $3(n - f - c) \geq 3q \geq 2n + 2f + 1$, hence $n \geq 5f + 3c + 1$.

Forced thresholds. At $n = 5f + 3c + 1$, the bound $3q \geq 2n + 2f + 1 = 12f + 6c + 3$ gives $q \geq 4f + 2c + 1$, while (L) gives $q \leq n - f - c = 4f + 2c + 1$; thus $q = 4f + 2c + 1$. Then (E) yields $k \geq n - q + f + 1 = 2f + c + 1$ and (R) yields $k \leq 2q - n - f = 2f + c + 1$; thus $k = 2f + c + 1$. Sufficiency of these thresholds is established by the remaining lemmas in this section. □

A. Quorum intersection and Fast-Path safety

Lemma 2 (Quorum Intersection)**.** *Let $n \geq 5f + 3c + 1$ and $q = n - f - c$. The intersection of any two quorums of size q contains at least $2f + c + 1$ non-equivocating replicas.*

Proof. The intersection of two sets of size q in a universe of size n has size at least $2q - n = 2(n - f - c) - n = n - 2f - 2c$. Under the client-side assumption for the Fast-Path (Section II), there are at most f equivocating replicas in the system. Thus, the number of non-equivocating replicas in the intersection (which includes Correct, Crash-faulty, and AbC replicas, as Fast-Path clients assume AbC replicas do not equivocate) is at least $(n - 2f - 2c) - f = n - 3f - 2c \geq 2f + c + 1$, where the last inequality uses $n \geq 5f + 3c + 1$. □

Lemma 3 (Fast-Path Uniqueness)**.** *For any given view v and slot s, there can be at most one valid VoteQC.*

Proof. A VoteQC requires q accept votes from distinct replicas for a specific digest h. Suppose, for the sake of contradiction, that two VoteQCs exist in view v for slot s: one for h and one for h' (with $h \neq h'$). By Lemma 2, their respective quorums intersect in at least $2f + c + 1 \geq f + 1$ non-equivocating replicas. This implies that at least one non-equivocating replica cast an accept vote for both h and h' in the same view v, which strictly violates the definition of a non-equivocating replica. Thus, $h = h'$. □

B. View-Change safety

To guarantee that a finalized Fast-Path commit is never overwritten by a subsequent leader, the View-Change protocol must enforce that any newly proposed branch $\mathcal{O}$ preserves previously committed digests. This is achieved via the view-change accept threshold $k = 2f + c + 1$.

Lemma 4 (View-Change Invariance)**.** *If a valid VoteQC for (v, s, h) is formed, then for any subsequent view $v' > v$, if a non-equivocating replica adopts a branch $\mathcal{O}$ proposed by Leader(v'), it must be that $\mathcal{O}[s] = h$.*

Proof. We proceed by strong induction on v', with the following induction hypothesis (IH): for every view w with $v < w < v'$, any branch adopted by a non-equivocating replica in view w satisfies $\mathcal{O}[s] = h$. (For $v' = v + 1$ this range is empty and the argument below applies unchanged.)

Let Q_{vote} be the quorum of q accept votes forming the VoteQC for (v, s, h), and let $\mathcal{V}$ be the set of q ViewChange messages collected by Leader(v'). By Lemma 2, the signers of Q_{vote} and of $\mathcal{V}$ intersect in at least k non-equivocating replicas.

We first claim that the highest-view vote for slot s reported by any non-equivocating member of Q_{vote} certifies h. Such a replica cast exactly one vote for s in view v, namely for h. Consider any view w with $v < w < v'$ in which it also voted for s: by the consistency guard of line 4, it voted only for a proposal matching the branch it adopted in view w, and by the IH that branch satisfies $\mathcal{O}[s] = h$; hence that vote is also for h. Votes in views below v are dominated by the view-v vote. Its highest-view report for s is therefore (w', h) for some $w' \geq v$, proving the claim. In particular, the at least k non-equivocating members of Q_{vote} whose reports appear in $\mathcal{V}$ all support h, so $h \in \mathcal{K}$.

We now show no conflicting digest enters $\mathcal{K}$. Each ViewChange message carries a single highest-view vote for s, so each replica supports at most one digest. By the claim, every non-equivocating member of Q_{vote} supports h; hence at most $(n - f) - (q - f) = n - q$ non-equivocating replicas can support a digest $h' \neq h$, joined by at most f Byzantine (possibly fabricated) reports. In total, h' gathers at most $n - q + f = 2f + c < k$ reports, so $h' \notin \mathcal{K}$.

Thus $\mathcal{K} = \{h\}$ and the view-change rule forces $\mathcal{O}[s] = h$. Since replicas deterministically recompute $\mathcal{K}$ from $\mathcal{V}$ before adopting a branch, a NewView carrying any other value for $\mathcal{O}[s]$ fails verification and is rejected; hence any branch adopted by a non-equivocating replica in view v' satisfies $\mathcal{O}[s] = h$. □

Theorem 5 (Fast-Path Safety)**.** *If two clients finalize (v, s, h) and (v', s, h') via the Fast-Path, then $h = h'$.*

Proof. If $v = v'$, Lemma 3 guarantees $h = h'$. If $v < v'$, the first finalization implies a VoteQC was formed in view v. By Lemma 4, every branch $\mathcal{O}$ adopted by a non-equivocating replica in view v' satisfies $\mathcal{O}[s] = h$, and by the consistency guard of line 4, such replicas vote in view v' only for a proposal B' with $H(B') = \mathcal{O}[s] = h$. A VoteQC for slot s in view v' contains at least $q - f \geq 1$ votes from non-equivocating replicas; hence it certifies h. Thus, $h = h'$. □

C. Liveness

Theorem 6 (Liveness (Core Execution))**.** *Under partial synchrony, in executions where at most f replicas equivocate (i.e., no AbC-induced fork epoch), the protocol ensures that correct replicas eventually commit new proposals; Resilient Path certificates are then produced as part of normal progress.*

Proof. After the Global Stabilization Time (GST), message delays are bounded by Δ. By standard view-synchronization arguments [7], correct replicas will eventually enter a common view v with a correct leader. During this view, the correct leader will broadcast a valid NewView message followed by Propose messages. Because there are at least $n - f - c = q$ correct replicas that remain active, they will all receive the leader's proposals,

validate them, and broadcast VOTE messages within the timeout. The leader (and all clients) will collect these q votes to form a VoteQC, achieving Fast-Path finality. Consequently, every honest replica that observes a VoteQC for slot s broadcasts a CHKPROP, and these proposals all carry identical content: any VoteQC for slot s—in any view—certifies the same digest h (Lemmas 3 and 4), honest replicas execute slots in order from the same checkpointed prefix, and APPLY is deterministic, so every honest CHKPROP for height s equals (s, h, σ_s). Since checkpoint artifacts carry no view tag, the $n - f - c \geq q$ matching proposals aggregate into a CheckpointQC regardless of which view's VoteQC each replica observed first. The $\sigma = \text{APPLY}(h)$ check then passes at every honest replica, so q matching CHKWITNESS messages are collected and a FinalityQC is output. Since we are in the core execution regime (at most f equivocators), no VoteQC forks occur; therefore progress continues indefinitely. If this regime is violated, replicas enter fork recovery, and only Resilient Path safety is claimed. □

D. Resilient Path and fork recovery safety

The Resilient Path provides an extended safety guarantee against an adversary capable of breaking the core f Byzantine limit by leveraging up to f_{abc} alive-but-corrupt replicas.

Lemma 7 (Unique CheckpointQC per Height)**.** *If $f_{abc} < n-3f-2c$, no two conflicting CheckpointQCs can be formed for the same slot height s.*

Proof. By the protocol (Step 2, line 8), an honest replica broadcasts at most one CHKPROP per slot height s. Let C_1 and C_2 be two CheckpointQCs for different state roots at height s. Each requires $q = n - f - c$ distinct signatures. Because honest replicas do not double-sign CHKPROPs at the same height, the intersection between the signers of C_1 and C_2 consists entirely of Byzantine or AbC replicas. The number of intersecting signers is at least $2q - n$. Thus, for two CheckpointQCs to form, the adversary must control at least $2q - n$ nodes. We require $f + f_{abc} < 2q - n$.

We know $q = n - f - c$, so $2q - n = 2(n - f - c) - n = n - 2f - 2c$. Given our assumption that $f_{abc} < n - 3f - 2c$, we have:

$$f + f_{abc} < f + (n - 3f - 2c) = n - 2f - 2c = 2q - n$$

This mathematically prohibits the adversary from successfully forming two CheckpointQCs, proving that at most one can exist per height. □

Lemma 7 rules out conflicts at a single height. To safely adopt the *highest* checkpoint during recovery, we additionally need checkpoints at different heights to agree on their common prefix. Recall (Section III) that state roots bind history: σ_s commits to σ_{s-1} and the applied digest.

Lemma 8 (Checkpoint Chain Consistency)**.** *If $f_{abc} < n - 3f - 2c$, then for any CheckpointQC C' at height s' and CheckpointQC C at height $s > s'$, the history committed by C contains the content (h', σ') certified by C' at height s'. Hence all CheckpointQCs lie on a single chain.*

Proof. The signer sets of C and C' each have size $q = n - f - c$ and intersect in at least $2q - n = n - 2f - 2c$ replicas. Since $f + f_{abc} < n - 2f - 2c$, at least one common signer R is honest. Honest replicas execute slots in order on a single, append-only local history and never re-execute a height they have checkpoint-proposed: in the core regime, Lemma 4 pins any slot with a VoteQC to its digest across views, and in fork epochs R halts normal operation upon detecting the fork and resumes only from the canonical checkpoint branch (Algorithm 2), which by induction on recovery epochs preserves all checkpointed heights. Since R signed C', its history at height s' is exactly (h', σ'); since R signed C, the root σ certified by C was computed on that same history and therefore commits (h', σ') at height s'. By collision resistance of the state commitment, every history consistent with C agrees with C' at height s'. □

Theorem 9 (Resilient Path Safety)**.** *If clients follow the Resilient Path (finalizing only upon seeing a FinalityQC), they will never commit conflicting states for the same slot s, even across view changes and synchronous recoveries, provided $f_{abc} < n - 3f - 2c$.*

Proof. A client finalizes a state only if it observes a FinalityQC, which requires an underlying valid CheckpointQC. By Lemma 7, all CheckpointQCs at height s certify the same content (h, σ). Since honest replicas only broadcast a CHKWITNESS matching this uniquely determined checkpoint content (and only if its payload executes to the correct state), no FinalityQC can be formed for a conflicting state.

During Fork Recovery (Algorithm 2), replicas exchange all known CheckpointQCs via a Byzantine Broadcast protocol (e.g., Dolev-Strong). Because Dolev-Strong does not rely on an honest majority, it guarantees identical outputs for all correct nodes despite a potentially dishonest majority ($f + f_{abc} > n/2$). Alternatively, if $f + f_{abc} < n/2$, an honest-majority broadcast can be deployed to terminate much faster (e.g., in expected constant rounds [16], [14]). In either case, all correct replicas will obtain the exact same set $\mathcal{C}$ of CheckpointQCs. By Lemma 7 at most one CheckpointQC exists per height, and by Lemma 8 the delivered CheckpointQCs lie on a single chain. All correct replicas deterministically adopt the branch of the CheckpointQC with the highest slot C_{max}, which extends the content certified at every lower checkpointed height—in particular, any state finalized by a Resilient Path client prior to the fork. □

V. ORCDAG: A DAG-BASED IMPLEMENTATION

Sections II and IV are independent of how proposals are represented: safety uses first-round votes and the thresholds

q and k of the vote-counting template (Theorem 1). The Orcaella protocol (Section III) uses q for every certificate round. Many high-throughput systems nonetheless realize the same logic over a *directed acyclic graph* of blocks or vertices, with edges for dependencies and leaders extending a frontier of the DAG [17], [18], [19], [20], [21], [22], [23], [24]. We present OrcDAG, a dag-based instantiate of Orcaella.

A DAG vertex (block) plays the role of the primary's proposal for a slot; a replica issues accept/reject once its local validity rules (including dependency checks) hold. The key idea of OrcDAG is that a vote is not a separate signed message but a causal edge: a round $R+1$ block votes for a block B precisely when its causal history includes B (the Vote of Algorithm 3), and blames B when it does not (the Blame of Algorithm 3). The DAG structure itself carries the first-round votes, so no explicit vote messages are needed even though the DAG is uncertified. The Direct Decision Rule of Algorithm 3 realizes the Fast-Path: collecting q agreeing first-round votes is exactly $\geq q$ distinct round $R+1$ authors voting for B, using the same quorum $q = n - f - c$ from Theorem 1; symmetrically, q blames force a Skip.

The Indirect Decision Rule of Algorithm 3 realizes the View-Change. The "later authenticated artifact" of the abstract protocol Orcaella is concretely a committed anchor A at a round $\geq R+2$: because reachability in the DAG certifies the votes enclosed in its causal past, A linking to $\geq k$ distinct round $R+1$ authors that vote for B plays the role of the embedded vote reports in the view change of Section III, with the accept threshold $k = 2f + c + 1$. This single anchor object subsumes the abstract alternatives—such as a commit certificate on a descendant, a checkpoint quorum, or a bundle hashing prior votes—as all collapse to the anchor's causal history, which reachability certifies without any separate certificate. The choice of k is exactly what the Reachability, Exclusivity, and Liveness conditions of Section IV require for safe indirect commits.

Appendix A provides detailed algorithms to formally define OrcDAG.

VI. Evaluation

OrcDAG is a DAG-based instantiation of Orcaella that we implement and benchmark, built in Rust as a fork of Mysticeti [25]. Appendix B details its implementation and testing methodology.

A. Evaluation scope

Our evaluation has a single goal: quantify the latency we gain by trading away fault tolerance, i.e., by moving from the optimal $n = 3f+1$ three-message-delay fault budget to Orcaella's $n = 5f + 3c + 1$ two-message-delay design space. Put differently: *what latency do we gain by giving up the fault tolerance represented by the orange wedge of Figure 1 (Section I)?* This is not trivial. A naive observer might expect a direct 33% latency reduction simply from cutting one of the three commit rounds. But two factors complicate the picture: (1) Orcaella needs a larger quorum of replicas per round than a typical $n = 3f + 1$ protocol; and (2) the saved round only reduces *commit latency* (from block proposal to commit), not *queuing latency* (from transaction submission to block proposal).

Our evaluation makes the following claims:

- **C1**: OrcDAG's lower latency is bought with fault tolerance alone, not throughput.
- **C2**: Under benign crash faults, OrcDAG degrades gracefully, retaining its throughput and low latency.
- **C3**: OrcDAG's latency reduction over the $n = 3f + 1$ baseline is load-dependent, ranging from ∼24% at low load to ∼14% at high load due to its larger quorum and queuing delay.
- **C4**: OrcDAG's commit latency is independent of how the fault budget splits between f and c in a typical geo-distribution, and a split budget deploys at strictly smaller n than the pure-Byzantine $5f + 1$ ($c = 0$) point.
- **C5**: Geo-distribution is decisive: OrcDAG's latency advantage holds when all protocols operate over the same geo-locations, but is negated when the baseline's smaller quorum can exclude a remote region that OrcDAG's larger quorum cannot.

Benchmarking BFT protocols under actual Byzantine behavior is an open problem [26], [27]; the state of the art establishes worst-case guarantees through formal proofs, which we give in Section IV.

B. Experimental setup

To demonstrate these claims, we deploy four OrcDAG configurations spanning the fault-budget spectrum (Figure 2): *Byzantine-only* (f=10, c=0), *Byzantine-heavy* (f=8, c=3), *Balanced* (f=6, c=6), and *Crash-heavy* (f=2, c=13). We use two baselines: *(i)* Mysticeti [19], the closest $n = 3f + 1$ protocol to OrcDAG in terms of both design and implementation, representing the standard 3-round PBFT-style baseline; *(ii)* Hydrangea [3], an alternative hybrid (f, c) design with $n = 3f + 2c + k + 1$ that can commit in a single round-trip (via an optimistic path) and is, to our knowledge, the only other protocol in this design space with a deployable implementation. The pure-Byzantine $c = 0$ point (an $n = 5f+1$ deployment) is itself an OrcDAG configuration, which we include as an additional reference rather than as a separate system.

Several recent protocols explore designs in the same neighborhood (Section VII), including Minimmit [2], Kudzu [4], and Alpenglow [28]. None of these provide deployable implementations with networking code; their codebases are intended for simulation only and are therefore excluded from our WAN measurements. The remaining systems, Mysticeti and Hydrangea, both have mature deployable implementations and serve as our baselines. We configure Hydrangea at f=9, c=10, k=2 for the n=50 committee; in this regime it tolerates more crash faults than OrcDAG, but, being built atop HotStuff, it inherits the data-dissemination bottleneck identified by Narwhal [17].

Algorithm 3: OrcDAG Instantiation (for leader block B at round R)

Definitions:
Vote: A block in round $R+1$ whose causal history includes B.
Blame: A block in round $R+1$ whose causal history does *not* include B.

Direct Decision Rule (evaluated when round $R+1$ blocks are delivered):
if blocks from $\geq q$ distinct authors in round $R+1$ vote for B **then** Commit(B)
if blocks from $\geq q$ distinct authors in round $R+1$ blame B **then** Skip(B)

Indirect Decision Rule (evaluated when anchor A is committed at round $\geq R+2$):
if B is undecided **then**
    if A links to blocks from $\geq k$ distinct authors in round $R+1$ that vote for B **then**
        Commit(B)
    else
        Skip(B)

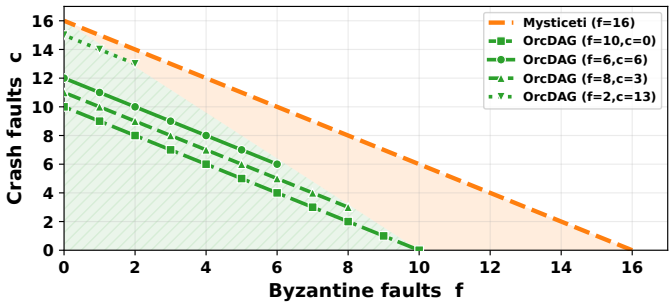

Fig. 2: Runtime fault tolerance of the benchmarked configurations.

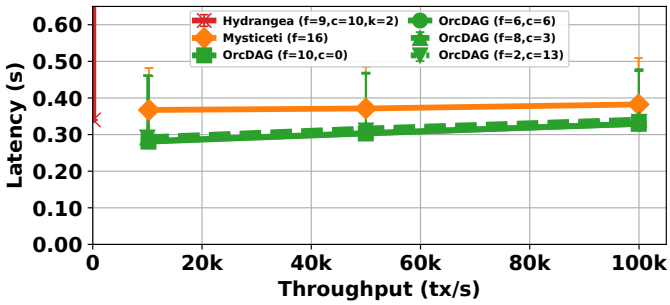

Fig. 3: Throughput-latency, no faults, committees of 50 replicas.

Unless stated otherwise, we emulate a typical blockchain replica distribution [29], [30]: a fast quorum region (EU-US) plus a remote tail (Tokyo). We report median (p50) commit latency with p90 whiskers. Appendix C describes the precise geo-distribution and testbed details used in this section. In all graphs, *latency* refers to the time elapsed from the moment a client submits a transaction to when it is committed by the replicas, and *throughput* refers to the number of (512 bytes) transactions committed per second.

C. Throughput vs. fault tolerance trade-off

Figure 3 evaluates a roughly 50-replica WAN deployment under failure-free conditions. The throughput of OrcDAG matches Mysticeti, which is expected as both build upon the same uncertified-DAG fabric, corroborating related work [17]. For cost reasons, we cap the offered load at 100,000 tx/s, which is two orders of magnitude above the peak throughput of existing blockchains and ample to stress the systems [31]. This confirms claim **C1**: OrcDAG is realizable without sacrificing throughput. Hydrangea performs significantly worse at scale (red line in fig. 3). At 50 replicas, its latency exceeds the plotted range even under minimal load, demonstrating that it does not scale gracefully to large committees. This limitation stems from its HotStuff-style data-dissemination mechanism, consistent with findings reported in related work [17].

The figure also shows that OrcDAG commits at lower latency than Mysticeti; we dissect this latency–fault-tolerance trade-off in Section VI-E.

D. Impact of benign crash faults

Figure 4 shows OrcDAG, Mysticeti, and Hydrangea under benign crash faults in committees of approximately 10 replicas. The DAG-based systems sustain the offered load with a graceful latency inflation relative to the fault-free runs, absorbing crashes by quickly skipping crashed leaders via the direct skip rule (Section V). Each system runs at the minimal committee for this fault budget—Mysticeti at n=7 (f=2), OrcDAG at n=9 (f=1, c=1), and the c=0 reference at n=11 (f=2)—so every committee operates at its maximum fault load. At these minimal committees, OrcDAG matches its own $c = 0$ point because crashed replicas count against c rather than f and so do not consume the Byzantine budget (the n=9 hybrid and the n=11 c=0 reference commit in 379 and 394 ms). This split remains latency-neutral while delivering a latency win over Mysticeti: at low load (10,000 tx/s) OrcDAG commits in 378 ms versus 492 ms (a ∼23% reduction), and near the high-load cap (50,000 tx/s) 395 ms versus 567 ms (a ∼30% reduction). This confirms claim **C2**. Hydrangea's latency exceeds the plotted range, confirming it does not tolerate faults gracefully, corroborating related work [17]. These small committees represent the worst case for the fault budget: a crash removes a large fraction of a 10-replica committee but a negligible fraction of a 50-replica one.

E. Understanding the latency trade-off

We zoom into Figure 3 to understand OrcDAG's latency–fault-tolerance trade-off. At 100,000 tx/s, OrcDAG commits in 330 ms (p50) versus 382 ms for Mysticeti, a ∼14%

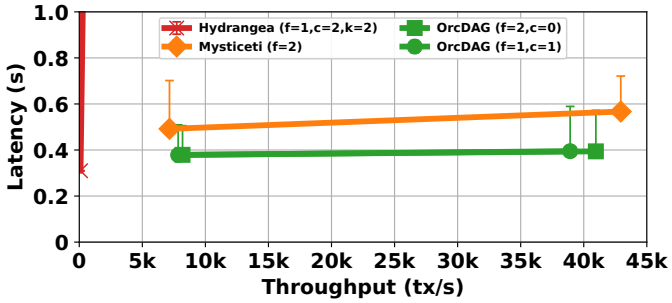

Fig. 4: Throughput-latency under 2 crash faults, minimal committees.

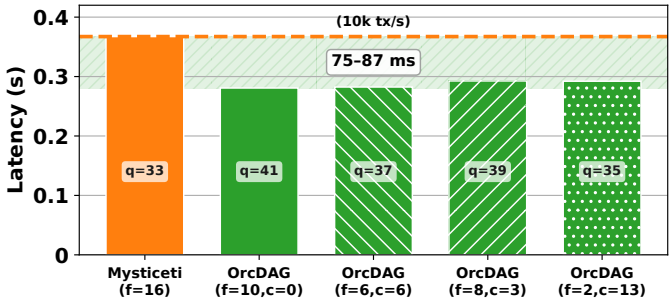

Fig. 5: Per-protocol latency at low load (10k tx/s); queuing ablated.

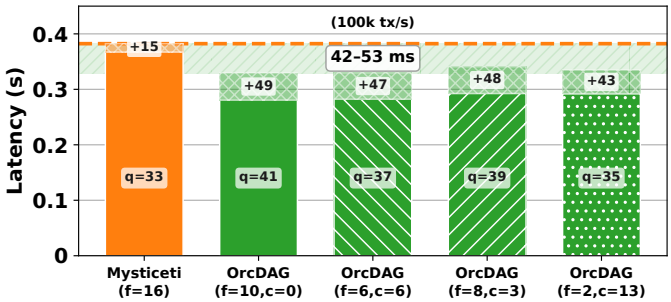

Fig. 6: Per-protocol latency at high load (100k tx/s); queuing segment stacked.

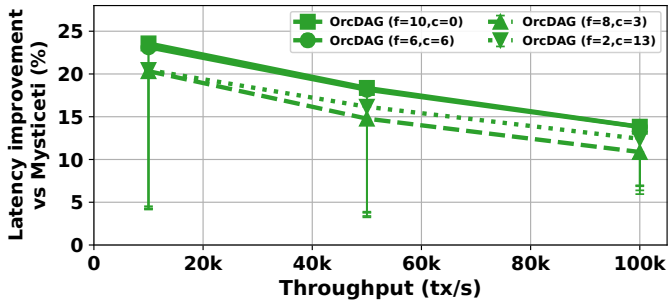

Fig. 7: Latency improvement over Mysticeti vs load, per (f, c) split.

reduction; at low load (10,000 tx/s) the gap widens to 282 ms versus 367 ms, a ∼23% reduction. All OrcDAG configurations, including the c=0 ($5f$+1) reference, fall within ∼12 ms of one another. A naive view expects the move from three to two message delays to give a ∼33% latency reduction. Two coupled effects erode this: (1) OrcDAG uses a larger quorum (∼80% of replicas versus ∼67% for Mysticeti); (2) per-replica queuing under load. The two are coupled: queuing widens the latency distribution, and the larger quorum must wait on a higher percentile of that distribution.

Figure 5 reports per-protocol latency at low load (10k tx/s), which isolates the quorum-width effect (1) because queuing is negligible. Here, a ∼20–25% reduction holds for all OrcDAG configurations. Figure 6 adds high load (100k tx/s), introducing effect (2). Per-replica queuing is the same for all protocols, but OrcDAG's larger quorum amplifies its impact on commit latency (OrcDAG adds +43–49 ms versus +15 ms for Mysticeti). As a result, the reduction erodes by about 10 percentage points, from ∼24% to ∼14% as load grows from 10k to 100k tx/s, and it does so similarly across all (f, c) splits. Figure 7 summarizes this by plotting the total latency reduction over Mysticeti against load, capturing both effects, and confirms claim **C3**: OrcDAG's latency reduction ranges from ∼24% to ∼14% depending on load.

F. The role of quorum location

Figures 5 and 6 show that, across OrcDAG configurations, commit latency is not ordered by quorum size. All configurations commit in two message delays over the same uncertified-DAG structure, so in a fault-free run their finalization latencies are effectively indistinguishable regardless of how the fault budget splits between f and c. This confirms claim **C4**: OrcDAG pays no performance penalty for the hybrid-fault analysis, and a split budget deploys at strictly smaller n than the pure-Byzantine $5f+1$ ($c = 0$) point. Concretely, the configurations cluster at 281–293 ms at low load and 329–341 ms at high load, versus 367 and 382 ms for Mysticeti; the $c = 0$ (Byzantine-only) point, which has the largest quorum (q=41), is among the fastest, so the ∼12 ms spread is run-to-run measurement noise (WAN and egress jitter). Crucially, this neutrality is conditional. The split sets the fast-path quorum size $q = 4f + 2c + 1$: a Byzantine-leaning split carries a larger quorum (q=41 for Byzantine-only) than a crash-leaning one (q=35 for Crash-heavy). The configurations are latency-neutral only because, in this graceful geo-distribution, all of these quorums still form within the fast region.

Figure 8 shows what happens when the quorum is forced out of the fast region. We use crash faults on minimal committees (2 crashes) to force the quorum to include a remote region (Tokyo). When every protocol must reach Tokyo, Mysticeti is penalized more because its extra (third) round waits on the remote replica once more than OrcDAG. But when Mysticeti's larger fault-tolerance budget gives it enough slack to exclude Tokyo while OrcDAG's tighter quorum cannot, the advantage reverses. Concretely, Mysticeti with f=3 ($n = 10$) keeps slack and excludes Tokyo (327 ms), whereas OrcDAG with f=1, c=1 ($n = 9$) and Mysticeti with f=2 ($n = 7$) have no slack and must

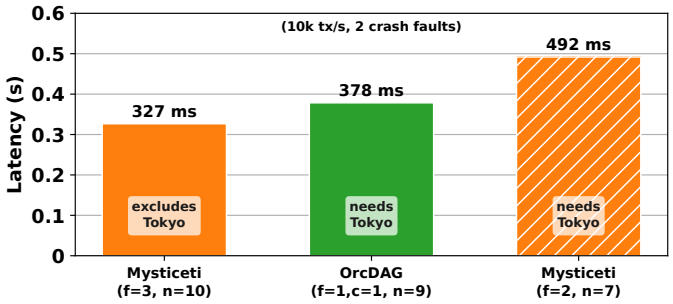

Fig. 8: When the quorum is forced to include the slow region (minimal committees, 2 crashes).

include Tokyo (378 ms and 492 ms). So OrcDAG beats the minimal Mysticeti (two rounds versus three) but loses to the over-provisioned Mysticeti that dodges Tokyo. This confirms claim C5. The same mechanism bounds claim C4: under a less graceful geo-distribution, a Byzantine-leaning configuration's larger quorum (q=41) could be forced to include the slow region while a crash-leaning one (q=35) avoids it, at which point the split would no longer be latency-neutral. Both claims therefore hold only while the quorum stays within the fast region.

VII. Related Work

Hybrid fault models. The formal separation of Byzantine and benign faults was first explored by Thambidurai and Park [32] for interactive consistency. While their work demonstrated that protocols can achieve higher resilience by not treating all failures as worst-case Byzantine, it did not address the latency limitations of state machine replication (SMR). Later pragmatic systems like UpRight [33] and XFT [34] applied this separation to cluster architectures. However, UpRight focuses on end-to-end service robustness rather than theoretical latency minimums, and XFT opportunistically tolerates Byzantine faults only when an honest majority communicates synchronously. In contrast, our work fundamentally re-examines the quorum intersections of SMR under partial synchrony, specifically optimizing for the minimal 2-message-delay commit path while treating f and c as distinct variables.

Low-latency SMR and $5f+1$ protocols. The demand for ultra-low latency in decentralized networks has driven recent protocols—such as Kudzu [4], Minimmit [2], and Alpenglow [28]—to instantiate optimal two-round decision paths. However, these systems rely on a pure counting model that conservatively requires $n \geq 5f+1$, treating any offline replica as a potential equivocator. Hydrangea [3] introduces a hybrid (f, c) analysis requiring $n = 3f + 2c + k + 1$, but it focuses on providing a fast-path under optimistic conditions that falls back to a slower 3-round path when faults increase. At the limit of Hydrangea it can be instantiated with $k = 2f + c - 3$resulting in $n = 5f + 3c - 2$, which allows for a two-round commit path with $p = f + c - 2$. Unlike these approaches, we prove that $n = 5f+3c+1$ is necessary for any protocol that commits in two message delays from a single vote quorum and recovers via a vote-counting view change (Theorem 1), and sufficient via Orcaella—eliminating the need for fallback paths and explicitly separating the cost of crash faults from Byzantine faults. Our choice allows us to also provide a resilient path for clients to tolerate an additional f_{abc} equivocators at the cost of two extra message delays, without ever degrading the core protocol's optimal liveness, but results in lower fast-path liveness gurantees than Hydrangea.

Client-side safety and Alive-but-Corrupt faults. FlexibleBFT [5] introduced the concept of *alive-but-corrupt* (f_{abc}) faults, cleanly separating replica-quorum liveness assumptions from client-visible safety guarantees. While FlexibleBFT primarily uses this slack to separate synchronous and asynchronous network assumptions, we apply it directly to the 2-delay quorum framework. Unlike existing systems that force all clients to accept the same latency-security trade-off, our dual-path architecture explicitly empowers clients. By chaining CheckpointQCs, our Resilient Path allows clients to tolerate an additional f_{abc} equivocators at the cost of two extra message delays, without ever degrading the core protocol's optimal liveness. Orthogonally, a line of low-latency payment systems—FastPay [35], Zef [36], and Stingray [37]—forgoes consensus entirely for single-owner transactions, trading general programmability for latency; Orcaella instead retains full SMR while minimizing the commit path.

VIII. Conclusion

We explored the fundamental limits of achieving optimal 2-message-delay consensus under a hybrid fault model. By cleanly separating Byzantine faults (f) from crash faults (c), we derived tight quorum intersections requiring $n \geq 5f + 3c+1$ with an optimal Fast-Path threshold of $q = n - f - c$ ($4f + 2c + 1$ at the minimal committee size). This allows modern decentralized systems to relax the severe liveness requirements associated with pure $5f+1$ protocols without sacrificing latency.

Building upon these thresholds, we introduced Orcaella, a dual-path consensus protocol that explicitly exposes a latency-safety trade-off to clients. Clients requiring ultra-low latency can finalize in two message delays via the Fast-Path (relying on VoteQCs), while clients prioritizing safety can wait four message delays for the Resilient Path (chaining CheckpointQCs and FinalityQCs). By enforcing a strict single-proposal rule for checkpoints, the Resilient Path guarantees safety against an extended set of f_{abc} alive-but-corrupt replicas. If the core f threshold is ever breached, the system safely halts and employs an Authenticated Byzantine Broadcast recovery mechanism to deterministically salvage the Resilient Path.

Finally, we demonstrated the practical applicability of our results by showing how they seamlessly map onto state-of-the-art DAG-based architectures. Future work

includes extending these hybrid threshold derivations to fully asynchronous consensus environments and developing formal economic models to dynamically adjust f, c, and f_{abc} budgets during live deployments.

Acknowledgements

This work is partially funded by Mysten Labs.

References

[1] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proc. OSDI*, 1999.
[2] B. K. Chou, A. Lewis-Pye, and P. O'Grady, "Minimmit: Fast finality with even faster blocks," arXiv:2508.10862, 2025.
[3] N. Shrestha, A. Kate, and K. Nayak, "Hydrangea: Optimistic two-round partial synchrony with improved fault resilience," Cryptology ePrint Archive, Report 2025/1112, 2025.
[4] V. Shoup, J. Sliwinski, and Y. Vonlanthen, "Kudzu: Fast and simple high-throughput BFT," in *DISC*, 2025.
[5] D. Malkhi, K. Nayak, and L. Ren, "Flexible byzantine fault tolerance," in *Proc. ACM CCS*, 2019.
[6] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, "SoK: Consensus in the age of blockchains," in *ACM AFT*, 2019.
[7] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, 1988.
[8] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, 1990.
[9] S. Cohen, R. Gelashvili, E. Kokoris-Kogias, Z. Li, D. Malkhi, A. Sonnino, and A. Spiegelman, "Be aware of your leaders," in *Financial Cryptography*, 2022.
[10] B. Y. Chan and E. Shi, "Streamlet: Textbook streamlined blockchains," in *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, 2020, pp. 1–11.
[11] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "HotStuff: BFT consensus with linearity and responsiveness," in *PODC*, 2019.
[12] R. Gelashvili, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman, and Z. Xiang, "Jolteon and Ditto: Network-adaptive efficient consensus with asynchronous fallback," in *Financial Cryptography and Data Security (FC)*, 2022.
[13] T. Kichidis, L. Kokoris-Kogias, A. Koshy, I. Sergey, A. Sonnino, M. Tian, and J. Zhang, "Beluga: Block synchronization for BFT consensus protocols," https://arxiv.org/abs/2511.15517, 2025.
[14] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin, "Sync hotstuff: Simple and practical synchronous state machine replication," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 106–118.
[15] D. Dolev and H. R. Strong, "Authenticated algorithms for byzantine agreement," *SIAM Journal on Computing*, vol. 12, no. 4, pp. 656–666, 1983.
[16] J. Katz and C.-Y. Koo, "On expected constant-round protocols for byzantine agreement," in *Advances in Cryptology–CRYPTO 2006*. Springer, 2006, pp. 445–462.
[17] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and Tusk: A DAG-based mempool and efficient BFT consensus," in *EuroSys*, 2022.
[18] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: DAG BFT protocols made practical," in *ACM CCS*, 2022.
[19] K. Babel, A. Chursin, G. Danezis, A. Kichidis, L. Kokoris-Kogias, A. Koshy, A. Sonnino, and M. Tian, "Mysticeti: Reaching the latency limits with uncertified DAGs," in *NDSS*, 2025.
[20] P. Vander Vos, A. Sonnino, G. Tsimos, P. Jovanovic, and L. Kokoris-Kogias, "BlueBottle: Fast and robust blockchains through subsystem specialization," 2025.
[21] P. Jovanovic, L. Kokoris-Kogias, B. Kumara, A. Sonnino, P. Tennage, and I. Zablotchi, "Mahi-Mahi: Low-latency asynchronous BFT DAG-based consensus," 2024.
[22] N. Shrestha, A. Kate, and K. Nayak, "Sailfish: Towards improving the latency of DAG-based BFT," in *IEEE Symposium on Security and Privacy (S&P)*, 2025.
[23] A. Spiegelman, B. Arun, R. Gelashvili, and Z. Li, "Shoal: Improving DAG-BFT latency and robustness," in *Financial Cryptography and Data Security (FC)*, 2024.
[24] G. Tsimos, A. Kichidis, A. Sonnino, and L. Kokoris-Kogias, "HammerHead: Leader reputation for dynamic scheduling," in *IEEE ICDCS*, 2024.
[25] M. Labs, "Mysticeti: Low-latency DAG consensus with fast commit path," https://github.com/asonnino/mysticeti, 2024.
[26] S. Bano, A. Sonnino, A. Chursin, D. Perelman, Z. Li, A. Ching, and D. Malkhi, "Twins: BFT systems made robust," in *Proc. International Conference on Principles of Distributed Systems (OPODIS)*, 2021.
[27] G. Giuliari, A. Sonnino, M. Frei, F. Streun, L. Kokoris-Kogias, and A. Perrig, "An empirical study of consensus protocols' DoS resilience," in *ACM AsiaCCS*, 2024.
[28] Q. Kniep, K. Sliwinski, and R. Wattenhofer, "Solana Alpenglow consensus: Increased bandwidth, reduced latency," Anza white paper, v1.1, 2025.
[29] Helius, "Solana decentralization: Facts and figures," https://www.helius.dev/blog/solana-decentralization-facts-and-figures, 2024, accessed June 2026.
[30] Suiscan, "Suiscan: Sui mainnet explorer," https://suiscan.xyz/mainnet/home, accessed June 2026.
[31] S. Blackshear, A. Chursin, G. Danezis, A. Kichidis, L. Kokoris-Kogias, X. Li, M. Logan, A. Menon, T. Nowacki, A. Sonnino, B. Williams, and L. Zhang, "Sui Lutris: A blockchain combining broadcast and consensus," in *ACM CCS*, 2024.
[32] P. Thambidurai and Y.-K. Park, "Interactive consistency with multiple failure modes," in *Proceedings of the 7th Symposium on Reliable Distributed Systems*. IEEE, 1988, pp. 93–100.
[33] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "Upright cluster services," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 277–290.
[34] S. Liu, P. Viotti, C. Cachin, V. Quema, and M. Vukolic, "XFT: Practical fault tolerance beyond crashes," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 485–500.
[35] M. Baudet, G. Danezis, and A. Sonnino, "FastPay: High-performance byzantine fault tolerant settlement," in *ACM AFT*, 2020.
[36] M. Baudet, A. Sonnino, M. Kelkar, and G. Danezis, "Zef: Low-latency, scalable, private payments," in *WPES@CCS*, 2023.
[37] S. Sridhar, A. Sonnino, and L. Kokoris-Kogias, "Stingray: Fast concurrent transactions without consensus," https://arxiv.org/abs/2501.06531, 2025.
[38] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All you need is DAG," in *PODC*, 2021.
[39] T. T. Team, "Tokio," https://tokio.rs, 2024.
[40] H. de Valence, "Ed25519 for consensus-critical contexts," https://crates.io/crates/ed25519-consensus, 2024.
[41] Z. Li, A. Sonnino, and P. Jovanovic, "Performance of EdDSA and BLS signatures in committee-based consensus," in *APPLIED@PODC*, 2023.
[42] RustCrypto, "Rustcrypto: Hashes," https://github.com/RustCrypto/hashes, 2024.
[43] Die.Net, "writev(3) – linux man page," https://linux.die.net/man/3/writev, 2024.
[44] Meta, "Sapling (Minibytes)," https://github.com/facebook/sapling/tree/main/eden/scm/lib/minibytes, 2024.
[45] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

Appendix A
Detailed Algorithms for the DAG-Based Variant

This appendix complements Section V by formally defining the commit logic of OrcDAG, the DAG-based variant of Orcaella, through detailed algorithms.

Algorithm 4: Decision Rules

```
1: leadersPerRound                                   ▷ A number between 1 and q
2: waveLength                                        ▷ Set to 2 for Orcaella

3: procedure TryDecide(r_committed, r_highest)
4:     S ← [ ]                                       ▷ Holds decisions
5:     for r ← r_highest down to r_committed + 1 do
6:         for l ← leadersPerRound − 1 down to 0 do
7:             i ← r mod waveLength
8:             D ← Decider(i, l)
9:             w ← D.WaveNumber(r)
10:            s ← D.TryDirectDecide(w)
11:            if s = ⊥ then s ← D.TryIndirectDecide(w, S)
12:            S ← s ∥ S
13:    return S

14: procedure ExtendCommitSeq(r_committed, r_highest)
15:    S ← TryDecide(r_committed, r_highest)
16:    S_commit ← [ ]                                ▷ Holds committed blocks
17:    for s ∈ S do
18:        if s = ⊥ then break
19:        if s = Commit(b_leader) then S_commit ← S_commit ∥ b_leader
20:    return LinearizeSubDags(S_commit) ▷ Same as DAG-Rider [38]
```

Algorithm 5: Decider Instance and Helpers

```
1: waveOffset = i                          ▷ The first parameter of the Decider (i)
2: leaderOffset = l                        ▷ The second parameter of the Decider (l)
3: waveLength                              ▷ Set to 2 for Orcaella
4: replicas                                ▷ The set of replicas

5: procedure WaveNumber(r)
6:     return (r − waveOffset)/waveLength

7: procedure ProposeRound(w)
8:     return (w ∗ waveLength) + waveOffset

9: procedure DecisionRound(w)
10:    return ProposeRound(w)+(waveLength − 1)

11: procedure StronglyCertifiedLeader(w, b_leader)
12:    B_decision ← GetDecisionBlocks(w)
13:    return |{b'.author : b' ∈ B_decision ∧ Link(b_leader, b')}| ≥ q   ▷
       q = n − f − c; count authors, as replicas may equivocate

14: procedure SkippedLeader(w, b_leader)
15:    B_decision ← GetDecisionBlocks(w)
16:    return |{b'.author : b' ∈ B_decision ∧ ¬Link(b_leader, b')}| ≥ q   ▷
       q = n − f − c

17: procedure TryDirectDecide(w)
18:    B_leader ← GetLeaderBlocks(w, leaderOffset)
19:    for b_leader ∈ B_leader do
20:        if SkippedLeader(w, b_leader) then return Skip
21:        if StronglyCertifiedLeader(w, b_leader) then return
           Commit(b_leader)
22:    return ⊥

23: procedure WeaklyCertifiedLeader(b_anchor, b_leader)
24:    w ← WaveNumber(b_leader.round)
25:    B_decision ← GetDecisionBlocks(w)
26:    return |{b.author : b ∈ B_decision ∧ Link(b_leader, b) ∧
       Link(b, b_anchor)}| ≥ k                   ▷ k = 2f + c + 1

27: procedure TryIndirectDecide(w, S)
28:    r_decision ←DecisionRound(w)
29:    s_anchor ← first s ∈ S s.t. r_decision < s.round ∧ s ≠ Skip
30:    if s_anchor = Commit(b_anchor) then
31:        B_leader ← GetLeaderBlocks(w, leaderOffset)
32:        if ∃ b_leader ∈ B_leader s.t. WeaklyCertifiedLeader(b_anchor, b_leader)
           then return Commit(b_leader)
33:        else return Skip
34:    return ⊥

35: procedure GetDecisionBlocks(w)
36:    r_decision ←DecisionRound(w)
37:    return DAG[r_decision]

38: procedure GetLeaderBlocks(w, rank) ▷ Replicas may equivocate
39:    r_propose ← ProposeRound(w)
40:    s ← r_propose
41:    leader ← replicas[(s + rank) mod |replicas|]
42:    return {b ∈ DAG[r_propose] : b.author = leader}

43: procedure Link(b_old, b_new)
44:    return ∃ a sequence of m ∈ ℕ blocks b_1, ..., b_m s.t.
       b_1 = b_old ∧ b_m = b_new ∧ ∀j ∈ [2, m] : b_j ∈ ⋃_{r≥1} DAG[r] ∧
       b_{j−1} ∈ b_j.parents
```

DAG-building layer. We assume the underlying DAG-building logic of Mysticeti [19]: replicas proceed in logical rounds; in each round every honest replica proposes one block referencing $\geq q$ distinct valid blocks from the previous round; blocks are disseminated to others; only blocks whose entire causal history has been validated are stored locally. The decision logic specified here operates on this local DAG and is independent of how blocks reach the replica.

Entry point and idempotency. Algorithm 4 is the commit-logic entry point. Inline with related work [19], [21], it is idempotent and stateless from the DAG engine's perspective: the engine may invoke it whenever it likes, typically upon receiving and integrating a new block, passing the highest round currently in the local DAG ($r_{highest}$) and the round of the last block already committed ($r_{committed}$). The procedure returns the extension to the commit sequence (possibly empty) that the engine should append to its committed prefix. The entry point is ExtendCommitSeq($r_{committed}, r_{highest}$), which internally calls TryDecide to evaluate each undecided leader slot using the rules in Algorithm 5, then linearises the causal sub-DAG of every newly committed leader (as introduced by DAG-Rider [38]). Algorithm 5 specifies the per-slot decision process and with the supporting helper procedures (GetDecisionBlocks, GetLeaderBlocks, Link).

Appendix B
Implementation

We implement a networked, multi-core OrcDAG replica in Rust by forking the Mysticeti codebase [25], [31]. Our implementation leverages `tokio` [39] for asynchronous networking, utilizing raw TCP sockets for communication without relying on any RPC frameworks. For cryptographic operations, we rely on `ed25519-consensus` [40], [41] for asymmetric cryptography and `blake2` [42] for cryptographic hashing. To ensure data persistence and crash recovery, we employ a Write-Ahead Log (WAL). The WAL optimizes I/O operations through vectored writes [43] and efficient memory-mapped file usage with the `minibytes` [44] crate, minimizing data copying and serialization. In addition to regular unit tests, we inherit and use two supplementary testing utilities from the

Mysticeti codebase. First, a simulation layer replicates the functionality of the `tokio` runtime and TCP networking; the simulated network reproduces realistic WAN latencies, while the `tokio` runtime simulator employs a discrete-event simulation approach to model the passage of time. Second, a command-line utility (called the *orchestrator*) [17], [18] deploys real-world clusters of OrcDAG replicas on machines distributed across the globe. We open-source our OrcDAG implementation, along with its simulator and orchestration tools, to ensure reproducibility of our results[1].

Appendix C
Testbed Details

This appendix complements Section VI by detailing the testbed and experimental setup used to evaluate OrcDAG.

We deploy all protocols on AWS, using `m5d.8xlarge` instances across 6 different AWS regions: Northern Virginia (us-east-1), Ohio (us-east-2), Frankfurt (eu-central-1), London (eu-west-2), Paris (eu-west-3), and Tokyo (ap-northeast-1). Replicas are distributed across those regions as equally as possible. Each machine provides 10 Gbps of bandwidth, 32 virtual CPUs (16 physical cores) on a 3.1 GHz Intel Xeon Skylake 8175M, 128 GB memory, and runs Linux Ubuntu server 24.04.

We instantiate several geo-distributed benchmark clients within each replica submitting transactions in an open-loop model at a fixed rate. We experimentally increase the load of transactions sent to the systems, and record the throughput and latency of commits. As a result, all plots in Section VI illustrate the steady-state latency of all systems under low load, as well as the maximal throughput they can provide after which latency grows quickly. Transactions in the benchmarks are arbitrary and contain 512 bytes. We configure Orcaella and Mysticeti with 2 leaders per round, and all protocols use a leader timeout of 1 second.

Appendix D
Crash-Only Deployment

When configured with $f = 0$ to tolerate only crashes and no Byzantine faults, Orcaella reduces to an optimal crash-fault-tolerant (CFT) protocol [45]. Figure 9 reports the performance of OrcDAG configured with $f = 0$ and $c = 1$ in a minimal $n = 3$ deployment, where the three replicas run in three distinct, nearby European regions—Frankfurt (eu-central-1), Ireland (eu-west-1), and London (eu-west-2)—to model a regional CFT deployment. The plot shows two loads, 1k and 10k tx/s. Each box spans the mean ± 1 standard deviation, and the whiskers denote ± 2 standard deviations. In this lightly-loaded regime the commit latency is load-independent and essentially flat: OrcDAG commits with a mean latency of about 22 ms at 1k tx/s and about 23 ms at 10k tx/s, with a standard deviation of about 7 ms. This is roughly two orders of magnitude below the hundreds of milliseconds observed in the geo-distributed WAN runs of the main evaluation (Section VI), highlighting the best-case latency of a regional crash-tolerant deployment.

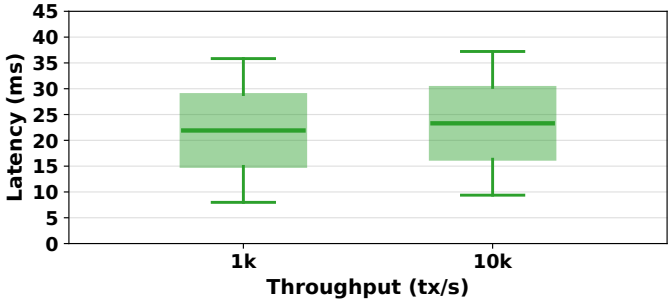

Fig. 9: EU (multi-region) crash-only OrcDAG run; mean ± stdev.

[1] https://github.com/asonnino/mysticeti (commit `96dee8d`)