nalysis of the XRP Ledger Consensus Protocol

Brad Chase Etha MacBrough

Ripple Research
{bchase,emacbr ugh}@ripple.c m

February 21, 2018

bstract

The XRP Ledger Co se sus Protocol is a previously developed co -se sus protocol poweri g the XRP Ledger. It is a low-late cy Byza ti e agreeme t protocol, capable of reachi g co se sus without full agreeme t o which odes are members of the etwork. We prese t a detailed expla-atio of the algorithm a d derive co ditio s for its safety a d live ess.

1 Introduction

The XRP Ledger is a distributed payment system enabling users to transfer value seamlessly around the world. Operating within a distributed peer-to-peer network, the XRP Ledger faces the same challenges as other digital currencies in preventing double-spending of funds and ensuring network-wide consensus on the state of user accounts and balances. First proposed and then implemented by Schwartz et al. [10], the algorithm underlying XRP solves these problems using a Byzantine fault tolerant agreement protocol over collectively trusted subnetworks, hereby referred to as the XRP Ledger Consensus Protocol, or XRP LCP for short.

bstractly, the XRP Ledger network is a replicated state machine [9]. The replicated state is the ledger maintained by each node in the network and state transitions correspond to transactions submitted by clients of the network. Once nodes agree on sets of transactions to apply to the state, a transaction processing protocol specifies deterministic rules for ordering transactions within each set and how to apply transactions to generate the new ledger state. Thus, the role of XRP LCP is only to make the network reach agreement on sets of transactions, not on the content or outcome of those transactions. s long as nodes agree on a transaction set, the transaction processing protocol guarantees that every node generates a consistent ledger. s a Byzantine fault tolerant protocol, XRP LCP must operate even in the presence of faulty or malicious participants.

Byzantine fault tolerant consensus protocols have a rich history, but most require preexisting agreement on the protocol participants [8, 3]. The distinguishing characteristic of XRP LCP is that it guarantees consistency with only partial agreement on who participates, allowing a decentralized open network. Compared to other decentralized consensus algorithms like proof-of-work [7] or proof-of-stake [2], XRP LCP has since its inception provided lower transaction latency and higher throughput for its users. However, without uniform agreement on the network participants, users still need a way to determine whether their choice of network peers will lead to a consistent network state. In this setting, each user individually defines a **unique node list** or **UNL**, which is the set of nodes whose messages it will listen to when making decisions about the network state. It is the intersection of any pair of correct nodes' UNLs that determines network safety. s described in the original whitepaper [10], the minimum overlap requirement was originally believed to be roughly 20% of the UNL. n independent analysis later suggested the correct bound was instead roughly $> 40\%$ [1].

Given this confusion, our goal in this work is to give a clear and detailed explanation of XRP LCP and derive the necessary conditions on UNL overlap for consistency and liveness. We will not discuss the transactional semantics of XRP's ledger or XRP's benefits as a digital currency, but instead view the algorithm as a general consensus protocol. We re-evaluate the two prior overlap results and provide a single corrected bound which is partway between the bounds of [10] and [1]. We also show that under a more general fault model which was not considered in the original whitepaper but is canonically used in the research literature, the minimum overlap is actually roughly $> 90\%$ of the UNL. Finally, we show that during the present stages of diversifying trusted network operators [12], the XRP network is both safe and cannot become "stuck" making no forward progress.

This research provides a definitive result about the safety of XRP Ledger in its current state. However, to encourage greater flexibility in choosing UNLs in the future, we would prefer an algorithm that gets closer to the original expected overlap bounds. In a sibling paper [6], we present a novel alternative consensus algorithm called **Cobalt** that lowers the overlap bound to only $> 60\%$ in the general fault model, cannot get stuck in *any* network that satisfies the overlap bound, and has several other properties that make it suitable for eventually replacing XRP LCP. This paper thus serves primarily to show that the XRP Ledger is safe in the interim while transitioning to Cobalt, and the relatively strict requirements on UNL configurations under XRP LCP should be viewed in light of this planned transition.

Section 2 defines the network model and defines the consensus problem. Section 3 is a detailed description of the Ripple consensus algorithm. In section 4, we prove the network conditions needed to guarantee correctness of the algorithm. Finally, section 5 concludes with a discussion of the results and directions for improvement.

2 Network Model and Problem Definition

Let $_i$ be a node in the network with unique identifier i, such as a cryptographic public key. Each node $_i$ is free to choose a **unique node list** UNL_i, which is the set of nodes (possibly including itself) whose messages $_i$ will listen to as part of the XRP LCP. The UNL represents a subset of the network which, when taken collectively, is trusted by $_i$ to not collude in an attempt to defraud the network (see [10] for the motivation of this name). The UNLs give structure to the network, with a node that is present in more UNLs implicitly having more influence. n individual node has complete discretion in the choice of their UNL, although we show in section 4 that minimum overlap with other UNLs is necessary for consistency and liveness with other honest nodes. We do not assume that trust is symmetric, so for instance there may be a node $_j \in \mathsf{UNL}_i$ such that $_i \notin \mathsf{UNL}_j$. Figure 1 shows an example trust network.

node that is not crashed and behaves exactly according to the XRP LCP specification is said to be **honest** or **correct**; we use the two terms interchangeably. ny node that does not behave according to protocol is said to be **Byzantine**. Byzantine behavior can include not responding to messages, sending incorrect messages, and even sending different messages to different parties. In section 4, we initially consider a restriction on the adversary called **Byzantine accountability**, which states that all nodes – even Byzantine ones – cannot send different messages to different nodes. This was part of the original whitepaper [10], which assumed such behavior in a peer-to-peer network could be identified and corrected by honest nodes. However, due to asynchrony and the possibility of honest nodes being temporarily partitioned, this assumption is in practice tenuous at best. Thus the bulk of our results do not depend on this assumption and we will clearly state when it is assumed.

For any node $_i$, we denote $_i = |\mathsf{UNL}_i|$ and define the **quorum**, denoted q_i, a parameter which roughly specifies the minimum number of agreeing nodes in UNL_i that $_i$ needs to hear from to commit to a decision. Each node sets q_i to be 80% of its UNL size, or, more exactly, $q_i = \lceil 0.8 \; _i \rceil$. We assume at most $t_i \quad _i - q_i$ nodes in UNL_i may be Byzantine faulty.

Let L represent a ledger, which is the shared state of the system and includes account settings, balances, order books, etc. Two ledgers L, L' are the same if they represent the same ordered history of transactions starting from the unique genesis ledger. Each ledger also has sequence number $\mathsf{seq}(L)$ that is one greater than its parent ledger's sequence number. The genesis ledger has $\mathsf{seq}(L) = 1$. ledger L is created by applying a sequence of transactions $T = [x_0, x_1, \ldots]$ to its parent $\mathsf{pare\;t}(L)$ according to the protocol rules. Two ledgers may have the same parent ledger and sequence number, but differ because they applied different transactions. Note though that the protocol specifies that every set of transactions has a deterministic ordering, so it is not possible for two correct nodes to apply the same transactions but in a different order.

The nodes communicate over a peer-to-peer network which has no prescribed relation with the UNL structure. We simply assume that for every node $_i$ and every node $_j \in \mathsf{UNL}_i$, there is a reliable authenticated channel for $_i$ to receive

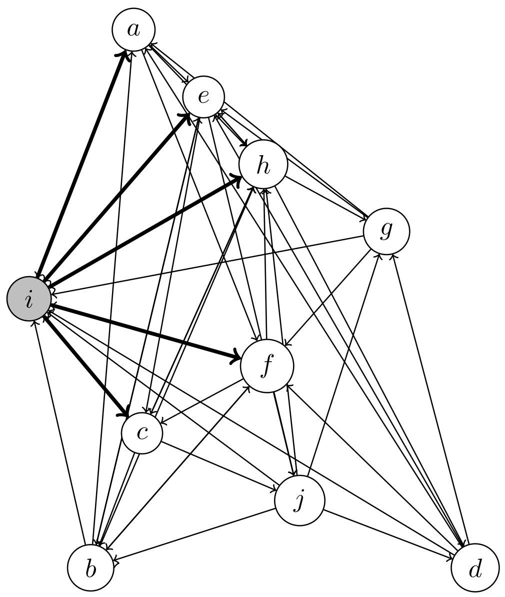

Figure 1: Example trust graph. The highlighted edges represent the UNL of node $_i$.

messages from $_j$. To implement such an authenticated channel, all messages are cryptographically signed and verified by receivers. Each node uses a single communication primitive **broadcast**, which when called from node $_j$ sends the same message to all nodes $_i$ for which $_j \in \mathsf{UNL}_i$. In the algorithms presented in appendix , we use a corresponding **receive** primitive which is called asynchronously upon the receipt of a broadcast message.

The outcome of XRP LCP is for a node to **fully validate** ledgers. fully validated ledger is irrevocable and authoritative, and reflects transactions submitted by network clients and accepted by the consensus algorithm. Fully validating a ledger also fully validates all of its ancestors. In this context, a **fork** is a situation in which two honest nodes fully validate contradictory ledgers, i.e., different ledgers with the same sequence number. The network is said to be **fork-safe** if it can never fork with a tolerated configuration of Byzantine nodes.

lthough XRP LCP is typically defined in terms of the fully validated chain of ledgers, since each ledger in the chain represents a deterministically-ordered sequence of transactions it also can be considered as an **atomic broadcast** protocol with batching of transactions for efficiency. Formally, an atomic broadcast protocol is an algorithm in which a set of **clients**, arbitrarily many of which may be Byzantine faulty, can broadcast **transactions**, and each node can **accept** some of those transactions according to the following properties:

1. BC- greement: If a correct node accepts a transaction into a ledger, then eventually all correct nodes accept the transaction into a ledger.

2. BC-Linearizability: If a correct node accepts transaction x before transaction x', then all correct nodes accept transaction x before x'.
3. BC-Censorship-Resilience: If a correct client broadcasts a valid transaction x to all correct nodes, x will eventually be accepted by all nodes.

n atomic broadcast algorithm is in particular a variant of consensus [4]. Note that Censorship Resilience is the formalized definition of forward progress. In practice, the peer-to-peer network weakens the requirement that clients submit their transaction to all correct nodes, since correct nodes will echo a submitted transaction to each other, flooding the network until every node receives it.

In order to evaluate the correctness of XRP LCP, we model the peer-to-peer network as if it were controlled by a **network adversary** that can behave arbitrarily. The adversary is controls the delivery order of all messages, as well as at most t_i nodes in UNL_i for any correct node $_i$. We assume though that the adversary is computationally bounded; in particular, it is unable to break generally accepted cryptographic protocols. The identities of Byzantine nodes are unknown in advance by honest nodes in the network.

Since atomic broadcast is a variant of consensus, by the FLP result [5] we cannot guarantee forward progress in the presence of arbitrary asynchrony and faulty nodes. Instead, we assume a form of "weak asynchrony": safety should hold under arbitrary asynchrony, but censorship resilience is only guaranteed to hold under the assumption that the network is eventually **civil**, meaning that messages are delivered within some protocol-specified maximum delay bound and no nodes are faulty.

In order to help enforce the bounded delay in the XRP ledger implementation, several heuristics are used to identify lagging nodes, prevent excessive flooding of messages and route traffic through the network. Protocol parameters also define maximum delays on different trusted messages in an attempt to aid liveness. lthough we ignore these details to simplify the presentation below, we stress that they are important practical considerations in the actual implementation and control the real world performance of the algorithm. The fact that XRP LCP is only weakly asynchronous and its performance depends on these parameters is a limitation of the algorithm. Cobalt, the proposed alternative algorithm to XRP LCP, does not have these limitations. It uses cryptographic randomness to evade the FLP result and guarantees forward progress even with the maximal number of tolerated faulty nodes and unbounded asynchrony [6].

Table 1 summarizes our notation, including some that will be explained in subsequent sections.

3 The XRP Ledger Consensus Protocol

The XRP Ledger Consensus Protocol consists of three primary components:

- **Deliberation**, in which nodes iteratively propose a transaction set to apply to a prior ledger, based on proposals received from other trusted

$_i$	node in the network
UNL_i	The unique node list (UNL) selected by $_i$
$_i, q_i, t_i$	The size, validation quorum and maximum Byzantine faults of UNL_i
L	ledger
$\mathsf{seq}(L)$	Sequence number of ledger L
$\hat{L}$	Fully validated ledger with largest sequence number (the fully validated tip ledger)
$\tilde{L}$	Current working ledger of deliberation
$T = \{x_0, x_1, \ldots\}$	set of transactions
$P_{T,r,L,i}$	Node $_i$'s r-th deliberation proposal of transactions T to apply to L
$V_{L,i}$	Node $_i$'s validation of ledger L
$\mathsf{supp}_{tip}(L), \mathsf{supp}_{branch}(L)$	Tip and branch support of a ledger L
$\mathsf{u\ committed}(s)$	Uncommitted support at sequence number s
$\phi(L, L')$	Ordering function that is 1 if $L > L'$ (by hash) and 0 otherwise

Table 1: Summary of notation

nodes. When a node believes enough proposals agree, it applies the corresponding transactions to the prior ledger according to the ledger protocol rules. It then issues a validation for the generated ledger.

- **Validation**, in which nodes decide whether to fully validate a ledger, based on the validations issued by trusted nodes. Once a quorum of validations for the same ledger is reached, that ledger and its ancestors are deemed fully validated and its state is authoritative and irrevocable.

- **Preferred Branch**, in which nodes determine the preferred working branch of ledger history. In times of asynchrony, network difficulty, or Byzantine failure, nodes may not initially validate the same ledger for a given sequence number. In order to make forward progress and fully validate later ledgers, nodes use the ledger ancestry of trusted validations to resume deliberating on the network's preferred ledger.

In short, for each sequence number s, each peer $_i$ issues a validation $V_{L,i}$ for the ledger L with $s = \mathsf{seq}(L)$ that it expects will be fully validated by validation. Under civil executions, the deliberation process makes it highly likely the validated ledger will match the ledger validated by its trusted peers. In cases when the network is not working normally, preferred branch ensures peers select a common branch such that nodes will later fully validate the same ledger L' with $\mathsf{seq}(L') > s$. This two-step sequence of deliberation and validation is similar to the proof of stake finality gadget recently introduced by Buterin and Griffith [2]. Indeed, the preferred branch protocol shares a common principle with the GHOST rule of Sompolinsky and Zohar [11].

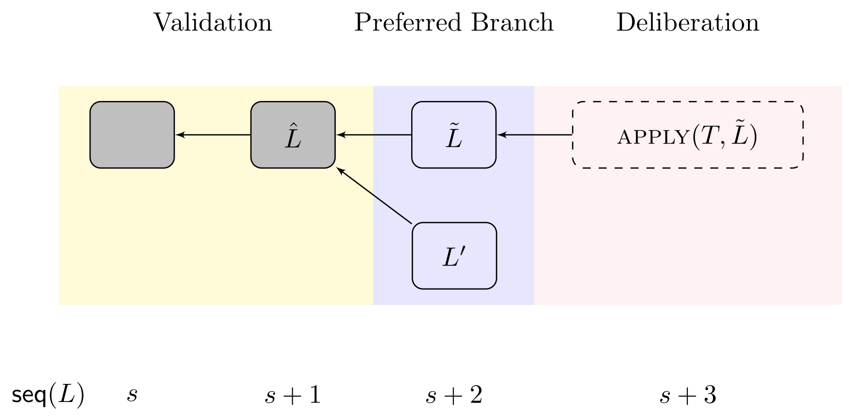

Figure 2: Components of XRP LCP. Each rounded rectangle is a ledger with an arrow pointing to its parent.

Figure 2 is a schematic view of ledger history and shows how these components interact to advance the ledger state from the perspective of a single node. Time flows to the right, with the two grey ledgers on the left defining the fully validated and authoritative ledger chain with tip ledger $\hat{L}$. The dashed ledger on the right represents the deliberation frontier, in which the node is currently negotiating with its trusted nodes on which transactions to apply towards the next ledger. The unfilled ledgers represent two conflicting ledgers $\tilde{L}, L'$ that have been validated by different nodes, but which have not received a quorum to fully validate. In this schematic, preferred branch determined the upper ledger $\tilde{L}$ is most likely to be fully validated, so that is the working parent ledger for this node's active deliberation round.

3.1 Deliberation

Deliberation is the component of Ripple consensus in which nodes attempt to agree on the set of transactions to apply towards ledgers they validate. Clients submit transactions to one or more nodes in the network, who in turn broadcast the transaction to the rest of the network. Each node maintains a set of these pending transactions that have not been included in a ledger. Starting from this set, a node iteratively proposes new transaction sets based on the support of individual transactions among the sets proposed by nodes in its UNL. Each proposal $P_{T,r,L,i}$ is a tuple of

- T, the proposing node's current guess of the consensus transaction set.
- r, the round number of this proposal relative to the other proposals from $_i$ based on prior ledger L.
- L, the prior ledger these transactions will apply to.

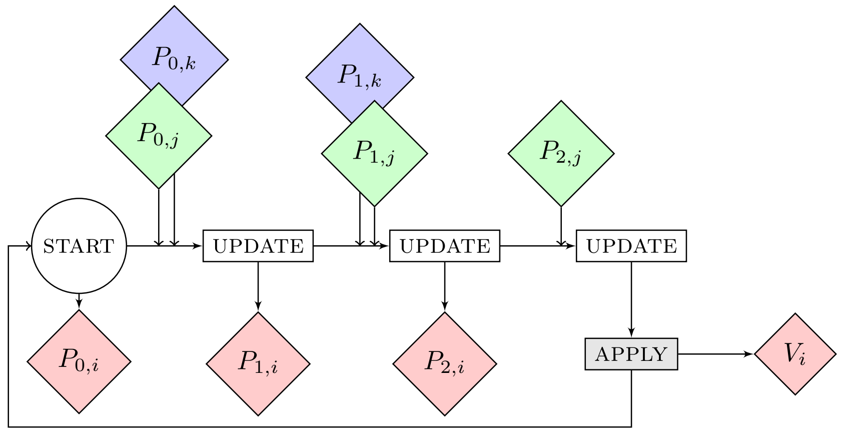

Figure 3: Overview of deliberation from the perspective of $_i$. The diamonds are proposals or validations, with all but the deliberation round and node identifier subscripts removed for brevity.

- i, the identifier of the node $_i$ that broadcasts this proposal.

When enough nodes in its UNL propose the same transaction set, a node issues a validation based on that set and begins the next round of deliberation.

Figure 3 is a high-level overview of deliberation, presented formally in algorithm 1. node initially calls START to begin a new deliberation round and propose its initial view of the consensus transactions. It asynchronously processes new proposals from trusted nodes, maintaining the set of most recent proposals from each. Proposals are only considered if they are for the same prior ledger $\tilde{L}$. The node regularly UPDATEs its proposed consensus transaction set in response to newly received node proposals, only including transactions present in at least $threshold(r)$ of the most recently received proposals from its trusted nodes. The threshold starts as a simple majority of the nodes in the UNL, but ratchets up as deliberation rounds proceed. This ensures slow nodes can't prevent consensus converging. In the XRP Ledger implementation, the threshold goes $0.5 \to 0.65 \to 0.70 \to 0.95$ as r increases. Each node $_i$ declares consensus reached when it sees the quorum q_i of its trusted nodes agree on the transaction set. It then applies the consensus transactions to generate the next ledger L, broadcasts its validation $V_{L,i}$ and begins a new round of deliberation.

It is important to note that a node may only validate one ledger with a given sequence number. In fact, the invariant is for node $_i$ to only issue a validation $V_{L,i}$ for a ledger L if $\mathsf{seq}(L)$ is greater than that of any ledger previously validated by $_i$. Thus if during deliberation, a node determines it is not working on the preferred branch, it will switch to work on the preferred ledger but will not issue a validation until it has caught back up to the sequence number it was on before switching.

In the XRP Ledger implementation of deliberation, protocol timing parameters determine the synchronization requirements of node proposals and conditions for ending deliberation. dditional waiting periods between phases of deliberation balance the throughput and latency of transaction processing as well as the network overhead of broadcasting proposals and transaction sets. There are also protocol rules that determine which transactions are in the initial proposal and how transactions that failed to be included are retried in subsequent deliberations. lthough this deviates from the abstract algorithm presented in this paper, we believe the changes only obscure the presentation of the algorithm and can be viewed as an optimization for increasing transaction throughput. Most importantly, the safety and liveness results in section 4 do not depend on these details of deliberation.

3.2 Validation

Validation is the simplest of the three components and is summarized in algorithm 2. Nodes in the network simply listen for validations from trusted nodes. If a node $_i$ sees a quorum q_i of validations for a ledger L, then it sets the new fully validated tip ledger $\hat{L}$ to L.

3.3 Preferred Branch

Validators normally validate a simple chain of ledgers, e.g. $L \quad \to L^B \to L^C \ldots$. However, during times of asynchrony, network difficulty, or temporary Byzantine failure during deliberation, not all correct nodes may end up receiving enough validations for any individual ledger to fully validate. When presented with conflicting ledgers, preferred branch is the strategy which determines the preferred chain of ledgers to switch to in order to continue making forward progress. It is based on the shared ancestry of the most recent validated ledgers, $lastVals$, and the following quantities:

1. The **tip support** of a ledger L, which is the number of trusted nodes whose most recent validated ledger is L,
$$\mathsf{supp}_{tip}(L) = |\{V_{L\ ,i} \in lastVals : L = L'\}|. \tag{1}$$
2. The **branch support** of a ledger L, which is the number of trusted nodes whose most recently validated ledger is either L or is descended from L,
$$\mathsf{supp}_{branch}(L) = \mathsf{supp}_{tip}(L) + |\{V_{L\ ,i} \in lastVals : L \in a\ \ cestors(L')\}|, \tag{2}$$
where $a\ \ cestors(L')$ is the set of ancestors of L', i.e. the parent, grandparent, great-grandparent, etc., all the way back to the genesis ledger.
3. The **uncommitted support** on a sequence number s, which is the number of trusted nodes whose most recent validated ledger is for a ledger with either sequence lower than s or with sequence lower than that of the largest ledger L validation that we personally have broadcasted:
$$\mathsf{u\ \ committed}(s) = |\{V_{L\ ,i} \in lastVals : \mathsf{seq}(L') < \max(s, \mathsf{seq}(L)). \tag{3}$$

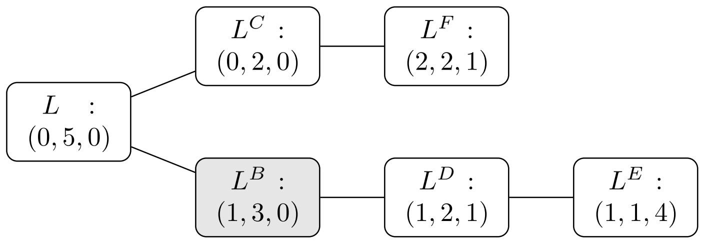

Figure 4: Ledger ancestry annotated with tuple of supp_{tip},supp_{branch}, and $\mathsf{u\ \ committed}$) from the perspective of the node that last validated L^F. Ledger L^D is preferred.

Figure 4 shows a motivating example, where each ledger is annotated with the tuple of ($\mathsf{supp}_{tip}, \mathsf{supp}_{branch}, \mathsf{u\ \ committed}$) from the perspective of a node that last validated L^F. There are 5 trusted nodes, two that last validated L^F and one each validating L^B, L^D and L^E. The preferred branch strategy determines that L^D is preferred.

The preferred branch protocol is provided in lgorithm 3. Intuitively, the idea is for each node to be conservative and only switch to a different branch when it knows enough nodes have committed to that chain of ledgers such that an alternative chain cannot have more support. The preferred ledger is found by walking the ancestry tree, starting from the ledger L that is the common ancestor ledger of the most recently validated ledgers. We then select the child ledger $L' \in childre\ \ (L)$ with highest $\mathsf{supp}_{branch}(L')$, that would still have the most support even if all $\mathsf{u\ \ committed}(\mathsf{seq}(L'))$ picked a conflicting sibling ledger. If we cannot select a child of L satisfying this requirement, then L is the preferred ledger. If we can find a preferred child L', then we repeat the process on the children of L'. In order to ensure a total ordering on ledgers and break ties between sibling ledgers, we rely on a function $\phi(L', L'')$, which is 1 if the hash of $L' > L''$ and is 0 otherwise. If the ledger found is an ancestor of our current working ledger $\tilde{L}$, we keep $\tilde{L}$ is the preferred ledger, since we do not yet know we are on the wrong branch.

Note that in cases of extreme asynchrony, a branch may only be initially supported by a single node when it becomes preferred. That means other nodes must verify the protocol invariants of that ledger before switching to deliberate on it.

4 nalysis

Having described the XRP LCP from a protocol perspective, we now formally prove results about its safety and liveness.

4.1 Safety

In this section we will prove conditions on the network configuration that guarantee different nodes running the XRP LCP will remain consistent.

In the following analysis, we find it convenient to assume that fully validating a ledger *does not* fully validate its ancestors. s we will find, it turns that it may be possible in some configurations for two nodes to fully validate contradictory ledgers with different sequence numbers even if fully validating contradictory ledgers with the same sequence number is impossible. The former is just as problematic as the latter, but we find it convenient to separate out the two failure types and prove conditions preventing them separately.

For the initial analysis, we make a simplifying assumption. Later in the paper we will reanalyze the problem without these assumptions. The assumption we make is that a Byzantine faulty node cannot convince two honest nodes that it validated different ledgers. This assumption was used in the original whitepaper [10] and is rationalized by the idea that all communication is done through generic multicast over a peer-to-peer network, so that the "echoes" of two contradictory messages would be noticed in time for the message to be ignored, since the messages are signed. Unfortunately this assumption does not hold in a fully asynchronous network, since a network partition could segregate the contradictory messages long enough for damage to occur. We call this assumption **Byzantine accountability**, and will always reference when we are using it as an assumption.

ssuming Byzantine accountability holds, we now analyze when it is possible for two nodes to fully validate different ledgers in a single round of consensus. Modifying the notation slightly, the condition suggested in the whitepaper [10] was that two nodes ${}_i$, ${}_j$ cannot fully validate conflicting ledgers if

$$|\mathsf{UNL}_i \cap \mathsf{UNL}_j| \geqslant \max\{ {}_i - q_i, \; {}_j - q_j\}.$$

With quorums of 80% as suggested, this condition is more easily identified with the actual condition given in the whitepaper,

$$|\mathsf{UNL}_i \cap \mathsf{UNL}_j| \geqslant 0.2 \max\{ {}_i, \; {}_j\}.$$

In a later independent analysis, rmknecht et al. [1] showed that this condition is incorrect. They instead suggest that ${}_i$, ${}_j$ cannot fully validate different ledgers if *and only if*

$$|\mathsf{UNL}_i \cap \mathsf{UNL}_j| > 2 \max\{ {}_i - q_i, \; {}_j - q_j\}.$$

It is true that if the above condition holds then ${}_i$ and ${}_j$ cannot fully validate different ledgers. However, the converse is not true as the following proposition shows.

Proposition 1. *ssuming Byzantine accountability, two honest nodes* ${}_i$, ${}_j$ *cannot fully validate different ledgers with the same sequence number iff*

$$|\mathsf{UNL}_i \cap \mathsf{UNL}_j| > {}_i - q_i + {}_j - q_j.$$

Note that $2\max\{\ {}_i - q_i,\ {}_j - q_j\} \geqslant\ {}_i - q_i +\ {}_j - q_j$, but ${}_i - q_i +\ {}_j - q_j$ is strictly smaller whenever ${}_i - q_i \neq\ {}_j - q_j$. Thus the condition suggested by rmknecht et al. is sufficient but not necessary.

Proof of proposition 1. We first prove sufficiency. Suppose ${}_i$ fully validates the ledger L and $|\mathsf{UNL}_i \cap \mathsf{UNL}_j| >\ {}_i - q_i +\ {}_j - q_j$.

Let S be the set of nodes in UNL_i that validated L. Since ${}_i$ fully validated L, $|S| \geqslant q_i$. By Byzantine accountability, every node in $S \cap \mathsf{UNL}_j$ could not have sent a validation to ${}_j$ for any ledger $L' \neq L$. Thus it suffices to show that $|S \cap \mathsf{UNL}_j| >\ {}_j - q_j$, since then there cannot be any ledger $L' \neq L$ with q_j support in UNL_j.

By the overlap hypothesis, we have

$$\begin{aligned}
|S \cap \mathsf{UNL}_j| &= |S| - |S \setminus \mathsf{UNL}_j| \\
&\geqslant |S| - |\mathsf{UNL}_i \setminus \mathsf{UNL}_j| \\
&= |S| - (|\mathsf{UNL}_i| - |\mathsf{UNL}_i \cap \mathsf{UNL}_j|) \\
&> |S| - (\ {}_i - (\ {}_i - q_i +\ {}_j - q_j)) \\
&\geqslant q_i - (\ {}_i - (\ {}_i - q_i +\ {}_j - q_j)) \\
&=\ {}_j - q_j.
\end{aligned}$$

For necessity, first suppose $|\mathsf{UNL}_i \cap \mathsf{UNL}_j| \quad {}_j - q_j$. Then all the nodes in UNL_i can validate L, while all the nodes in $\mathsf{UNL}_j \setminus \mathsf{UNL}_i$ validate L', and by assumption $|\mathsf{UNL}_j \setminus \mathsf{UNL}_i| \geqslant q_j$ so ${}_j$ fully validates L' while ${}_i$ fully validates L.

Now suppose $|\mathsf{UNL}_i \cap \mathsf{UNL}_j| \quad {}_i - q_i +\ {}_j - q_j$ and $|\mathsf{UNL}_i \cap \mathsf{UNL}_j| >\ {}_j - q_j$. Then

$$\begin{aligned}
|\mathsf{UNL}_i \setminus \mathsf{UNL}\ | &= |\mathsf{UNL}_i| - |\mathsf{UNL}_i \cap \mathsf{UNL}\ | \\
&\geqslant\ {}_i - (\ {}_i - q_i +\ {}_j - q_j) \\
&= q_i + q_j -\ {}_j.
\end{aligned}$$

Thus if all nodes in $\mathsf{UNL}_i \setminus \mathsf{UNL}\ $ validate L and ${}_j - q_j$ nodes in $\mathsf{UNL}_i \cap \mathsf{UNL}_j$ also validate L (which is possible since $|\mathsf{UNL}_i \cap \mathsf{UNL}_j| >\ {}_j - q_j$ by assumption), then ${}_i$ will receive $(q_i + q_j -\ {}_j) + (\ {}_j - q_j) = q_i$ validations for L and fully validate L. Meanwhile, if all the other nodes in UNL_j validate L', then since only ${}_j - q_j$ nodes in UNL_j validated L, ${}_j - (\ {}_j - q_j) = q_j$ nodes will validate L', so ${}_j$ will fully validate L'. □

ssuming a quorum of 80%, these overlap conditions may be summarized as follows:

- Schwartz et al.: Every pair of nodes needs an overlap of 20% the *maximum* size of their respective UNLs.
- rmknecht et al.: Every pair of nodes needs an overlap of 41% the *maximum* size of their respective UNLs.

- ctual condition: Every pair of nodes needs an overlap of 41% of the *average* size of their respective UNLs.

For the remainder of the paper we will no longer assume Byzantine accountability. In a live network, one would prefer absolute safety rather than relying on brittle heuristics that suggest it is unlikely that a Byzantine node could send conflicting messages to different nodes without getting caught. Thus we follow the research convention and assume that Byzantine nodes can send arbitrary messages to arbitrary nodes.

For any pair of nodes ${}_i$ and ${}_j$, let $\mathsf{O}_{i,j} = |\mathsf{UNL}_i \cap \mathsf{UNL}_j|$ and let $t_{i,j} = \min\{t_i, t_j, \mathsf{O}_{i,j}\}$. $t_{i,j}$ is the maximum number of allowed Byzantine faults in $\mathsf{UNL}_i \cap \mathsf{UNL}_j$, assuming that there are at most t_i faults in UNL_i and at most t_j faults in UNL_j.

The following lemma will be useful throughout the paper.

Lemma 2. *If an honest node ${}_i$ sees m validations for the ledger L with $\mathsf{seq}(L) = s$, then for any other honest node ${}_j$, there are at least $\mathsf{O}_{i,j}+m-{}_i-t_{i,j}$ honest nodes in UNL_j that validated L. Furthermore, there can be exactly $\mathsf{O}_{i,j}+m-{}_i-t_{i,j}$ honest nodes in UNL_j that validated L.*

Corollary 3. *If an honest node ${}_i$ sees m validations for the ledger L with $\mathsf{seq}(L) = s$, then ${}_j$ can see at most ${}_i + {}_j - \mathsf{O}_{i,j} - m + t_{i,j}$ validations for any contradictory ledger L' with $\mathsf{seq}(L') = s$. Furthermore, it is possible for ${}_j$ to see exactly ${}_i + {}_j - \mathsf{O}_{i,j} - m + t_{i,j}$ validations for a contradictory ledger L' with sequence number s.*

Proof. If an honest node validates L, then ${}_j$ cannot receive a validation for any contradictory ledger L' from it. By lemma 2, there are at least $\mathsf{O}_{i,j}+m-{}_i-t_{i,j}$ honest nodes that validate L. If every other node in UNL_j sends a validation to ${}_j$ for some contradictory ledger L', then ${}_j$ can receive up to (and including, by the secondary clause of lemma 2)

$${}_j - (\mathsf{O}_{i,j} + m - {}_i - t_{i,j}) = {}_i + {}_j - \mathsf{O}_{i,j} - m + t_{i,j}$$

validations for L' with $\mathsf{seq}(L') = s$. □

Note that corollary 3 does not preclude the possibility that ${}_j$ will see more than ${}_i + {}_j - \mathsf{O}_{i,j} - m + t_{i,j}$ validations for a contradictory ledger with a larger sequence number than s. Indeed, without assuming totality, it turns out that such an occurrence is possible, which forces the algorithm to use much tighter safety margins.

Proof of lemma 2. The proof is similar to the proof of proposition 1.

Suppose ${}_i$ sees m validations for L. gain letting S be the set of nodes in UNL_i that sent validations to ${}_i$ for L, then

$$\begin{aligned} |S \cap \mathsf{UNL}_j| &= |S| - |S \setminus \mathsf{UNL}_j| \\ &\geqslant |S| - |\mathsf{UNL}_i \setminus \mathsf{UNL}_j| \\ &= |S| - ({}_i - \mathsf{O}_{i,j}) \\ &= m - {}_i + \mathsf{O}_{i,j}. \end{aligned}$$

There could be $t_{i,j}$ Byzantine nodes in $S \cap \mathsf{UNL}_j$ that send ${}_j$ a validation for something other than L, so at least $m - {}_i + \mathsf{O}_{i,j} - t_{i,j}$ *honest* nodes validated L.

For the second point, assume that every node in $\mathsf{UNL}_i \setminus \mathsf{UNL}_j$ validates L, $t_{i,j}$ Byzantine nodes in $\mathsf{UNL}_i \cap \mathsf{UNL}_j$ send a validation for L to ${}_i$ and L' to ${}_j$. Then there are exactly $m - {}_i + \mathsf{O}_{i,j} - t_{i,j}$ honest nodes in $\mathsf{UNL}_i \cap \mathsf{UNL}_j$ that send a validation for L to ${}_j$. Since every node in $\mathsf{UNL}_j \setminus \mathsf{UNL}_i$ can validate some ledger other than L, there can be exactly $m - {}_i + \mathsf{O}_{i,j} - t_{i,j}$ honest nodes in UNL_j that send a validation for L to ${}_j$. □

Proposition 4. ${}_i$ *fully validating some ledger L with $\mathsf{seq}(L) = s$ implies that ${}_j$ cannot fully validate any contradictory ledger with the same sequence number s iff $\mathsf{O}_{i,j} > ({}_i - q_i) + ({}_j - q_j) + t_{i,j}$.*

Proof. By letting $m = q_i$, corollary 3 tells us that ${}_i$ fully validating L implies that ${}_j$ can see at most ${}_i + {}_j - \mathsf{O}_{i,j} - q_i + t_{i,j}$ validations for any contradictory ledger with sequence number s.

Thus if

$$
\begin{aligned}
q_j &> {}_i + {}_j - \mathsf{O}_{i,j} - q_i + t_{i,j} \\
\mathsf{O}_{i,j} &> {}_i - q_i + {}_j - q_j + t_{i,j}
\end{aligned}
$$

then ${}_j$ cannot fully validate any contradictory ledger with seqeuence number s.

For necessity, if $\mathsf{O}_{i,j} \quad ({}_i - q_i) + ({}_j - q_j) + t_{i,j}$, then the second clause of corollary 3 implies that ${}_j$ can see exactly

$$
\begin{aligned}
{}_i + {}_j - \mathsf{O}_{i,j} - q_i + t_{i,j} &\geqslant {}_i + {}_j - (({}_i - q_i) + ({}_j - q_j) + t_{i,j}) - q_i + t_{i,j} \\
&= q_j
\end{aligned}
$$

validations for a contradictory ledger L', allowing ${}_j$ to fully validate L'. □

Once again assuming 80% quorums and 20% fault tolerance as in the whitepaper, this overlap condition can be summarized as requiring roughly 61% UNL overlaps.

To see why the overlap hypothesis in proposition 4 does not guarantee full safety, note that it is possible for a node to exit from deliberation for sequence s and then be unable to fully validate any ledger with sequence s, as the following example shows.

Example 5.

Consider a complete network with 10 nodes. Let X denote the first 5 nodes and Y denote the other 5 nodes. s shown in figure 5, suppose all the nodes in X begin deliberation proposing the transaction set $T = \{x_0, x_1\}$ and all the nodes in Y begin deliberation proposing the set $T' = \{x_0\}$. Thus for a transaction threshold of τ, receiving τ proposals for T will cause an honest node to propose

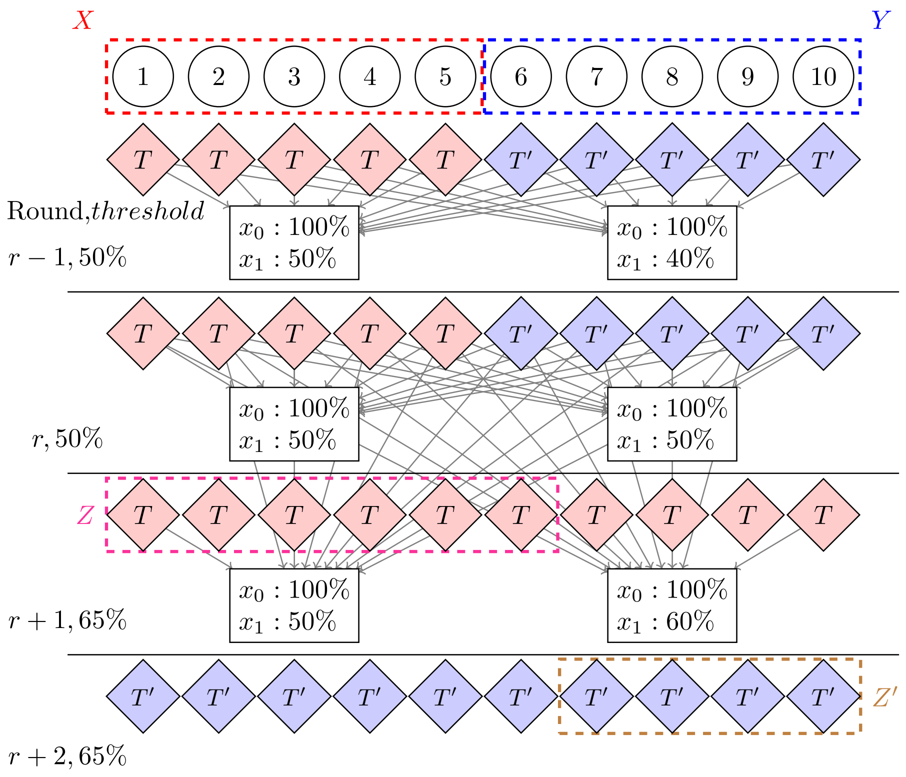

Figure 5: Schematic of example 5. The two node groups X and Y begin by proposing $T = \{x_0, x_1\}$ and $T' = \{x_0\}$ respectively. The left (right) boxes reflect proposals seen by node 1 (10) and are representive of all nodes in group X (Y). Gray arrows indicate proposals that were received in time to calculate the thresholds for a given round. Note that in round $r-1$, the proposal from node 5 was not received by nodes in Y. The two partitions Z and Z' are represented by the dashed boxes. Note that all nodes share the same single UNL.

T, while receiving less that τ proposals for T but τ proposals for either T *or* T' will cause an honest node to propose T', since $T \cap T' = T'$.

Let r be such that the transaction threshold is 50% in deliberation round r and 65% in round $r+1$. By the ratcheting threshold protocol described in subsection 3.1, such an r exists. During the first $r-1$ rounds, the nodes in X receive all the proposals, while the nodes in Y receive all the proposals except for one proposal from a node in X. Since the nodes in X propose T while the nodes in Y propose T', the nodes in X continue proposing T, while the nodes in Y only receive 4 proposals for T and continue proposing T' by the assumptions on T and T'.

Now in round r all nodes receive all proposals. This causes all nodes to propose T in round $r+1$. But in round $r+1$, a network failure causes none of the nodes to receive anyone else's proposals. Based on the most recently received deliberation proposals, everyone assumes that all the other nodes are still proposing what they proposed in the previous round. Thus the nodes in X see only 5 proposals for T while the nodes in Y see only 6 proposals for T (since they of course receive their own updated proposals). No one sees 65% support for x_1, but everyone sees 100% support for x_0, so everyone proposes T' in round $r+2$.

Now pick an arbitrary partition of the network into two sets, Z and Z'. The nodes in Z receive all the proposals from round $r+1$ but none of the proposals from round $r+2$. Thus they see 100% support for T and validate T. Meanwhile the nodes in Z' receive all the proposals from round $k+2$, see 100% support for T', and validate T'. Thus we can exit from deliberation with two arbitrary subsets validating different ledgers. In this case if $|Z| > 2$ and $|Z'| > 2$, then none of the nodes will fully validate a ledger during this consensus round.

Because of examples like example 5, we therefore make the broad assumption that deliberation can terminate with an arbitrary result. In practice, this may require a significantly degraded network, but is nonetheless a real risk. From a theoretical perspective, deliberation is therefore completely irrelevant; it is purely an optimization that makes it so that during civil executions most nodes will go into validation with the same ledger, allowing every node to fully validate usually, and it could be removed without fundamentally changing the protocol.

Thus we shift our focus towards validation without making any assumptions about the result of deliberation. We need to prove that if any node fully validates a ledger L, then it is never possible for a node to fully validate a ledger L' such that $\mathsf{seq}(L') \geqslant \mathsf{seq}(L)$ and L' is not a descendant of L. The following lemma provides the route for guaranteeing this.

Lemma 6. *If for every node* $_i$*, there are more than* $_i/2$ *honest nodes in* UNL_i *that submit a validation for some ledger* L *with* $\mathsf{seq}(L)$*, then for every node* $_i$ *more than* $_i/2$ *honest nodes in* UNL_i *will* always *submit validations for ledgers descended from* L*.*

Proof. We prove this by contradiction. If the lemma is not true, then under the lemma's hypotheses there must be some honest node which submits a validation for L and also eventually submits a validation for some ledger L' with $\mathsf{seq}(L') >$ $\mathsf{seq}(L)$ and L' not descended from L. Since correct nodes can only submit new validations for ledgers with sequence strictly greater than any ledger they have previously submitted a validation for, such a node must submit its validation for L' *after* having submitted its validation for L. Thus let $_i$ be the honest node that submitted a validation for L and later is the first to submit a validation for a ledger L' not descended from L.

The only way for $_i$ to later submit a validation for a ledger off of the L branch is if it runs the preferred branch algorithm and sees a ledger off the L branch as preferred. For a given s $\mathsf{seq}(L)$, let $\mathsf{pare\ t}(s, L)$ denote the ancestor of L with sequence s. Since $_i$ submitted a validation for L by assumption, it considers all validations for ledgers with sequence below $\mathsf{seq}(L)$ as uncommitted in the preferred branch protocol. But since $_i$ is assumed to be the first to switch away from the L branch, more than $_i/2$ nodes in UNL_i cannot have sent out a validation for any ledger L' with $\mathsf{seq}(L') \geqslant s$ and L' not descended from L. Thus for every s $\mathsf{seq}(L)$, $_i$ sees a majority of nodes in UNL_i as being either uncommitted support at s or branch support for $\mathsf{pare\ t}(s, L)$. In other words, for all s $\mathsf{seq}(L)$,

$$\mathsf{supp}_{branch}(\mathsf{pare\ t}(s, L)) > \ _i/2 - \mathsf{u\ committed}(s).$$

For a given $s < \mathsf{seq}(L)$, suppose $\mathsf{pare\ t}(s, L)$ is the current base ledger in the loop on line 10 of lgorithm 3. Then either $C[0] = \mathsf{pare\ t}(s+1, L)$ or there is some $L' \neq \mathsf{pare\ t}(s+1, L)$ with $\mathsf{pare\ t}(L') = \mathsf{pare\ t}(s, L)$ and $C[0] = L'$. In the latter case, the branch support for $C[1]$ must be at least equal to the branch support for $\mathsf{pare\ t}(s+1, L)$ (breaking ties with ϕ) by definition of the ordering of C. Further, the branch support for $C[0]$ must be less than $_i/2$. Thus in line 16,

$$\begin{aligned} \Delta &= \mathsf{supp}_{branch}(C[0]) - \mathsf{supp}_{branch}(C[1]) + \phi(C[0], C[1]) \\ & \quad \mathsf{supp}_{branch}(C[0]) - \mathsf{supp}_{branch}(\mathsf{pare\ t}(s+1, L)) + 1 \\ &< \ _i/2 - \mathsf{supp}_{branch}(\mathsf{pare\ t}(s+1, L)) + 1 \\ &< \ _i/2 - (\ _i/2 - \mathsf{u\ committed}(s+1)) + 1 \\ & \quad \mathsf{u\ committed}(s+1) + 1, \end{aligned}$$

so the condition $\Delta > \mathsf{u\ committed}(s+1)$ is always false. Thus in the latter case $_i$ sees $\mathsf{pare\ t}(s, L)$ as the preferred ledger. In the former case, $_i$ either sees $\mathsf{pare\ t}(s+1, L)$ as the preferred ledger or continues the loop with $\mathsf{pare\ t}(s+1, L)$ as the base ledger. By induction, $_i$ is guaranteed to see some ledger on the L branch as preferred, so $_i$ cannot leave the L branch, contradicting our assumption about $_i$. □

If more than $_i/2$ honest nodes in UNL_i only ever validate descendants of L, then certainly $_i$ cannot fully validate a ledger that doesn't descend from L, since otherwise there would be $q_i > \ _i/2$ nodes that sent validations for some ledger L' with sequence number s' that *doesn't* descend from L. Thus we can show that consensus is safe if we can guarantee that if any honest node fully validates a ledger L with sequence number s, then for every node ${}_i$, more than ${}_i/2$ honest nodes in UNL_i must have validated L. The following proposition gives the overlap condition guaranteeing this property.

Proposition 7. *Given two honest nodes* ${}_i$, ${}_j$, ${}_i$ *fully validating a ledger* L *with* $\mathsf{seq}(L) = s$ *implies that there are more than* ${}_j/2$ *honest nodes in* UNL_j *which validated* L *iff* $\mathsf{O}_{i,j} > {}_j/2 + {}_i - q_i + t_{i,j}$.

Proof. The proof is directly analogous to the proof of proposition 4, except rather than bounding the formula by q_j we bound it by ${}_j/2$. □

Theorem 8. *XRP LCP guarantees fork safety if* $\mathsf{O}_{i,j} > {}_j/2 + {}_i - q_i + t_{i,j}$ *for every pair of nodes* ${}_i$, ${}_j$. □

Note that although proposition 7 is an iff statement, the overlap condition in theorem 8 is only sufficient but not necessary for XRP LCP safety. This is because lemma 6 is not an iff statement. Further, there may be some validation configurations that cannot come out of deliberation, breaking our broad assumption that anything can come out of deliberation. However, it is the weakest condition that can be expressed purely as a bound on the size of overlaps.

Once again assuming 80% quorums and 20% faults, the overlap condition in theorem 8 can be summarized as requiring roughly $> 90\%$ UNL overlaps. lthough quite a narrow margin (and certainly far more narrow than originally expected), this does still allow a small amount of variation, which is very important for the XRP Ledger network's transition to a recommended UNL comprised of independent entities. Having some flexibility in the UNLs is important both for after the diversification of trusted operators (as one can never guarantee total agreement on participants when the participants are independent entities) and also during the diversification process (if tiny disagreements during changes to the UNL list could cause a fork, then diversification would always be too risky to execute).

4.2 Liveness

Now that we have a concrete metric of when it is impossible for the network to fork, we would like to know when it makes forward progress. If a live network stops making forward progress, that is almost as damaging as forking, since businesses might be relying on being able to make transfers on time. Unfortunately, by the FLP result [5] it is impossible to guarantee forward progress in a fully asynchronous network.

In the absence of being able to prove that the network always makes forward progress, we would like to at least be able to prove that the network cannot get "stuck". In other words, that the network cannot get into a state in which some honest nodes can never fully validate a new ledger.

Unfortunately, it is very difficult in general to guarantee forward progress with XRP LCP. The following example shows that it is possible to get stuck even with 99% UNL overlaps and no Byzantine faults.

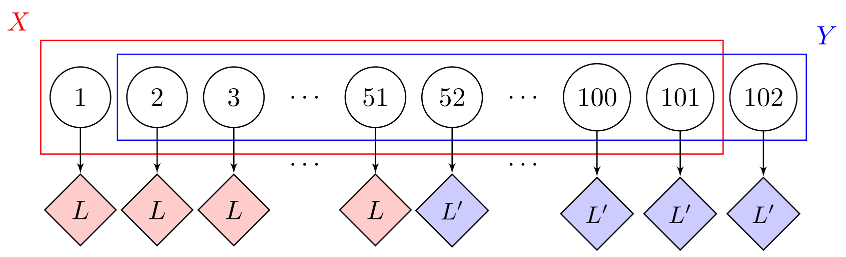

Figure 6: Example of stuck network with 99% UNL overlap and no Byzantine faults.

Example 9.

Consider a network of 102 peers drawin in figure 6. There are two UNLs, the red $X = \{ \ _1, \ _2, \ldots, \ _{101}\}$ and blue $Y = \{ \ _2, \ _3, \ldots, \ _{102}\}$. Peers $1-51$ use X and peers $52-102$ use Y. There are two ledgers, L and L'. The nodes listening to X all validate a descendant of L, while the nodes listening to Y all validate a descendant of L'. Since $51 > 0.5|X|$ nodes in X validate a descendant of L. Thus according to the preferred branch protocol all, the nodes listening to X cannot switch branch to L'. Similarly, since $51 > 0.5|Y|$ nodes in Y all validate a descendant of L, the nodes listening to Y cannot switch branch to L'. The network cannot ever rejoin without manual intervention.

s of the time this paper was written, the recommended XRP trust model has all nodes listening to either a single UNL consisting of 5 nodes, or a UNL consisting of those 5 nodes plus one extra node (typically the extra node is oneself; nodes that listen to these extended UNLs are thus called "leaves", since they branch off slightly from the core network). The short-term plan for decentralization involves expanding to a larger, but still agreed-upon, single UNL and diversifying the node operators. Losing forward progress while adjusting to a new node list is not a huge problem (since as soon as everyone agrees on the node list again forward progress will resume, and the previous section guarantees for "small" changes it will not fork during the interim); thus we could at least get a positive result by proving that the network cannot get stuck in a complete graph with leaves.

The following lemma simplifies the problem to only needing to verify that complete networks cannot get stuck.

Lemma 10. *Suppose N is a closed subset of the network (i.e., the UNL of every node in N is contained in N, so that from the perspective of the nodes inside of N, N is the entire network) which cannot get stuck and cannot fork. Suppose $\ _i$ is a node not in N such that $\mathsf{UNL}_i = \{ \ _i\} \cup N'$, where $N' \subseteq N$ and $|N'| \geqslant q_i$. Then $N \cup \{ \ _i\}$ cannot get stuck either.*

Proof. Since N cannot get stuck, it is always true that all the nodes in N will eventually fully validate a new ledger. Since N cannot fork, all the nodes in N

can only fully validate the same ledger, so eventually there is some ledger L that gets fully validated by every node in N. Thus every node in N will validate L, and since $|\mathsf{UNL}_i \cap N| = |N'| \geqslant q_i$, $\;_i$ can fully validate L as well. Thus it is always true that $\;_i$ will eventually fully validate a new ledger, so $\;_i$ cannot get stuck. ☐

Theorem 11. *Suppose for all nodes the UNL quorum is set to* $\; - \lfloor (\; - 1)/k \rfloor$ *for some integer* k*. XRP LCP cannot get stuck in a network consisting of a single agreed-upon UNL* X *of size at least* k *along with an arbitrary number of leaf validators.*

Proof. For any leaf validator $\;_i$, $\;_i = |X| + 1$; since $q_i = |X| + 1 - \lfloor (|X| + 1 - 1)/k \rfloor = |X| + 1 - \lfloor |X|/k \rfloor \quad |X|$, by lemma 10 it suffices to show that complete networks cannot get stuck.

Thus suppose there is a single agreed-upon UNL X. Suppose in some round r all validation messages are delivered quickly enough so that Byzantine accountability holds and every node sees every other node's validations from round r. Then the preferred branch algorithm will deterministically push all nodes onto the most popular ledger L. Thus in the next round all nodes will validate a child of L. If messages sent during deliberation are delivered synchronously, then all nodes see the same proposals from round 1 and either all exit deliberation with the same ledger (if some transaction set is shared with 80% of the nodes in X) or else build their proposal for round 2 deterministically from the same proposals from round 1, so all nodes propose the same set of transactions in round 2, and all nodes leave deliberation with the same set of transactions. Thus every node submits a validation for the same ledger, and all nodes fully validate that ledger. ☐

5 Conclusion

We have given a detailed description and thorough analysis of the XRP Ledger Consensus Protocol, which is a protocol for reaching consensus without universal agreement of network participants. Our work corrects prior analysis in [10, 1]. We show in theorem 8 that roughly $> 90\%$ agreement on participants is needed to ensure network safety. In the restricted case of a single expanding UNL with leaves, theorem 11 shows we can always make forward progress during periods when no nodes are faulty and network messages are delivered with bounded delay. In the more general case with even minor disagreement of participants, we cannot guarantee that the network makes forward progress.

It is an open question whether the sufficient overlap condition in theorem 8 can be improved by a more detailed consideration of the trust topology of the network.   more complicated condition that does not simply take into account pairwise overlaps but also the way in which messages flow indirectly through the network might have potential for giving a more precise condition for guaranteeing safety. Likewise, we might be able to leverage the trust structure to better explain cases when deliberation can fail, which in turn might allow a more refined understanding on forward progress.

lthough we have shown that XRP LCP is provably safe with the current and near-future network structures, in an attempt to alleviate some of its shortcomings, in the sibling paper [6] we present an alternative consensus protocol called Cobalt. Similar to XRP LCP, Cobalt can also be used in networks that lack uniform agreement on participants or trust, but makes forward progress at a steady rate in the presence of maximum tolerated Byzantine faults and arbitrary asynchrony. It only needs $> 60\%$ overlap to match the XRP LCP safety tolerances. Cobalt also has several other properties that make it simpler to analyze the health of networks in practice. For these reasons we believe Cobalt represents an encouraging direction for even greater future decentralization of the XRP network.

cknowledgments. Thank you to Haoxing Du and Joseph McGee for collaborating on the early stages of this research. We also thank David Schwartz and Stefan Thomas for providing useful discussions and guidance, and Rome Reginelli for careful editing. We lastly thank all our colleagues at Ripple for their support. This work was funded by Ripple.

References

[1] Frederik rmknecht, Ghassan O. Karame, vikarsha Mandal, Franck Youssef, and Erik Zenner. Ripple: Overview and outlook. In *Trust and Trustworthy Computing: 8th International Conference, TRUST 2015, Proceedings*, pages 163–180. Springer International Publishing, 2015. ISBN 978-3-319-22846-4. doi: 10.1007/978-3-319-22846-4_10.

[2] V. Buterin and V. Griffith. Casper the friendly finality gadget. *rXiv e-prints*, October 2017. URL `https://arxiv.org/abs/1710.09437`.

[3] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, C , US , 1999. USENIX ssociation. ISBN 1-880446-39-1.

[4] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. CM*, 43(2):225–267, March 1996. ISSN 0004-5411. doi: 10.1145/226643.226647. URL `http://doi.acm.org/10.1145/226643.226647`.

[5] Michael J. Fischer, Nancy . Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. CM*, 32(2):374–382, pril 1985. ISSN 0004-5411. doi: 10.1145/3149.214121.

[6] Ethan MacBrough. Cobalt: BFT governance in open networks. *rXiv e-prints*, February 2018.

[7] Satoshi Nakamoto. Bitcoin: peer-to-peer electronic cash system. 2009. URL `http://www.bitcoi .org/bitcoi .pdf`.

[8] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. CM*, 27(2):228–234, pril 1980. ISSN 0004-5411. doi: 10.1145/322186.322188. URL `http://doi.acm.org/10.1145/322186.322188`.

[9] Fred B. Schneider. Distributed systems (2nd ed.). chapter Replication Management Using the State-machine pproach, pages 169–197. CM Press/ ddison-Wesley Publishing Co., New York, NY, US , 1993. ISBN 0-201-62427-3.

[10] David Schwartz, Noah Youngs, and rthur Britto. The Ripple protocol consensus algorithm. *Ripple Labs Inc White Paper*, 2014. URL `https://ripple.com/files/ripple_co se sus_whitepaper.pdf`.

[11] Yonatan Sompolinsky and viv Zohar. Secure high-rate transaction processing in bitcoin. In *Financial Cryptography and Data Security*, pages 507–527, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-47854-7.

[12] Stefan Thomas. How we are further decentralizing the XRP ledger to bolster robustness for enterprise use, 2017. URL `https://ripple.com/i sights/how-we-are-further-dece tralizi g-the-ripple-co se sus-ledg`

lgorithms

In this appendix, we provide pseudo-code for the three components of Ripple consensus described in section 3. Note that under the network model in section 2, a message which is **broadcast** by a node $_i$ is **receive**d by all nodes, including $_i$ itself.

lgorithm 1 Deliberation from the perspective of $\ _i$

```
$s_{max} \leftarrow 0$                    ▷ Track the largest validated ledger sequence number

function START($L$)
    $\tilde{L} \leftarrow L, r \leftarrow 0$
    $T \leftarrow$ pending transactions
    $props \leftarrow \{\}$                              ▷ $props$ is a map from node to proposal
    Initialize $props$ with previously received proposals for $\tilde{L}$
    broadcast $P_{T,r,L,i}$
end function

receive $P_{T\ ,r\ ,L,j}$ do
    if $\ _j \in \mathsf{UNL}_i$ and $\tilde{L} = L$ and $r' > props[j].r$ then
        $props[j] = P_{T\ ,r\ ,L,j}$
    end if
end receive

function UPDATE()              ▷ Called at a regular, protocol defined interval
    if $\tilde{L} \neq$PREFERREDLEDGER() then
        START(PREFERREDLEDGER())
    else
        UPDATEPOSITIO ()
        if CHECKCO SE SUS() then
            $\tilde{L} \leftarrow$ APPLY($T, \tilde{L}$)
            if seq($\tilde{L}$) $> s_{max}$ then
                broadcast $V_{L,i}$
                $s_{max} \leftarrow$ seq($\tilde{L}$)
            end if
            START($\tilde{L}$)
        end if
    end if
end function
```

lgorithm 1 Deliberation from the perspective of $_i$ (continued)

```
29: function UpdatePositio ()
30:     $T_{all} \leftarrow \quad _{P \quad props} P.T$          ▷ Set of all proposed transactions
31:     $\tau \leftarrow threshold(r) \quad _i$
32:     $T \leftarrow \{x \in T_{all} : \text{SUPPORT}(x) > \tau\}$          ▷ SUPPORT is number of nodes
                                                   proposing $x$
33:     $r \leftarrow r + 1$
3 :     broadcast $P_{T,r,L,i}$
35: end function

36: function CheckCo se sus()
37:     $_a \leftarrow |\{P \in props : P.T = T\}|$   ▷ Node positions agreeing with our posi-
                                                   tion
38:     return $_a \geq q_i$
39: end function
```

lgorithm 2 Validation from the perspective of $_i$

```
1: $vals = \{\}$          ▷ $vals$ is a map from $L$ to the set of nodes that validated $L$

2: receive $V_{L,j}$ do
3:     if $_j \in \mathsf{UNL}_i$ then
 :         $vals[L] \leftarrow vals[L] \cup j$
5:         if $|vals[L]| \geq q_i$ and $\mathsf{seq}(L) > \mathsf{seq}(\hat{L})$ then
6:             $\hat{L} \leftarrow L$
7:         end if
8:     end if
9: end receive
```

lgorithm 3 Preferred branch from the perspective of $_i$

```
$lastVals = \{\}$        ▷ $lastVals$ is a map from trusted node to its most recent validated ledger
**receive** $V_{L,j}$ **do**
    **if** $_j \in \mathsf{UNL}_i$ **then**
        $lastVals[j] \leftarrow L$
    **end if**
**end receive**

**function** PreferredLedger( )
    $L \leftarrow$ earliest common ancestor of ledgers in $lastVals$
    $do\ \ e =$ False
    **while** $|childre\ \ (L)| > 0$ and not $do\ \ e$ **do**
        $C \leftarrow$ Sorted array of $childre\ \ (L)$ by descreasing $\mathsf{supp}_{branch}$, breaking ties with $\phi$
        $\Delta \leftarrow \mathsf{supp}_{branch} C[0]$
        **if** $|childre\ \ (L)| > 1$ **then**
            $\Delta \leftarrow \Delta - \mathsf{supp}_{branch} C[1] + \phi(C[0], C[1])$
        **end if**
        **if** $\Delta > \mathsf{u}\ \ \mathsf{committed}(\mathsf{seq}(L) + 1)$ **then**
            $L \leftarrow C[0]$
        **else**
            $do\ \ e \leftarrow$ True
        **end if**
    **end while**
    **if** $L \in a\ \ cestors(\tilde{L})$ **then**
        **return** $\tilde{L}$
    **else**
        **return** $L$
    **end if**
**end function**
```