JEPSEN

Radix DLT 1.0-beta.35.1

Kyle Kingsbury

2022-02-05

Radix DLT is a distributed, byzantine-fault-tolerant ledger for cryptocurrencies based on delegated proof-of-stake. We evaluated Radix DLT at version 1.0-beta.35.1, 1.0.0, 1.0.1, and 1.0.2, as well as various development builds—all versions associated with Radix's Olympia technology milestone. We found 11 safety errors, ranging from stale reads which violated per-server monotonicity, to aborted and intermediate reads, as well as the partial or total loss of committed transactions. At least some of these issues affected users of the Radix Olympia Public Network. We also observed what appeared to be liveness issues with indeterminate transactions and performance degradation during single-node faults. RDX Works reports that all safety issues we found had been resolved in version 1.1.0, in large part by replacing the archive API subsystem with a new Gateway API. RDX Works also reports that their internal load tests show they have resolved the issue with indeterminate transactions. Jepsen has not verified these claims. RDX Works has also written a companion blog post to this report. This work was funded by Radix Tokens (Jersey) Limited, and conducted in collaboration with RDX Works Ltd, in accordance with the Jepsen ethics policy.

1 Background

Radix DLT (Distributed Ledger Technology) is a distributed ledger: a serializable log of transactions over a state machine, along with mechanisms for querying that state. Throughout this report, "Radix" refers to the Radix DLT software.

RDX Works, the makers of Radix, intend to develop and release a smart contract system à la Ethereum—a feature which is now available as a part of the Alexandria developer preview. The implementation discussed in this report, and which is presently deployed as the Radix Olympia Public Network, does not include smart contracts. Instead, it provides a set of *accounts* which can hold and transfer units of virtual currencies, called *tokens*. A "native token" called *XRD* is used for core Radix operations like paying network fees. Users can create their own tokens as well.

Radix's homepage advertises "1000x More Scalability than Ethereum / Solana / Polkadot / Avalanche / …," which refers to their parallelized Byzantine-fault-tolerant (BFT) consensus protocol named Cerberus. Rather than serialize all operations through a single instance of a consensus system, Cerberus runs several independent shards of consensus, each based on the HotStuff BFT consensus protocol. For cross-shard operations, Cerberus establishes transient consensus instances which "braid" those shards together. This should allow Radix to offer linearly scalable transaction throughput.

From May through November 2021, Radix's homepage advertised 1.4 million transactions per second using a 2019 sharded consensus prototype.

> 1.4m TPS on a DLT
>
> Radix' last consensus algorithm 'Tempo' publicly achieved 1.4m TPS in 2018, the current world record. The new algorithm 'Cerberus' is theoretically infinitely scalable and builds on many of the insights we learnt from replaying the entire history of Bitcoin in less than 30 minutes!

There appears to be some confusion over these prototype test results versus the behavior of the Radix DLT software which currently runs the Radix Olympia Public Network. Claims of 1.4 million transactions per second and unlimited scalability are frequently repeated by proponents of Radix on social media. For example, StrongHandzSP90 writes:

> Radix DLT #XRD is an innately sharded DLT that is NOT a blockchain and has infinite scalability (1.4million TPS confirmed and verifiable), enhanced security and decentralised all WITHOUT BREAKING COMPOSABILITY. This is the future of #DeFi!

When asked, RDX Works executives informed Jepsen that blockchain/DLT readers would normally understand present-tense English statements like these to be referring to potential future behavior, rather than the present.

Jepsen is no stranger to ambitious claims, and aims to understand, analyze, and report on systems as they presently behave—in the context of their documentation, marketing, and community understanding. Jepsen encourages vendors and community members alike to take care with this sort of linguistic ambiguity.

Indeed, as the Radix Roadmap clarifies, the Cerberus sharded consensus system is not yet implemented. Instead Radix currently processes all transactions through a single consensus instance, also based on HotStuff. This means the Olympia Public Network has constant, rather than linear scalability. The roadmap indicates that Olympia offers 50 transactions per second, and while Radix presently declines to publish network throughput statistics, an independent dashboard indicates the Olympia Public Radix network is currently processing three to five transactions per second. In our testing with five to ten-node clusters of m5.large instances, we saw transactions start timing out with as little as one request per second, and goodput generally peaked at ~16 transactions per second.[1]

During our analysis, Radix's documentation explained that Radix nodes can run in three principal ways:

> An individual Radix Node has its own account on the Radix network. It can be configured in three different ways depending on its purpose:
>
> A Full Node simply connects to the network, synchronizes ledger state, and observes the status of the network. It can be thought of as a kind of "wallet" that is connected directly to the network, with the Node's own account available for programmatic control.
>
> A Validator Node starts life as a Full Node, but has also "registered" itself to the network as a Validator by submitting a special transaction from its account. Registration means that it may now accept XRD token "stake" and potentially be included in the validator set of 100 nodes that conduct network consensus.
>
> An Archive Node not only synchronizes ledger state (as with a Full Node) but also heavily indexes that state to support the JSON-RPC API endpoint the Archive Node offers. The Node API is useful for client applications, like wallets or exchange integrations, as well as general account/transaction queries and programmatic control of accounts.

Both full and validator nodes can also be archive nodes. Archive nodes are simply those which have set `api.archive.enable = true`.

Validators do the work of consensus. Accounts on a Radix network can *stake* a portion of their XRD to one or more validators, indicating they believe those validators to be trustworthy network participants. In the Radix Olympia Public Network, the 100 validators with the highest stake are selected to execute the consensus protocol. Every *epoch* (a period determined by the number of consensus rounds) a fresh set of validators is selected. The Radix protocol is intended to guarantee the safety of the ledger state and the liveness of the network at large, under the condition that no more than 1/3 of active stake supports nodes which are either unresponsive or malicious.

Radix transactions (in the Olympia series of releases) are an ordered series of operations performed by a single account. The most common operations come in three basic flavors:

1. Transferring tokens to another account.
2. Staking XRD to a validator.
3. Unstaking XRD from a validator.

Transactions can not perform read operations. However, clients can observe the state of the Radix ledger by querying a node's archive API, which provides methods for fetching the status of a transaction, the balance of a single account, and the complete history of transactions on a single account. The archive API is essentially a read-only layer around Radix's ledger, and during our testing was the primary way for users to observe Radix state.[2]

1.1 Safety and Liveness

This report discusses safety and liveness properties. As Lamport's 1977 Proving the Correctness of Multiprocess Programs succinctly put it:

> A safety property is one which states that something [bad] will *not* happen.
>
> A liveness property is one which states that something [good] *must* happen.

These senses have been standard in concurrent and distributed systems verification for several decades; their definitions are widely understood throughout the field. We use these senses throughout Jepsen reports.

Four days prior to publication, RDX Works informed Jepsen that the blockchain/DLT community had developed idiosyncratic definitions of safety and liveness. Their definitions are:

> A *safety violation* is defined as two healthy consensus nodes disagreeing on what is the correct ledger state. Most notably, this is a result of a *double-spend* having been permitted. Specifically in the Radix Olympia Network, this means a single substate being successfully "downed" more than once in the ledger.

[1]Our tests used the standard Radix JSON-RPC API to construct and submit transactions, and used an exponential distribution of requests across a rotating pool of accounts to achieve a variety of high- and low-contention scenarios. Radix's test figures bypassed large parts of the API, and used the Bitcoin ledger as their source of transactions. It seems reasonable to imagine that Bitcoin's history contains no concurrent conflicting transactions, since these conflicts would have been resolved via consensus at the time of submission. These factors could help account for our observed performance disparities.

[2]RDX Works is replacing the Archive API with a completely new service for interacting with Radix's internal state: the Network Gateway. Its first release was January 19th, 2022, after the completion of this work.

> A *liveness break* is defined as the network halting and being unable to process further transactions.

These definitions are of course specific examples of safety and liveness properties, but they allow many behaviors which would reasonably be termed safety or liveness issues. For example, a system which acknowledges user transactions and then throws them away on every node trivially satisfies this safety property, but one would be hard-pressed to call such a system *safe*.

To Jepsen's surprise, RDX Works asserted that phenomena such as aborted read, intermediate read, and lost writes do not constitute safety violations (in the DLT sense). RDX Works claims that to describe these errors as safety violations would not be understood by readers from a DLT background; this report is therefore "factually incorrect". On these grounds, RDX Works requested that Jepsen delete any mention of our findings from the abstract of this report.

Jepsen respectfully declines to do so.

1.2 Consistency

As of November 5, 2021, Radix's documentation offered essentially no description of Radix's consistency guarantees or behavior during faults. However, in our initial conversations RDX Works staff indicated that transactions should be strict serializable. This means that transactions appear to execute in a total order, such that each transaction takes effect at some point in time between that transaction's submission and confirmation.

On the other hand, RDX Works indicated that read operations are *not* intended to be strict serializable. Instead they read from a snapshot of committed state on the local node. This means that histories of transactions and reads should still be serializable, but reads may observe stale state.

Moreover, each archive node is supposed to enforce a sort of local sequential consistency: reads against a single node should observe monotonically increasing states, and successive transactions and reads to the same node should be executed in order. When a node says that a transaction is confirmed, any future read on that same node is guaranteed to reflect that transaction.

2 Test Design

We designed a test suite for Radix DLT using the Jepsen testing library. Our test suite created local clusters of Radix DLT nodes, completely independent from public Radix networks, and could also (with limitations) interact with Stokenet: a public test network. Our tests ran on 5–10 node clusters of Debian Buster machines, in both LXC and on EC2 virtual machines. Our LXC tests ran on a single 48-way Xeon with 128 GB of RAM. In EC2, we used m5.large instances backed by EBS for each node. Every node in our local clusters was configured as a validator node with the archive API enabled.

We tested Radix version 1.0-beta.35.1, and moved on to versions 1.0.0, 1.0.1, 1.0.2, and a series of development builds from June 15th through November 5th, 2021. This series of production releases were associated with Radix's Olympia technology milestone: the first iteration of the Radix Public Network.

Our tests submitted transactions and read the state of accounts using the Radix Java client library, which talks to Radix's archive API via JSON-RPC. We began with version 1.0-beta.35-SNAPSHOT and proceeded through several development builds as the API evolved.

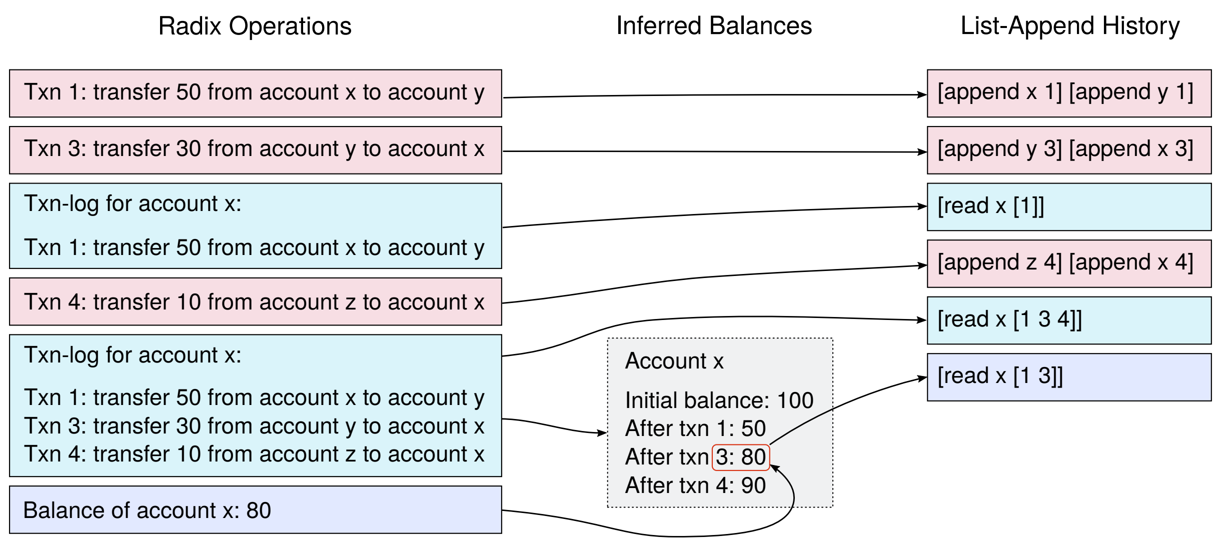
Our principal workload submitted randomly generated transactions which transferred XRD from a single account to 1-4 others. Accounts were selected from an exponential distribution, and generally limited to 64 transactions per account to limit the quadratic cost of reading and verifying long transaction logs. Meanwhile, we issued single-account balance and transaction-log reads across that same pool of accounts.

Radix transactions frequently timed out, remaining in state `PENDING` after 10 seconds of polling. Because indeterminate transactions reduce the accuracy and increase the cost of our analyses, we attempted to resolve these whenever possible. We maintained a cyclic buffer of all pending transactions, and periodically checked transactions from that buffer to see if they'd resolved to a `CONFIRMED` or `FAILED` state.

2.1 Transaction Ordering

Instead of designing a new dedicated safety checker for Radix's data model, we translated Radix transactions, balance reads, and transaction-log reads into histories Jepsen can already check: transactions made up of reads and appends to lists, where each list is identified by a unique key. To do this, we interpret Radix accounts as lists of *transaction IDs*, which are uniquely generated by Jepsen and stored in each transaction's `message` field.

We rewrite each Radix transaction T_i to an abstract *list-append transaction* comprising a series of operations which appended T_i to every account T_i touches. We rewrite each read of an account's transaction log to a transaction which performed a single read of that account ID, returning the list of transaction IDs extracted from the `message` field of each transaction in the log. Finally, we take the longest transaction log for each account and play forward its sequence of transactions to derive a sequence of balances the account took on during the test. This allows us to map most (though not necessarily all) balance reads to a "virtual" read of some prefix of the transaction log.



Radix Operations
Inferred Balances
List-Append History
Txn 1: transfer 50 from account x to account y
Txn 3: transfer 30 from account y to account x
Txn-log for account x:
Txn 1: transfer 50 from account x to account y
Txn 4: transfer 10 from account z to account x
Txn-log for account x:
Txn 1: transfer 50 from account x to account y
Txn 3: transfer 30 from account y to account x
Txn 4: transfer 10 from account z to account x
Balance of account x: 80
Account x
Initial balance: 100
After txn 1: 50
After txn 3: 80
After txn 4: 90
[append x 1] [append y 1]
[append y 3] [append x 3]
[read x [1]]
[append z 4] [append x 4]
[read x [1 3 4]]
[read x [1 3]]

For example, in this diagram we take a Radix history involving three transactions numbered 1, 3, and 4. Transaction 1 transfers 50 XRD from account x to account y, transaction 3 transfers 30 XRD from y to x, and transaction 4 transfers 10 XRD from z to x. An early read of x's transaction log shows transaction 1, and a second read shows transactions 1, 3, and 4. Finally, a read of x's balance shows 80 XRD.

Transfers and transaction-log reads are directly translated to list-append transactions. Transaction 1 is rewritten as a list-append transaction which appends 1 (the transaction ID) to keys x and y (the two accounts involved). The transaction-log read of transactions 1, 3, and 4 becomes a list-append read of key x, returning the list `[1 3 4]`.

To analyze balance reads of account x, we take the longest transaction log for x and simulate the effects of applying each of those transactions to x in turn. Knowing the initial balance of x is 100 XRD, we obtain successive balances of 50, 80, and 90 XRD by applying transactions 1, 3, and 4.

We then examine the balance read of $x = 80$ XRD. Since 80 appears only once in the computed series of balances, we know that this balance read should have observed the state of x resulting from applying transaction 1, then 3. We translate this transaction to a list-append read of x returning `[1 3]`.

There were some additional subtleties here. Each transaction costs a fee which is destroyed as a part of execution—we recorded fees as a part of transaction submission and took them into account when computing balances. Because Radix histories with reads are only serializable rather than strict serializable, we could fail to observe some transactions which actually executed. Furthermore, not all balances may have been uniquely resolvable to specific transactions. However, these ambiguities did not prevent our inference from being sound—they only reduced completeness. In general, only a handful of unobserved or ambiguous transactions occur during a test.

With transactions encoded as appends and reads of lists, we use Elle to check that the resulting history is serializable. We additionally construct a graph of real-time dependency edges between non-concurrent transactions: if T_1 is confirmed before T_2 is submitted, T_1 must precede T_2 in the serialization order. We also compute dependencies between all non-concurrent operations on a single node: this allows us to check (for example) that two reads against node `n1` observe logically increasing states of the system. Elle then merges these dependency graphs together, along with inferred read-write, write-write, and write-read dependencies derived from transaction structure, and looks for cycles in the resulting graph. Each cycle corresponds to a consistency anomaly.

2.2 Additional Checks

Projecting transactions into list-append operations allows us to check for aborted reads, transaction ordering, etc. However, this projection focuses primarily on ordering, and mostly ignores the semantic meaning of transfers and account balances. We therefore complement our list-append checker with additional safety checks. For instance, we verify that transactions are faithfully represented in transaction logs: they have the same number of operations as the transactions which were submitted, they interact with the same accounts, transfer the same amounts, and so on. We compute the set of all possible balances for each account over time, and make sure that balance reads always observe some plausible amount. We check to make sure that account balances (both via balance reads and those implied by transaction logs) never become negative.

2.3 Raw Reads

To distinguish between issues in the underlying transaction log versus those in the per-account indices derived from that log, RDX Works team members added an API for querying the raw transaction log directly. Our tests integrated that API and verified that the raw transaction log was consistent with submitted transactions, exhibited a total order, etc. We also projected the raw transaction log into per-account logs, and used that information as a part of our transaction ordering inference.

We added similar support for a testing-only API which exposed the raw balances of accounts.

2.4 Faults

Throughout our tests we injected a variety of faults into our clusters, including process crashes, process pauses, network partitions, clock errors, and membership changes.

Membership changes were particularly tricky: the membership state machine is complex, highly asynchronous, and easy to get into "stuck" states where no transactions can proceed. For example, our original tactic for removing nodes simply killed the node and deleted its data files, as might happen if a validator node caught on fire and backups of its keys were not available, or the organization running it closed down shop unexpectedly. In Radix, validators are generally expected to politely remove themselves, then remain on the network until the end of the current epoch, or ensure that fewer than 1/3 of current validators (by stake) have also removed themselves: the concurrent loss of 1/3 of validators by stake causes Radix to halt. Although we attempted to preserve a 2/3 supermajority of active stake through each membership transition, our test harness struggled with liveness breaks when nodes were removed impolitely. To address this, we introduced new membership transitions for registering and unregistering validators, and had our nodes politely unregister themselves before shutting down.

In the Radix Olympia Public Network, three factors help ensure liveness. First, RDX Works encourages users to stake their XRD on validators with a small stake, to prevent a few nodes from holding 1/3 of all stake. Second, validators which do not participate in rounds do not receive XRD rewards, which provides incentive for stakers to redistribute their stake away from failed validators. Third, if a network suffers the loss of more than 1/3rd of validators by stake, it can be restarted through a manual process involving political coordination.

2.5 Stokenet

Due to differences in network size, latency, and workload, we suspected that issues identified in local test networks might not manifest in large-scale deployments of Radix. To that end, we adapted our test workload to run on the "Stokenet" public test network. While transaction fees limited the amount of testing we could perform, we were able to use this mechanism to reproduce key results, such as lost updates.

2.6 Mainnet

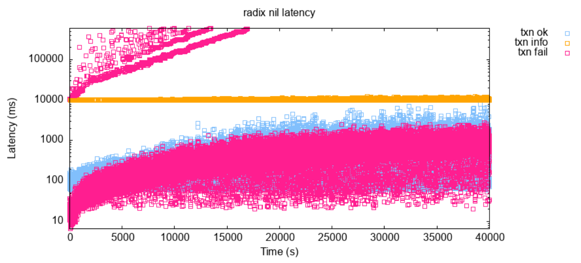
We also designed a passive checker which performed read-only queries against the Radix Olympia Public Network in order to look for traces of consistency anomalies. Our public-network checker started with a single validator account address, and used transaction log queries to traverse approximately six thousand reachable addresses and fifty thousand transactions.

We compared those transaction logs to search for cases where a transaction between accounts a and b was present in a's log, but not in b's log: a lost update. We also checked the status of each transaction to identify those which were in state `FAILED`, but which nonetheless appeared in transaction logs: aborted reads.

3 Results

3.1 Indeterminate Transactions During Normal Operation (#1)

As of summer 2021, RDX Works was aiming for a maximum transaction confirmation time of roughly five seconds, assuming the network was not overloaded. Transactions in our low-latency, five-to-ten node clusters generally took 100–1,000 ms to execute. However, even under healthy conditions a significant number of transactions took hundreds of seconds to definitively commit or fail. Consider this timeseries plot of transaction latencies during one 11-hour test run:



Even at only ~3 transactions per second, a significant number of transactions (the orange streak) timed out after 10 seconds. A few of those could eventually be resolved to a successful or failed state (see points above 10 seconds), but the time it took to resolve them increased exponentially over time.[3] Eventually timed-out transactions failed to resolve altogether—perhaps by falling out of cache.

High latencies are frustrating for users, but in theory tolerable so long as transactions eventually resolve to

[3]Our workload maintained a circular buffer of unresolved transactions and checked transactions from that buffer incrementally throughout the test. We expect observed latencies to rise linearly with the size of the buffer—and since a good fraction of transactions *never* resolve, the buffer size should increase semi-linearly over the course of the test. This likely accounts for some of the observed increase in failed latencies above 10,000 ms, but does not explain the exponential curve. Perhaps other factors are at play.

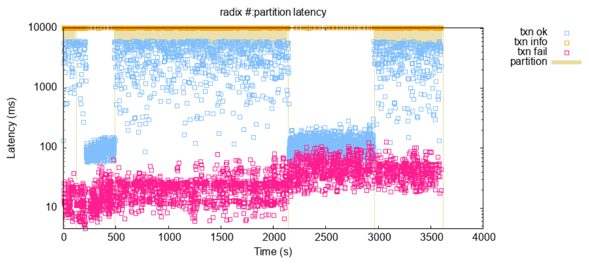
a definitive state: e.g. CONFIRMED or FAILED. However, at roughly five transactions per second, approximately 5–10% of submitted transactions never resolved.

This poses a hazard for Radix users: when one submits a transaction, there is a good chance that one simply will never know whether it took place or not. Users cannot assume it committed without running the risk that it actually failed (and presumably, their payment never going through). They also cannot assume the transaction failed and resubmit their transfer request: if the transaction *did* commit, they could pay twice as much as intended. Clients could theoretically save the computed raw transaction to resubmit it on the user's behalf: using the same UTXO states should prevent double-spending. However, this assumes that the client is smart enough to save those raw transaction states, detect user retries, reliably differentiate them from intentional submission of duplicate transactions, and resubmit the saved transaction when desired.

After our work together, RDX Works replaced the archive API with a new Core API and Network Gateway. RDX Works asserts that this issue is resolved as of version 1.1.0. Since this system was developed after our testing, Jepsen has not evaluated it.

3.2 High Latencies During Single Faults (#2)

Another potentially surprising finding: when even a single node is unavailable due to a crash, pause, or network partition, median transaction latencies on the remaining nodes remain dramatically elevated for the duration of the fault. In our tests, a single fault in a five-node cluster with the default tuning options caused transaction latencies to rise by roughly 1.5 orders of magnitude.



The above plot shows the latency distribution of transactions over time from a test where we submitted roughly five transactions per second, while periodically isolating a single node via a network partition. Partitions are represented by shaded tan bars from 2 to 117 seconds, from 485 to 2,135 seconds, and 2,957 to 3,612 seconds. During each of these intervals, median transaction latency jumped from ~90 ms to 5–10 seconds. A good number of operations timed out after 10 seconds: Radix does this normally, but more operations time out during faults.

RDX Works believes this behavior is a consequence of Radix's consensus design. Consensus proceeds in rounds, and each round is coordinated by a single validator: the *leader*. Validators take turns being the leader, proportional to their stake. When a validator is down, the consensus rounds which that validator should have led will fail, blocking transactions from committing until the next round led by a healthy validator begins. There is presently no mechanism for detecting faulty nodes and skipping them: every time a faulty node takes a turn as the leader, the network must wait for that node to time out before proceeding to the next round.

One might expect that with five evenly-staked validators a single-node fault would cause only ~20% of transactions to experience high latencies. Instead, it appeared that a single-node fault affected almost every transaction. This could be because healthy leaders complete their rounds in a handful of milliseconds, but a faulty leader blocks consensus for several seconds before the cluster moves on to the next leader. If requests arrive uniformly over time, almost all requests will arrive during the faulty leader's round, and must wait for that round to complete before a healthy leader can process them. A single faulty node can therefore affect almost all requests!

We suspect that two factors mitigate this issue in the Radix Olympia Public Network today. First, a large pool of validators (e.g. 100 rather than 5) increases the number of rounds led by healthy nodes. Second, higher inter-node latencies increase the time it takes for each healthy round of consensus, which means a smaller fraction of requests arrive during the unhealthy round. By contrast, our test environment featured low latencies and a small pool of validators, both of which amplify the effects of single-node faults.

RDX Works also has ideas for improving latency in the future. The present leader timeout is set to a relatively conservative three seconds. In our tests, lowering this timeout to 300 milliseconds cuts the upper bound on transaction latency from 10 seconds to roughly 1 second. RDX Works may be able to reduce this time-out through an update to the Radix Olympia consensus protocol, to reduce the duration of consensus rounds led by an unavailable validator. RDX Works also reports unfinished designs for mechanisms to reduce the number of proposals a seemingly faulty leader is called upon to make by allowing a validator which is timing out to be gradually reduced to zero participation, regardless of stake. Validators could fully rejoin consensus once in good health.

3.3 Non-Monotonic Reads (#3)

Under normal operation, the transaction history of an account can fail to include committed transactions—even when those transactions are already known to be committed by that node! For instance, consider this test run in which Jepsen process #9, talking to node n5, submitted and confirmed transaction 4 (t4), which transferred 99 XRD from account 4 to account 3.

```
{:process 9
 :type    :invoke
 :f       :txn
 :value   {:id 4
           :from 4
```

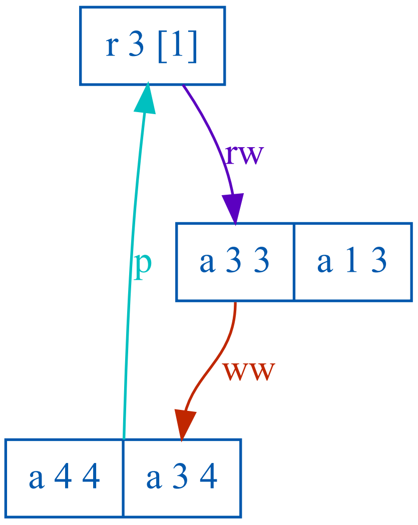
```
          :ops [[:transfer 4 3 99]]}
 :time    424827567
 :index   15}
{:process 9
 :type    :ok
 :f       :txn
 :value   {:id 4
          :from 4
          :ops [[:transfer 4 3 99]]}
 :time    542953296
 :index   23
```

0.032 seconds after that transaction was known to be committed on n5, process 9 initiated a read of account 4's transaction history. That read returned two transactions:

```
{:f       :txn-log
 :value   {:account 3}
 :time    575230138
 :process 9
 :type    :invoke
 :index   27}
{:f       :txn-log
 :value  {:account 3
          :txns
          [{:fee     0N
            :message nil
            :actions [...]}
           {:fee     100000000000000000N
            :message "t1"
            :actions
            [{:type      :transfer
              :to        "brx...wq5"
              :rri       "xrd_rb1qya85pwq"
              :validator nil
              :from      "brx...ahh"
              :amount    68N}]}]}
 :time    592084345
 :process 9
 :type    :ok
 :index   30}
```

The first transaction affecting account 3 was an initial setup transaction and not a part of our test workload. The second transaction was a Jepsen-initiated transfer transaction labelled t1. So node n5 knew that t4 was committed, and that t4 affected account 3, but also failed to show t4 in account 3's history! This is a stale read—not only from the perspective of the cluster as a whole, but also as viewed by n5 alone. We call this a *non-monotonic read* because successive reads performed against a single node may appear to go "backwards in time": observing then un-observing the effects of a transaction.[4]

Our Elle-based checker renders this anomaly as a cycle involving three operations. The top operation is the read of account 3 observing only t1. The middle operation was t3, which transferred funds from account 3 to account 1. Like t4, it must have logically occurred after the top read, because the top read did not observe t3: we render this as a read-write dependency labeled rw. The bottom operation is t4, which transferred funds from account 4 to account 3. We know that t4 executed after t3 thanks to a later read not shown here: since t4 overwrote t3, there is a write-write (ww) dependency between them. Finally, since the top and bottom transactions took place on the same node in strict order, there is a per-process edge (p) between them.



In our testing of 1.0-beta.35.1, 1.0.0, 1.0.1, and 1.0.2, non-monotonic reads occurred frequently in healthy clusters, but were generally no more than a quarter-second out of date.

As of version 655dad3, balance and transaction-log reads on single nodes appeared (mostly) monotonic in our tests. In version 1.1.0, RDX Works asserts that the new Network Gateway in 1.1.0 does not exhibit this behavior.

3.4 Missing & Extra Actions in Transaction Logs (#4)

Radix's archive transaction logs may not faithfully represent the transactions which are submitted. As of 1.0.0 (but not in 1.0-beta.35.1) the transaction log always eliminates transfers from an account to itself. Moreover, Radix archive nodes would occasionally insert actions into transactions with type UNKNOWN and no values for the from, to, validator, rri, or amount field. For example, consider this transaction as it was submitted to Radix:

```
{:message "t53265"
 :actions [{:type :transfer
            :from 2902
            :to 2901
            :amount 300000000000000000N
            :rri "xrd_dr1qyrs8qwl"}
           {:type :transfer
            :from 2902
```

[4]Technically this example is a violation of read-your-writes, if one interprets the transaction as a write, rather than a read. Even more technically it *is* a violation of monotonic read, since the "transaction" operation actually had two parts: submitting the transaction, then reading to see if it was confirmed.

```
            :to 2901
            :amount 8400000000000000000N
            :rri "xrd_dr1qyrs8qwl"}]}
```

… versus that same transaction's representation in the archive transaction log:

```
{:fee 1772000000000000000
 :message "t53265"
 :actions [{:amount 3000000000000000000
            :validator nil
            :type :transfer
            :rri "xrd_dr1qyrs8qwl"
            :from 2902
            :to 2901}
           {:amount 8400000000000000000
            :validator nil
            :type :transfer
            :rri "xrd_dr1qyrs8qwl"
            :from 2902
            :to 2901}
           {:amount nil
            :validator nil
            :type :unknown
            :rri nil
            :from nil
            :to nil}]}
```

Transaction 53265 somehow gained an extra `unknown` action. This behavior occurred in healthy clusters running version 1.0.0, but was (initially) relatively infrequent. In recent development builds, we observed these anomalies in up to ~50% of submitted transactions.

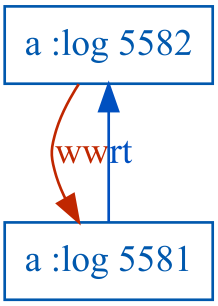
Both the omission of self-transfers and the insertion of spurious `unknown` actions seem like relatively minor problems: self-transfers don't (by definition) affect the overall balance of an account, and the `unknown` actions don't either. However, this could be surprising for Radix users who expected to see the transactions they originally submitted.

The omission of self-transfers is a consequence of how Radix's archive subsystem interpreted unspent transaction outputs: the encoding of self-transfers in Radix's ledger makes them indistinguishable from "getting change back" from another transfer. Radix is unsure why the archive API inserts spurious unknown actions into transaction logs.

These questions are largely moot: RDX Works has removed the archive subsystem altogether in version 1.1.0. The Network Gateway now infers the structure of actions in transaction logs from Radix's ledger.

3.5 Premature Commits in Development Builds (#5)

The unreleased development build 48461c4 dramatically improved Radix latencies—but also exhibited a new class of anomaly: the write order of transactions could be contrary to the real-time order in which they were submitted. These issues manifested within minutes in healthy clusters, and were exemplified by cycles like the following:



Here transaction 5581 was confirmed 9 milliseconds before transaction 5582 was even submitted: we have a real-time (`rt`) edge from 5581 to 5582. However, later reads of the raw transaction log returned values like `(... 5577 5579 5582 5581)`, which tells us that 5582 committed before 5581: a write-write (`ww`) dependency. Transactions in Radix were supposed to be strict serializable, which means they should appear to take effect in an order consistent with the real-time order in which they were submitted. However, transaction 5582 took effect before 5581, even though 5581 was (apparently) committed before 5582 even began!

We also frequently observed cases in version 48461c4 where two transactions which conflicted with one another would both appear to commit. This allowed a form of double-spend attack: Alexis could pay Brendan $1,000 for a laptop using Radix, and both Alexis and Brendan would see that payment transaction as committed in Radix.[5] Brendan would then ship Alexis the laptop. Contemporaneously, Alexis could stash their entire (pre-payment) account balance in another account they controlled. If the stash transaction committed, Alexis would retain all their money *and* get a free laptop.

We wrote a test workload to perform this attack and found that frequently, both spend and stash transactions would be successfully confirmed—even though at most one could have possibly succeeded. Spend and stash transactions could be conducted concurrently, or the stash could be executed strictly after the payment was confirmed, or vice-versa. All approaches allowed double-spend, but the most successful approach was to submit the stash transaction first—doing so allowed roughly 8% of payments to appear to succeed, but for Alexis to actually keep their money.

Both of these problems stemmed from a bug in the transaction status service which returned `CONFIRMED` for pending transactions, not just confirmed transactions. This allowed transactions to be reported as confirmed before their actual commit time (causing G0-real-time) and for transactions to be reported as

[5]RDX Works clarified that while this phenomenon involves a user spending the same coins in two different transactions, and both transactions appearing to succeed, it does not satisfy their definition of a double-spend. RDX Works would prefer to discuss double-spend only in the context of internal ledger state, rather than the consensus system made up of Radix nodes and clients together.

confirmed when they would later go on to fail. That bug was patched in 350ac77, which appears to have resolved the issue and returned commit latencies to their previous range. Version 48461c4 never made it into a release, and users should not have been affected.

3.6 Committed Transactions Could Have Status `FAILED` (#6)

Under normal operation without faults, transactions with status `FAILED` could actually be committed. In this test run with no faults, a cluster of transactions were reported as `FAILED` but were actually visible in the transaction log. Here is transaction 10750, which had status `FAILED`…

```
{:time    927331078170
 :process 886
 :type    :fail
 :f       :txn
 :value   {:id     10750
           :from   570
           :ops    [...]
           :txn-id "c33...6d9"
           :fee    1626000000000000000N}
 :index   54689}
```

… but also appeared in subsequent reads of the transaction log!

```
{:time 928355977217
 :process 12
 :type :ok
 :f :txn-log
 :value
 {:account 570
  :txns
  [...
   {:fee 1626000000000000000N
    :message "t10750"
    :actions [...]}]}
 :index 54743}}
```

This applied both to the archive API and the raw transaction logs. Version 1.0.0 was affected; 1.0.1 and 1.0.2 were likely susceptible as well. Since failed transactions were visible to reads, this anomaly is akin to phenomenon G1a: aborted read.

This issue was reproducible in our test clusters with as few as 1.5 transactions per second. It also occurred in the Radix Olympia Public Network. For example, transaction 50c46b1, submitted on October 1 2021, appeared in the transaction logs for both involved accounts. However, its transaction state on October 1 (read after observing that transaction in account logs) was `FAILED`. Two days later, on October 3rd, its state flipped to `CONFIRMED`.

On October 4th we ran our public-network checker again and found *four* apparently-failed-but-actually-committed transactions (6ebb247, 8d05488, ab8a78a, and 5428543) submitted in a ten-minute window; all had status `FAILED` roughly an hour after submission, but flipped to `CONFIRMED` shortly thereafter. A fifth committed transaction (e563bad) persisted in state `FAILED` for several hours.

RDX Works suspects this issue occurred when a transaction committed normally but had been gossipped to other nodes' mempools; if those nodes then gossiped the transaction *back* to the original node, that node would recognize that the transaction had already committed and reject the gossip message. Concluding the transaction was rejected, Radix would then overwrite the transaction's status to flag it as `FAILED`. When the transaction later fell out of cache, subsequent reads would query the log directly, and observe its state as `CONFIRMED`.

This issue was initially resolved in 48461c4. RDX Works asserts it does not appear in version 1.1.0's Network Gateway service either.

3.7 Missing Transactions (#7)

Under normal operation, committed transactions may fail to appear in transaction logs. A validator will insist that the transaction is confirmed, and the balances of involved accounts will change, but some (but not necessarily all!) of those accounts' transaction logs will never contain the transaction. For example, consider this test run, in which transaction-log reads of account 4 all began with the following:[6]

```
{:account 4
 :txns
 [...
  {:message "t0"
   :actions
   [{:type :transfer
     :from 1, :to 4, :amount 46N}]}
  {:message "t8"
   :actions
   [{:type :transfer
     :from 1, :to 4, :amount 36N}]}
  {:message "t10"
   :actions
   [{:type :transfer
     :from 4, :to 1, :amount 1N}
    {:type :transfer
     :from 4, :to 5, :amount 28N}]}
  {:message "t12"
   :actions
   [{:type :transfer
     :from 4, :to 4, :amount 85N}
    {:type :transfer
     :from 4, :to 4, :amount 7N}]}
  {:message "t14"
   :actions
   [{:type :transfer
     :from 5, :to 4, :amount 43N}]}
 ...]}
```

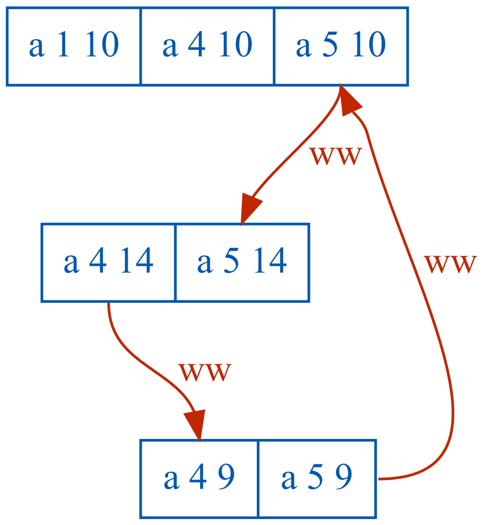
Meanwhile, transaction logs of account 5 all began with:

[6]We omit the initial setup transaction, fees, validators, and RRIs (token names) in the interest of brevity.

```
{:account 5
 :txns
 [{:message "t4"
   :actions
   [{:type :transfer
     :from 3, :to 5, :amount 28N}]}
  {:message "t7"
   :actions
   [{:type :transfer
     :from 2, :to 5, :amount 81N}
    {:type :transfer
     :from 2, :to 5, :amount 94N}]}
  {:message "t9"
   :actions
   [{:type :transfer
     :from 5, :to 4, :amount 42N}]}
  {:message "t10"
   :actions
   [{:type :transfer
     :from 4, :to 1, :amount 1N}
    {:type :transfer
     :from 4, :to 5, :amount 28N}]}
  {:message "t14"
   :actions
   [{:type :transfer
     :from 5, :to 4, :amount 43N}]}]}
```

The problem here is not immediately apparent—but on closer inspection transaction 9 (`t9`), which transferred 42 XRD from account 5 to 4, was present in account 5's log but missing from account 4. Account 4 skips directly from `t8` to `t10`!

If we take these transaction logs at face value, then we must conclude that on account 4 `t9` must have executed after `t14`. It certainly can't have executed before, or it would have appeared in the log. On account 5, `t9` executes directly before `t10`. In list-append terms, we have the following cycle:



From account 5 we know `t9` (bottom) executed on account 5 before `t10` (top), and it was followed by `t14` (middle). However, on account 4, `t14` must have executed before `t9`: a cycle. Since all of the edges in this cycle are write-write dependencies, this anomaly is called G0, or *write cycle*, and it implies this history violates read-uncommitted. It is also therefore not serializable.

In this particular example `t9` appeared to have committed. The transaction status API claimed that `t9` committed, and account 4's balance increased by 42 XRD during `t9`'s window of execution. It could be that this issue is limited only to transaction-log reads, and the internal transactions themselves are still strict serializable.

However, even a read-only omission of a transaction has serious consequences. The balance of a Radix account, as visible to users, might not be the sum of its recorded transactions: it is possible to gain or lose tokens and not be able to explain why. Two account histories can disagree on whether a transaction took place. From an accounting perspective, this is a violation of double-ledger bookkeeping principles. Balances can also (at least according to the transaction log) become negative—withdrawing more money from an account than the account ever contained.

This problem occurred in healthy clusters running version 1.0-beta.35.1, 1.0.0, 1.0.1, 1.0.2, and numerous development builds. It affected all nodes equally—when a transaction disappeared, clients could not recover it by reading from another node. It occurred even at low throughputs: at just 0.125 transactions per second, we were able to observe dozens of "confirmed" transactions which failed to appear in some or all transaction logs. We were also able to reproduce this issue in Stokenet, Radix's public test network: even at rates as low as 1 transaction per second, 5–10% of transactions vanished from some (but not necessarily all!) account transaction logs.

At least one transaction has already gone missing from account histories in the Radix Olympia Public Network. For instance, transaction 63b8485… transferred 0.9 XRD from rdx…q96rxfx to rdx…ctcgge2. As of October 1, 2021, that transaction appeared in `suq96rxfx`'s transaction log, but did *not* appear in `ctcgge2`'s log.

These issues were addressed by replacing the transaction log archive system in version 655dad3. RDX Works asserts this issue does not appear in the Network Gateway, as of version 1.1.0.

3.8 Contradictory Logs (#8)

Transaction log anomalies were not limited to simple omissions. Two reads of an account's transaction log executed against a single node could contradict one another. For example, take this test run where process 39, making a series of transaction-log requests to node `n5`, observed the following logs for account 27:

Process	Time (s)	Transaction IDs
39	289.03	()
39	292.85	()
39	293.82	()
39	296.54	(5310)
39	297.29	(5310)
39	297.51	(5310)
39	297.63	(5310)
39	298.55	(5310)
39	298.83	(5310)
39	299.15	(5310)
39	300.41	(5310)
39	300.62	(5310, *5336*)
39	301.64	(5310, 5334)
39	302.69	(5310, 5334)
39	305.30	(5310, 5334)
39	307.58	(5310, 5334, 5345)
39	307.97	(5310, 5334, 5345)

Transaction 5336 was briefly visible in the transaction log, then replaced by transaction 5334. This should be impossible: if transactions are only appended to the log for a given account, they should never disappear; nor should two views of the transaction log disagree about the transaction at a particular index. This is worse than simply omitting a transaction from the log!

These errors occurred in 1.0-beta.35.1 and 1.0.0, in healthy clusters under normal operation. They likely affected 1.0.1 and 1.0.2 as well. However, they appeared infrequently: roughly one in 20,000 transaction log requests. They also seemed to be transient—a transaction would appear for a single read, then immediately disappear and never be seen again. They appeared not only in the archive API's view of a single account's transactions, but also in the raw transaction log.

We don't know what caused this issue, but it was no longer reproducible as of version 655dad3. It's possible that raising the BerkeleyDB safety level (which also allowed transaction loss) resolved this problem in raw logs, and replacing the archive subsystem for transaction logs addressed its occurrence in per-account logs.

After our work together, RDX Works completely replaced the archive node system with their new Core API/Network Gateway architecture. Transaction logs no longer exist in 1.1.0; RDX Works believes this issue no longer applies.

3.9 Split-Brain Transaction Loss (#9)

In version 1.0.0, process crashes could lead some (but not all!) nodes to lose a committed transaction from the history for an account. Queries for transaction history on that node would omit that transaction, but queries against other nodes would reflect it. This state would persist indefinitely. For example consider this test run, in which reads of account 16, performed on various nodes, observed the following transaction logs:

Node	Time	Transaction IDs
n1	72	(264)
n3	74	(264)
…		
n4	96	(264)
n5	98	(264, 267)
…		
n5	127	(264, 267, 474)
n4	127	(264, 267, 474)
n5	128	(264, 267, 474)
n2	134	(264, 267, 474)
n1	134	(264)
n1	138	(264, *474*)
n4	138	(264, 267, 474)
n1	139	(264, *474*)
n5	141	(264, 267, 474)
…		
n4	333	(264, 267, 474, 812, 831, 1022, 1075)
n1	335	(264, *474*, 812, 831, 1022, 1075)
n5	339	(264, 267, 474, 812, 831, 1022, 1075)
n3	340	(264, 267, 474, 812, 831, 1022, 1075)
…		
n1	592	(264, *474*, 812, 831, 1022, 1075, …)

Transaction 267 was submitted at 64 seconds, confirmed as committed by 98.95 seconds, and visible in transaction histories on nodes `n2`, `n3`, `n4`, and `n5` beginning at 98 seconds. However, node `n1` (which was killed 114 seconds into the test) never observed transaction 267—despite recording additional transactions over the next several hundred seconds.

We never identified the cause of this issue. It was initially resolved in version 655dad3 by completely rewriting the archive transaction log subsystem. RDX Works asserts that this issue is also resolved in the new Core API/Gateway design.

3.10 Raw Log Write Loss On Crash (#10)

During process crashes, Radix 1.0.0's raw transaction log could lose committed transactions—even those which were universally agreed upon. For example take this test, where transaction 4643 committed between 354 and 359 seconds into the test. At 362 seconds, Jepsen killed every node in the cluster. Reads of the raw transaction log returned the following lists of transactions:

Node	Time	Transaction IDs
n3	359	(3766, 3767, 4643)
n4	359	(3766, 3767, 4643)
n2	359	(3766, 3767, 4643, 4640, 4647)
n2	359	(3766, 3767, 4643, 4640, 4647)
n1	359	(3766, 3767, 4643, 4640, 4647)
n5	359	(3766, 3767, 4643, 4640, 4647)
n3	359	(3766, 3767, 4643, 4640, 4647, 4648)
n2	360	(3766, 3767, 4643, 4640, 4647, 4648)
n2	360	(3766, 3767, 4643, 4640, 4647, 4648)

n5	369	(3766, 3767)
…	…	…
n5	375	(3766, 3767, 4894, 4904, 4905)

Despite transaction `t4643` being visible on every node, `t4643` (along with `t4640`, `t4647`, and `t4648`) was lost after Jepsen killed the entire cluster. `t4643` never reappeared, and Radix continued as if it had never happened.

This problem occurred only in cases where every node was killed at roughly the same time. This suggested a problem with disk persistence: perhaps validators did not actually write transactions to disk before considering them acknowledged. Indeed, RDX Works had chosen `COMMIT_NO_SYNC` when configuring the ledger's underlying BerkeleyDB storage system. Changing the durability level to `COMMIT_SYNC` appears to have addressed the issue. This fix was first available in version 1.1.0.

3.11 Intermediate Balance Reads (#11)

Under normal operation, Radix regularly returned balances for accounts which did not correspond to any point in the transaction log—or indeed, to *any* combination of possible transactions. For example, consider account 52 from this test run, whose first two transactions were:[7]

```
{:fee 1,
 :actions ({:amount 75, :type :transfer,
            :from 51, :to 52}
           {:amount 29, :type :transfer,
            :from 51, :to 50}
           {:amount 67, :type :transfer,
            :from 51, :to 52}),
 :id 717,
 :balance 0,
 :balance' 142}
{:fee 1,
 :actions ({:amount 26, :type :transfer,
            :from 51, :to 51}
           {:amount 66, :type :transfer,
            :from 51, :to 52}
           {:amount 69, :type :transfer,
            :from 51, :to 52}
           {:amount 48, :type :transfer,
            :from 51, :to 51}),
 :id 720,
 :balance 142,
 :balance' 277}
```

No other transactions were concurrent during this time, so the only possible values account 52 could have taken on were 0 (the initial state), 142 (after transaction 717), and 277 (after transaction 720). Yet after transaction 720, a read of account 52 observed a balance of 208!

A close look at `t720` reveals the problem. It began with an inferred balance of 142: the result of applying `t717`. It then transferred 66 XRD from account 51 to account 52, which would have resulted in a balance of 208—before moving on to transfer another 69 XRD to 52, resulting in a final balance of 277. It appears that this balance read observed a value from *partway through* `t720`: an intermediate read. This implies Radix was not actually read committed.

This behavior occurred regularly in healthy clusters. At just ten transactions per second, roughly one in three hundred reads observed an intermediate state. They were present in 1.0-beta.35.1 as well as 1.0.0. RDX Works initially fixed this issue in fb1bc43 by rewriting the account info storage service. RDX Works asserts that this issue is also resolved in the new Core API/Gateway design.

3.12 More Committed Transactions With Status `FAILED` (#12)

Version 655dad3 addressed many of the most frequent problems with transaction logs. However it still exhibited aborted reads in which transactions could (very rarely) have status `FAILED` but appear in transaction logs. For instance, consider this test run, in which transaction 4743 was submitted to node n3 at 296.27 seconds, checked on n5 at 330.65 seconds, and found to be `FAILED`. However, every subsequent read of account 28 included transaction 4743, beginning at 331.53 seconds.

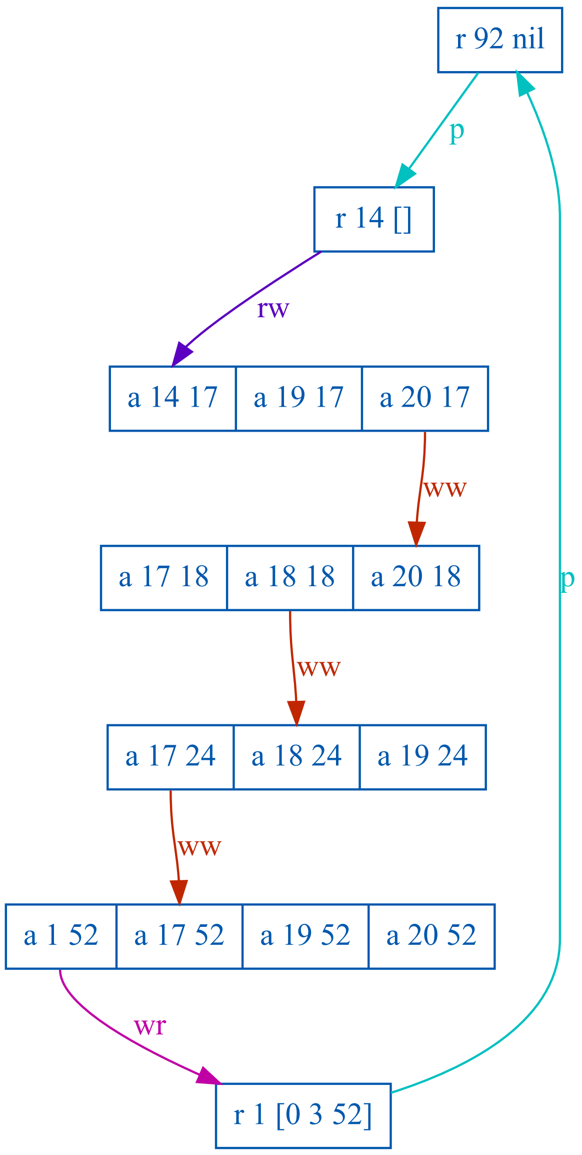
This problem appeared to be much rarer than previous aborted reads: we observed it only four times in ~20 hours of testing. Thus far it has appeared only in tests which included network partitions, or with combined process crashes and membership changes.

RDX Works did not report a cause for this issue. However, they assert the new Core API/Network Gateway architecture in version 1.1.0 resolves it.

3.13 More Non-Monotonic Reads (#13)

In tests of version 655dad3 with membership changes and process crashes, a pair of reads performed sequentially against a single recently-joined Radix node could observe a later state *before* an earlier state. For instance, this test run contained the following cycle:

[7]To make this easier to read, all amounts are divided by 10^{17}.



The bottom-most transaction in this cycle was a balance read of account 1, which observed a balance that could only have resulted from applying transactions 0, 3, and 52. At the top of the cycle, two subsequent transactions performed on the same node observed no balance at all for account 92, and an empty transaction log for account 14. However, that empty state of account 14 must have *preceded* transaction 52 on account 1, through a chain of read-write (rw) and write-write (ww) dependencies. This implies that the state of this single node "went backwards" relative to the transaction-log order.

RDX Works did not report a cause for this issue. However, they assert that it does not appear in their new Core API/Network Gateway architecture, in version 1.1.0.

№	Summary	Event Required	Reported Fixed
1	Indeterminate transactions during normal operation	None	1.1.0
2	High latencies during single faults	Single crash, partition, etc.	Unresolved
3	Non-monotonic reads	None	1.1.0
4	Missing & extra actions in transaction logs	None	1.1.0
5	Premature commits in development builds	None	350ac77
6	Committed transactions have status `FAILED`	None	1.1.0
7	Missing transactions in transaction logs	None	1.1.0
8	Contradictory transaction logs	None	1.1.0
9	Split-brain transaction loss	Single-node crash	1.1.0
10	Loss of committed transactions from raw log	All nodes crash	1.1.0
11	Intermediate balance reads	None	1.1.0
12	More committed transactions with status `FAILED`	Network partitions	1.1.0
13	More non-monotonic reads	Membership changes and crashes	1.1.0

4 Discussion

RDX Works intended to offer a distributed ledger with strict-serializable transactions and per-node monotonicity. However, under normal operation, our Radix test clusters exhibited stale reads of balances and transaction logs, intermediate reads of balances, transient and long-lasting loss of transactions from account histories, and aborted reads where failed transactions could actually commit. We observed transaction loss and aborted reads in the Radix Olympia Public Network. Transaction logs failed to faithfully represent submitted transactions by omitting some actions and spuriously inserting `UNKNOWN` actions. Nodes could lose transactions and enter permanent split-brain in response to process crashes.

Many of these issues stemmed from the archive API's index structures, which failed to properly track the underlying ledger. However, Radix could also lose transactions from the raw ledger itself when all nodes crashed concurrently, due to an inappropriate choice of `COMMIT_NO_SYNC` as the safety level for the underlying BerkeleyDB storage system.[8]

In addition, we found significant liveness and performance issues. Transaction throughput peaked at ~16 transactions per second in five- to ten-node clusters with near-zero latency. A significant fraction of transactions in our testing took tens or even hundreds of seconds to resolve to `CONFIRMED` or `FAILED`; far more never resolved at all. Single-node faults caused significantly elevated latencies for almost all transactions, though this behavior may be specific to our low-latency, 5–10-node test clusters.

In response to these issues, RDX Works opted to replace their archive node system with a different architecture. The updated architecture involves a split between a low-level event stream exposed by one or more Radix nodes (the *Core API*), and a new network service which consumes and aggregates Core API information, and exposes it to clients via HTTP (the *Gateway API*).

The Core API was released in Radix 1.1.0, on January 17th, 2022. The Gateway API was released with Network Gateway 1.0.0 on January 20th, 2022. A Radix Wallet which integrated the Gateway API was released at version 1.3.0 on January 27th, 2022, and an updated version of the Radix Explorer web site was released on the same date.

RDX Works asserts that as of January 27th, 2022, all identified safety issues have been fixed, liveness issues with individual transactions have been fixed, and one issue relating to performance in networks with non-participating validators remains. They also report that with multiple test passes running hundreds of thousands of transactions across multiple test networks, RDX Works is no longer able to reproduce the resolved issues. Jepsen congratulates RDX Works on these advances.

Jepsen has verified none of RDX Works's claims since the end of our testing on November 5th, 2021.

4.1 Ordering

By design, Radix does not offer strict serializability: reads of the transaction log or balances do not go through consensus, but instead return whatever state the local node happens to have. This state could be arbitrarily stale. If a client issued queries to multiple nodes it could observe a transaction commit, then fail to see that transaction's effects; under normal operation we routinely observed stale reads. Since this is not documented, we believe it is worth stating explicitly. Users should be aware that reads from archive nodes may not reflect the most recent state of Radix. Confirmed transactions may not be visible depending on which validator one is talking to. RDX Works states that they do not believe this behavior is acceptable.

RDX Works reports the new Gateway API includes a ledger state object with all read responses. This ledger state includes the epoch, round number, and state version which the request observed. Clients can employ

[8]We report this fault as requiring all nodes to crash, but suspect it might actually be something like "more than 2/3 of validators by stake", since not all validators need participate in a consensus decision.

these numbers as a causality token to obtain monotonic views of the ledger, even across gateways and validators. This is not sufficient to prevent stale reads in general, but Jepsen (from a cursory discussion of the feature) suspects it might be sufficient for sequential consistency—and, by extension, read your writes.

4.2 Performance

As of July 2021, the Radix Olympia Public Mainnet targeted ~50 transactions per second on a globally distributed network of 100 validator nodes, with transactions confirmed on ledger generally within five seconds, so long as the network is not overloaded. In our testing with five to ten validators, transaction throughput rarely exceeded 16 transactions per second. Even request rates as low as 1 transaction per second resulted in strongly bimodal latencies: most transactions confirming or failing within ~100–1000 ms, and ~5–10% taking tens or even hundreds of seconds to resolve—or never resolving at all. We routinely observed transactions get "stuck" in a pending state for at least 11 hours.[9]

As of February 2nd, 2022, RDX Works assserts that the issue with some transactions taking abnormally long to process has been resolved (when the network is operating within capacity). Jepsen has not verified this assertion. Whenever possible, clients should cache the finalized transaction just prior to submission and save it for resubmission in the event that it does not resolve—as well as providing explicit workflows for retries. The alternative is to risk transactions never going through, or to potentially pay twice (or three times!) the required amount.

RDX Works states that they consistently observe sustained throughput of 40–50 transactions per second in globally distributed Olympia test networks, and believe that our lower observed throughput is a consequence of our testing methodology rather than the network being incapable of processing a greater throughput. In particular, RDX Works points to the fact that our test harness used Radix's Java client and HTTP APIs to construct and submit transactions from outside the network, and that these transactions might have higher contention.

RDX Works also reports they have conducted sustained stress tests using their new Core API/Network Gateway architecture, which replaced the archive API used in this test. With independent worker processes submitting transactions "far beyond network throughput capacity," RDX Works observed a maximum transaction latency—a far outlier—of 81 seconds.

Most classic fault-tolerant consensus systems can handle the failure of a minority of nodes without significant impact on latencies. Radix behaved differently: in our tests, a single node failure increased latencies for almost every transaction from ~100 ms up to 5–10 seconds. The fraction of affected requests and the magnitude of the latency increase likely depend on the number of validators and the inter-node latency. The Radix Olympia Public Network has 100 rather than 10 validators, which increases the probability that a single node is down at any time, but also increases the number of healthy consensus rounds that can occur during a single-node fault. The public network's higher inter-node latencies might also mask the impact of faults by slowing down healthy consensus rounds relative to the faulty validator timeout. Users may need to plan for latency spikes, but without long-running measurements of Radix public networks, we can't say for sure.

Jepsen typically tests systems capable of hundreds to tens of thousands of operations per second, with nominal latencies on the order of 1–100 milliseconds. Higher throughput and lower latencies generally make it possible to find more bugs: we have more chances for race conditions to occur, and finer-grained temporal resolution to identify timing anomalies. Radix's low throughput and high latency may have masked safety violations. In particular, our tests required several hours to reproduce e.g. aborted read (#13). Improving Radix performance may make it possible to identify and fix bugs faster.

4.3 Public Impact

We began testing Radix 1.0-beta.35.1 on June 15th, 2021. Jepsen reported significant safety issues to the RDX Works team, including missing transactions in transaction logs by June 28th, intermediate balance reads by July 6th, and contradictory transaction logs by July 9th. Aware that these issues occurred in healthy clusters, RDX Works chose to launch their public "Olympia" mainnet running version 1.0.0 on July 28th.

Since 1.0.0, 1.0.1, and 1.0.2 did not address any of the issues we identified, Radix users who made requests from July 28th, 2021 to January 27th, 2022 may have observed non-monotonic states on single nodes, intermediate balances, aborted reads, missing and spurious actions in transactions, inconsistency between account logs, transient or permanent loss of committed transactions, and long-lasting split-brain in which transaction logs disagree about the order of transactions. Transaction logs could appear to sum to negative balances. Transaction logs could disagree on whether a transaction happened or not. Clients could execute a transaction, see its state as `CONFIRMED`, observe it in account logs, then have it vanish later.

These issues were not merely theoretical: we were able to reproduce the omission of committed transactions from transaction logs on Stokenet, a Radix public test network, within seconds. Even at less than 1 transaction/sec and with no concurrency, roughly 5–10% of our transactions went missing from transaction logs. We also passively observed both transaction

[9]Again, we note that Jepsen primarily evaluates distributed systems safety: our workloads are designed to create high contention on at least some accounts. This pattern may not hold true in Radix's various public networks.

loss and aborted reads in the Radix Olympia Public Network.[10]

RDX Works reports that during the course of our testing, RDX Works undertook their own concerted testing efforts to attempt to reproduce the identified safety issues using the Radix Wallet, and were unable to do so. They were able to reproduce them when submitting transactions programmatically, at "the fastest possible speed".

RDX Works's position is that adversarial actions are extraordinarily common on public ledgers, both in the forms of technical and social engineering attacks. Given their inability to reproduce issues from within the Radix Wallet, they determined that the risk of harm to the end user was greater if any disclosure was made ahead of a fix being successfully implemented, tested, and deployed.

Bitfinex, the first exchange to integrate with the Radix Olympia Public Network, and the sole known candidate for expected high-rate usage, enabled the purchase and sale of XRD on August 23rd, 2021. RDX Works reports that Bitfinex was made aware of the aborted read issue prior to launch, and adjusted their use profile to avoid it.

Jepsen asked whether these issues might have affected transactions processed through the Instabridge Ethereum-Radix bridge. RDX Works relays that Metaverse Ltd examined Instabridge system records to search for cases where a submitted transaction appeared to fail but was later recorded as successful. No such cases were found. Instabridge does not automatically retry in the case of failure, so RDX Works believes there is no chance of such an event being "hidden" by a successful retry.

RDX Works asserts that the Network Gateway (which is now in use by the Radix Wallet and Explorer) provides an accurate view to clients of network state, including accurate transaction histories for all accounts which may have had incorrect information reported by the preceding archive node system.

Our collaboration concluded on November 5th, 2021. RDX Works declined to inform the public of these occurrences until the release of this report on February 5th, 2022.

4.4 Future Work

The RDX Works team plans to continue work towards their language for smart contracts (Scrypto) and sharded consensus implementation (Cerberus). Future testing could investigate Scrypto semantics and verify that Cerberus provides the same ordering guarantees as the current (non-sharded) HotStuff consensus system.

Our testing with membership changes was limited in scope: while our fault injection system added, registered, staked, unregistered, and removed nodes, the asynchronous nature of Radix's cluster view, the complexity of the membership state machine and the lack of guardrails within Radix to prevent (e.g.) unregistering *every* validator from the system meant that tests with membership changes tended to render the cluster unusable after a few dozen transitions—despite attempting to preserve a 2/3 majority of stake. While sufficient for basic testing, further work could improve the robustness of the membership fault scheduler.

The performance issues we identified in local testing suggest the need for ongoing monitoring of Radix Public Network latencies and transaction outcomes. In particular, it would be helpful to know how the failure of a single production Radix validator impacts user-facing latencies, the distribution of finality times, and what fraction of submitted transactions can be expected to get stuck in an indeterminate state indefinitely.

As typical for Jepsen reports, our work here focused on accidental faults: partitions, crashes, pauses, etc. Jepsen has not evaluated the robustness of Radix against malicious attackers. Future work might include writing intentionally malicious versions of the Radix validator and verifying that safety properties hold regardless.

4.5 Toward a Culture of Safety

These findings suggest important questions for the cryptocurrency, blockchain, and distributed ledger (DLT) community. What are distributed ledgers supposed to *do* in terms of safety properties? What do they actually do? Where should we measure them? And what do users expect, anyway? These questions (and Jepsen's suggestions) are hardly novel, but we present them in the hopes that they might prove a helpful jumping-off point for DLT engineers, marketers, and community members.

First, Jepsen believes distributed ledgers—like all distributed systems—might benefit from publishing more formal descriptions of their consistency semantics. For ledgers powered by consensus a range of powerful consistency models are possible. A DLT could opt for strict serializability, which ensures a global real-time order of all operations. Or serializability, which ensures only a total order regardless of realtime. There are concrete benefits to each: strict serializability ensures that users must immediately observe any confirmed transaction, whereas serializability allows much faster (but stale!) reads. Session-related models might also be applicable: strong session serializable would ensure each client observes a monotonically increasing state and never fails to observe their own prior transactions.

DLTs often reify a separation between write and read paths. In some systems, writes (DLT transactions) go through consensus and mutate the ledger, whereas reads are serviced by any node's local state and may

[10] Our ability to infer safety violations in the public network was limited by the fact that we had no visibility into what other Radix clients saw or what transactions were actually submitted to the network—we can't tell whether the other anomalies we observed in our local test clusters also occurred in the public network.

therefore observe any point in time. Reads might be made stronger through the use of a logical clock, allowing clients to enforce per-session or causal orders. Some reads could even be strict serializable: waiting on the underlying consensus system's incidental transaction flow to ensure recency. As in Zookeeper, Ordered Sequential Consistency might allow DLTs to describe the consistency semantics of interacting writes and reads.

When a DLT clearly defines its intended safety properties, we can investigate whether it satisfies those claims. A broad spectrum of software assurance techniques are available: from proofs to model checking to types to unit tests to end-to-end integration tests—like Jepsen. These techniques build confidence in different ways: a model checker for any nontrivial distributed system is rarely exhaustive, and an end-to-end test like Jepsen is even less likely to explore unusual corners of the state space. A solid proof, on the other hand, provides strong confidence in correctness—but it may not map perfectly to an implementation's behavior. For this we need tests.

Like any database, DLTs stand to benefit not only from example-based tests, but also by writing property-based tests which generate randomized inputs to explore paths hand-written tests might not have thought to travel. As distributed systems DLTs may find particular value in simulation or scheduler interposition techniques which explore novel orderings of events, and from fault injection, which deliberately causes network, node, and other failures to drive the system into atypical regimes. Since large-scale distributed systems experience faults frequently, this type of testing is an important part of creating safety.

As RDX Works aptly observes, DLTs are complex, multi-layered systems. In Radix, for example, a core consensus protocol running the ledger state machine was coupled to an ancillary index (the archive subsystem), which in turn provided an HTTP API for clients. Those clients, in turn, connected the ledger to other software systems and human beings.

Although a DLT specialist might not phrase it this way, each layer of this architecture is in fact a consensus system. The core ledger is often built explicitly as a consensus system: one where validators serve as proposers, acceptors, and learners. But by coming to eventual, nontrivial, single-valued agreement on the outcome of transactions, the archive API (in conjunction with the validators) is *also* a consensus system: the archive's internal data structures serve as additional learners. So too is the composition of clients, the archive, and validators: clients serve as both proposers and learners. In this consensus system clients ought to agree on whether or not transactions happened, what those transactions did, and what their order was. At the end of the day, human beings (and external software systems) want to use the ledger to help them agree.

When defining and measuring safety we should consider each of these layers. A core ledger system which violates consensus is, of course, likely to break consensus for clients as well. What is not so obvious is that a ledger can be perfectly correct and yet fail to provide consensus for clients—if, for example, its behavior is masked by a faulty intermediary. This is precisely the situation we observed throughout this report, and hints at why the scope of safety properties matters.

Jepsen typically explores the behavior of software deployed in a local testing cluster, rather than a public network. Following our discovery of anomalies in local clusters we designed a checker which passively explored the Radix Olympia Public Network by making HTTP read requests to public archive nodes. We began with a single account (taken from a public validator page) and traversed as many accounts and transaction logs as we could find from there, checking to make sure that transaction states and logs all lined up with one another. Since reads are free and open to everyone, this analysis was easy to perform—and it observed two bugs! Jepsen wonders whether other DLT teams or third-party evaluators might design their own crawlers to passively verify safety properties in public networks.

No non-trivial software is perfectly correct. There will always be bugs, and sometimes design shortcomings which impact safety. DLTs sit at the intersection of distributed systems, concurrency control, consensus, caching, and security. Systems which implement smart contracts must also tackle language, VM, and compiler design. These are challenging problems: it would be surprising if any system did not exhibit at least some safety violations.

This raises the question: to what extent do DLT users—from high-frequency automated exchanges to individuals—expect safety? Are they accustomed to aborted reads and forgiving of lost transactions? Or do they expect strict serializability at all times? If safety violations occur, what frequency are users willing to accept? These questions depend on workload throughput, latency, and concurrency demands, and are modulated by the probability and severity of anomalies. Still, it would be helpful to have some idea of what these expectations *are* for various use cases, so we can find out if systems live up to them.

On the other hand, users attempting to select a DLT for writing applications or for use as a financial network are confronted with a dizzying array of potential options. What safety guarantees does each ledger offer? Since vendors and users frequently have a financial or affiliative stake in these networks, there exist significant incentives to paint an optimistic picture of the technology. Marketing claims leverage ambiguity. Future plans are conflated with present behavior. Systems which have not undergone rigorous testing may operate under the presumption of safety. Moreover, even people with the best intentions may struggle to communicate safety invariants clearly. They're just plain *tough* to reason about—for end users, marketers, and engineers alike. All of this makes evaluating technologies more difficult.

If this sounds familiar, you're not alone. Roughly twelve years ago the rise of NoSQL accompanied an explosion of interest in distributed databases, queues, and other systems. Homegrown consensus algorithms which lost data during network partitions flourished, marketing claims soared to fantastic heights, and systems cut corners on safety to achieve better benchmark results. More than one system claimed to beat the CAP theorem, or asserted that network partitions or power failures fell outside their fault model.

As this cohort of the distributed database industry matured they developed a more nuanced and rigorous engineering culture. Their engineers learned through experience at scale and from the distributed systems literature. They expanded their fault models and adapted more robust algorithms. They more formally codified their safety guarantees, and began to explore formal models, simulation, and testing to gain confidence that those guarantees held. Users began to request stronger safety properties from their systems and developed a more nuanced view of performance, availability, and consistency tradeoffs. Marketing claims—while ever-optimistic—cooled somewhat. The distributed database community is far from perfect, but has made significant strides towards building, discussing, and evaluating safer systems. Jepsen is proud to have been a small part of that process, and looks forward to watching the DLT community build their own culture of safety.

4.6 Final Remarks

This report is provided for informational purposes only, and does not constitute financial advice. Neither Jepsen nor the author have any financial position involving XRD, eXRD, other Radix tokens, or shares in Radix Tokens (Jersey) Limited, RDX Works, or any other Radix-related entity.

As always, we note that Jepsen takes an experimental approach to safety verification: we can prove the presence of bugs, but not their absence. While we try hard to find problems, we cannot prove the correctness of any distributed system.

This work was funded by Radix Tokens (Jersey) Limited, and conducted in collaboration with RDX Works Ltd, in accordance with the Jepsen ethics policy. Jepsen wishes to thank the RDX Works team for their assistance—especially Russell Harvey, Matthew Hine, Joshua Primero, Shambu Pujar, Piers Ridyard, and Sergiy Yevtushenko. We would also like to thank Irene Kannyo for her editorial support during preparation of this manuscript.