Assessing Blockchain Protocols in Reputation-Based Networks

Sarayu Namineni (snaminen)

May 9, 2022

Abstract

Building on Local Trust (BoLT) is a reputation-based lending platform that uses a public ledger to create a transparent and universally accessible system. Our project aims to improve the performance of the BoLT ledger by integrating layer-1 and layer-2 blockchain protocols into our application. In this proposal, we assess the flexibility and performance of a highly-scalable layer-1 solution, Solana, in the context of the BoLT ecosystem.

1 Introduction

Solana is a high-performance layer-1 blockchain protocol which uses a hybrid of Proof-of-Stake (PoS) and Proof-of-History (PoH) to reach consensus. It boasts a throughput with a theoretical upper limit equivalent to centralized databases and in practice, its throughput is higher and its costs are lower than its main competitors [1]. The most popular blockchains, Bitcoin and Ethereum, can process 10 and 34 transactions per second, respectively. Newer blockchain technologies which rely on PoS consensus can reach higher throughputs; for instance, the foremost blockchain of proof-of-stake consensus,

Algorand, can handle up to 1,300 transactions per second. Solana surpasses all of these technologies, reaching speeds of up to 50,000 transactions per second (Figure 1) [2].

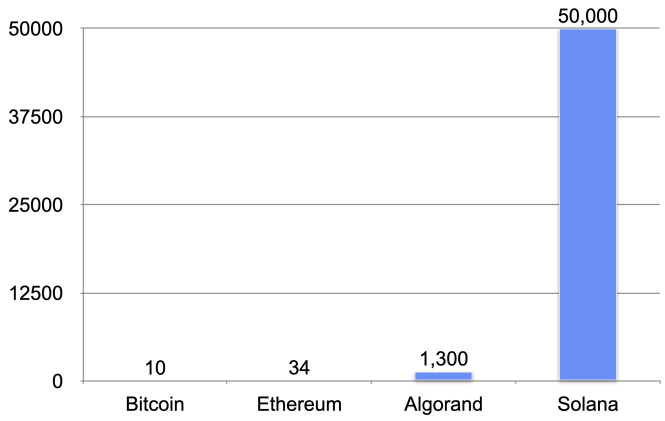

Figure 1: Throughput of mainstream blockchains

Solana's cost model also differs from its competitors in significant ways. Most blockchains have "gas fees," or costs for performing queries or transactions on the blockchain. For instance, Ethereum's gas fees range between $5 and $150. Instead of issuing a gas fee, Solana charges all accounts with a rent fee, or a cost associated with storing the data on the blockchain. Rent is collected for an account once per epoch as well as each time the account is referenced by a transaction. The rate at which it is charged rent is specified in genesis in lamports per byte per year. Instead of paying for rent, it is common for users to make their accounts "rent-exempt," or to pay for 2 years of rent at a time. Furthermore, Solana charges all transactions with a fee based on the number of signatures attached to it. The transaction fee is fixed at 5000 lamports per signature, where each lamport is a billionth of Solana's native cryptocurrency, SOL. Since the transaction fee is fixed for any application, Solana's cost model is largely proportional to the rent fee, or the

number of bytes of account data that are stored per smart contract [3].

In this project, we implement a prototype of the BoLT ledger on top of the Solana blockchain in order to assess the flexibility and performance of its smart contract language. The BoLT ledger presents a unique challenge as a multi-token system. Unlike a conventional blockchain which operates on a single token, all individuals on the BoLT network have the ability to define and mint their own tokens, or bolt specifications. These specifications can be of variable length, having multiple interest rates, maturity dates, and buyback options.

In particular, we experimented with query optimizations within the architecture of our BoLT ledger prototype. We hypothesized that we would see a linear tradeoff between cost and time for queries and transactions. Since Solana's cost model is largely proportional to the amount of data that is stored on the blockchain, we conjectured that reducing the amount of data that was stored in pointers, specifically account addresses, would reduce cost linearly in terms of the number of bytes. Furthermore, we hypothesized that query and transaction times would increase linearly in relation to the number of lookups that would need to be performed.

2 Ledger Design

The architecture for the ledger prototype is largely based on the Solana Program Library (SPL) token program [4]. The BoLT smart contract program differs from SPL in the fact that it allows users to define variable length specifications with multiple interest rates, maturity rates, and buyback options. This prototype of the BoLT ledger supports instructions such as defining bolt specifications and minting and transferring bolt instances.

All data, including the executable smart contract that represents the ledger

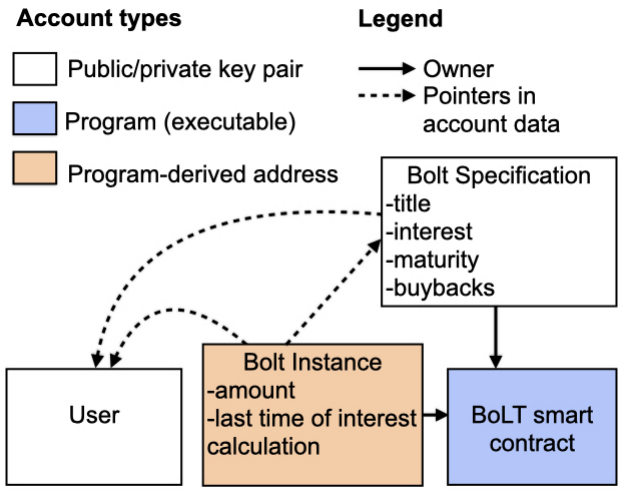

Figure 2: Architecture of BoLT smart contract

itself, is stored in its own account on the Solana blockchain. The subset of accounts which are executable are called "programs." All accounts are addressable by a 32-byte key, which can be a public key or a program-derived address. The main difference between these two types of addresses is that a program-derived address does not have an associated private key. All accounts are owned by a program. The owner of an account is allowed to modify its data.

The BoLT smart contract is the owner of all the bolt specification and bolt instance accounts. Depending on the experimental setup, the BoLT smart contract can also be the owner of all user accounts, but this is not enforced in the general case.

All of the accounts are addressable by a public key, and all of them, except for the bolt instance accounts, have an associated private key. Bolt instance accounts have program-derived addresses, which means that they are generated from a base public key, or account address, and a list of seeds. Our

list of seeds includes the associated user and bolt specification account. The function that generates this address returns an additional seed, called a bump seed, that ensures its placement outside of the ed25519 curve. The computation of the bump seed, and thus the program-derived address, can be done deterministically.

A bolt specification represents the details of a user-defined token. In our ledger prototype, a bolt specification has the fields specified in Figure 3. Bolt specifications have fields for interest rate, maturity, and buyback objects. Interest rate calculations are made during transactions such as minting and transferring bolts. The maturity object represents the date at which instances of the token can be redeemed for fiat currency, goods and services, or other bolt specifications, as specified by the issuer. The buyback object represents the types of bolt specifications that the issuer can use to buy back all instances of their bolt specification.

```
pub struct BoltSpec {
    pub account_type: u8, // indicates what type of account it is
                          // (i.e. BoltSpec, BoltInstance, ...)
    pub issuer: UID, // user account that created this bolt spec
    pub title_len: u8, // length of name
    pub title: String, // name of bolt spec
    pub interest_len: u8, // number of interest rates
    pub interest: Vec<Interest>, // list of interest rates and the
                                 // number of days they are applicable for
    pub maturity_len: u8, // number of maturity dates
    pub maturity: Vec<Maturity>, // list of maturity dates and what can be
                                 //  redeemed (fiat, goods and services, or
                                 // another bolt spec) at that date
    pub buyback_len: u8, // number of buybacks
    pub buyback: Vec<Buyback> // list of buyback options and fees
}
```

Figure 3: Definition of bolt spec

A bolt instance represents the details of a token held in a user wallet. In our

ledger prototype, a bolt instance has the fields specified in Figure 4.

```
pub struct BoltInstance {
    pub account_type: u8, // indicates what type of account it is
                          // (i.e. BoltSpec, BoltInstance, ...)
    pub user: UID, // user account that holds these bolts
    pub bolt_spec: UID, // associated bolt spec account
    pub amount: u64, // bolt balance
    pub last_calculated: Date // last time interested was calculated
}
```

Figure 4: Definition of bolt instance

The pointers to other accounts, namely the pointers to user accounts from the bolt instance and bolt spec accounts and the pointers to bolt spec accounts from bolt instance accounts, have type `UID`. This is because we tested implementations of type `UID` as a 4-byte unique identifier and a 32-byte public key. In our implementation using 4-byte unique identifiers, we had to add an extra 4-byte field to each of the user, bolt instance, and bolt specification accounts, which specified the `UID` of the respective account. This is because for certain queries and transactions using 4-byte unique identifiers, we would have to lookup the account addresses, leading to "double querying" in some instances.

3 Experimental Results

We measure the difference in performance and cost when using 4-byte unique identifiers instead of 32-byte public keys to address accounts. In all tests, there are 1000 bolt instances, 1000 bolt specifications, and 100 users. The 1000 bolt specifications and their associated 1000 bolt instance accounts were distributed uniformly across all 100 users.

The table in Figure 5 summarizes the times per task when the queries and transactions were tested on the maximal load our experimental network

Task	4-byte UID (ms)	32-byte pubkeys (ms)
Querying bolt specs	5.024	0.069
Querying bolt instances (filtration)	5.444	17.231
Querying bolt instances (computation)	10.945	0.497
Querying user wallets	6.479	17.422
Querying issuer specs	6.107	20.066
Minting	448.342	437.060
Transferring	453.908	444.833

Figure 5: Summary table of times per tasks

supported. This means that queries for bolt specifications and bolt instances and transactions were tested on 1000 bolt specifications and queries for user wallets and issuer specifications were tested on 100 users.

For tasks such as querying bolt specifications and bolt instances, holding 32-byte public keys performs faster than the 4-byte unique identifiers. The method for querying bolt specifications relies on a primitive that retrieves account information given the account public key. The fastest method for querying bolt instances relies on computing the bolt instance public key from the given user and bolt spec public keys and using the prior method in order to retrieve its account information (marked as "Querying bolt instances (computation)" in Figure 5). These results can be attributed to the fact that retrieving information associated with a public key is highly optimized on the Solana blockchain.

However, the results for the alternative method for querying bolt instances indicates that for more complex queries, having 4-byte UIDs improves performance (marked as "Querying bolt instances (filtration)" in Figure 5). This is because such queries rely on filtering through all the accounts owned by our smart contract and comparing bytes of account data at specified offsets. As we scaled the number of these queries to the size of our experimental network, as seen in Figures 6 and 7, we noticed that the time per

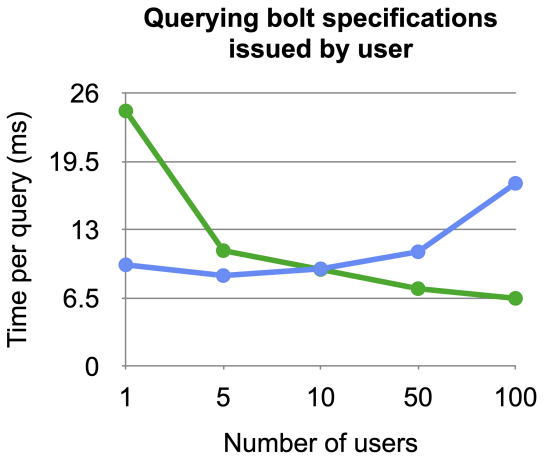

Figure 6: Time per query for bolt specifications held by user

query decreased. With smaller identifiers, we have less data to compare, which means that we begin to see improvements in query performance. We see a 2-4 times performance in tasks such as querying user wallets and querying issuer specifications.

Out of all the listed queries, querying for user wallets and issuer specifications are the ones that have the most utility to the BoLT application. Not only does having 4-byte UIDs reduce the amount of time it takes per query, but it also reduces its cost, since rent fees are proportional to the number of bytes stored in an account. The amount of data stored in pointers would reduce by a factor close to 8-fold in accounts with more pointers, such as bolt instance accounts or bolt specification accounts with several interest rate, maturity, or buyback objects.

With respect to other operations, there is little to no difference in times per transaction, such as minting or transferring bolts.

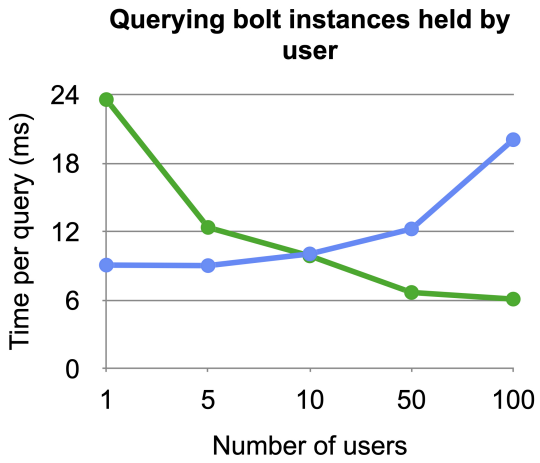

Figure 7: Time per query for bolt instances held by user

4 Surprises and Lessons Learned

Since highly-scalable blockchain protocols such as Solana are still in their nascency, there is limited support for on-chain development. I ran many surprises when developing the BoLT ledger prototype. There is a surprising lack of documentation for many of the basic ledger operations, such as adding signers to transactions. I had to reference the codebase for Solana's native token program in order to implement these operations. Furthermore, I learned that access to system primitives, such as the system clock, are restricted, likely due to the distributed nature of the protocol architecture [5]. Having access to system time is important in order to ensure interest rate calculations are handled correctly.

5 Conclusions and Future Work

In conclusion, we learned that having 4-byte unique identifiers improves performance as well as costs. Whereas as we thought that we would observe a

linear tradeoff between cost and time, with the time per query increasing with respect to the number of lookups that we would perform, we observed that the time per query decreased as we scaled up. This is because with smaller identifiers, we have less data to compare when filtering through the accounts owned by our smart contract, so we see improvements in query performance. Furthermore, since costs are proportional to the amount of data stored, addressing accounts by 4-bytes instead of 32-bytes results in a reduction in storage, and thereby, costs.

Some areas for future work in this project include fleshing out our prototype of the BoLT ledger to implement primitives such as extending trust lines, buybacks, and delegation. Another direction for future work will be experimenting with various layer-1 and layer-2 protocols to comparatively assess their flexibility and performance in the BoLT ecosystem. Current efforts are being made to standardize the interface to test various ledger prototypes implemented on technologies such as Celo, Algorand, and the Hyperledger Fabric.

References

[1] A. Yakovenko, "Solana: A new architecture for a high performance blockchain v0.8.13," Nov 2017.

[2] CNBC. "US 'test' CBDC hits record transaction speed," Feb 2022.

[3] "Solana docs." https://docs.solana.com/.

[4] "Token program." https://spl.solana.com/token.

[5] "Solana wiki." https://solana.wiki/docs/.