SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF COMPUTER ENGINEERING

Benchmarking Solana DLT

DEPENDABLE DISTRIBUTED SYSTEMS

Professors:
Silvia Bonomi
Giovanni Farina

Students:
Francesco Bianchi
Matricola: 1942637
Valerio Massimo Camaiani
Matricola: 1935012

Academic Year 2023/2024

Contents

1 Introduction to Solana

In this report, we are going to analyze the performance of Solana, a blockchain platform that stands out for its innovative approach to scalability and speed. Introduced in 2017 by Anatoly Yakovenko, Solana is designed to support high transaction volumes without sacrificing decentralization, featuring several key technologies that distinguish it from other blockchains.

Proof of History (PoH) underpins Solana's architecture, offering a unique method of recording the passage of time between events, crucial for the synchronization and efficiency of the network. This mechanism facilitates the blockchain's impressive throughput capabilities.

Solana operates in a gas-less environment, setting it apart from many other blockchains that rely on gas fees to manage network congestion. Instead of traditional gas fees, Solana uses a system of priority fees where users can opt to pay more to expedite their transactions. This model not only simplifies transaction costs but also enhances user experience by making fees predictable and often lower than on gas-based networks.

The ecosystem is rich with development tools predominantly in Rust, supported by a robust suite of libraries. Applications on Solana span from decentralized finance (DeFi) and non-fungible tokens (NFTs) to emerging sectors like gaming and metaverse applications, benefiting from its low-cost and high-speed environment.

As Solana experiences rapid growth and increasing token valuation, it has become a major player in the blockchain space. However, its accelerated expansion comes with challenges, particularly in terms of reliability. Recent network downtimes have raised concerns about stability, which are critical for user trust and continuous operation. Despite operating in a gas-less environment, Solana has introduced priority fees to manage transaction processing without the traditional gas model, aiming to balance demand with network capacity.

2 Testing Environment

2.1 Blockchain

In this chapter, we outline the setup for conducting benchmarks on the Solana blockchain. We have chosen to perform these benchmarks locally by spawning a Solana local validator which allows us to factor out networking and consensus delays. As of version 1.18, Solana has introduced a new schedulers for dequeuing transactions and we desidere to test if there was a noticible performace difference, however, the test node configuration defaults to using the legacy scheduler. To accommodate the new scheduler, minor code modifications have been implemented, presenting a challenging aspect for obscurity of the Rust programming language.

For the development of smart contracts, we utilize the Anchor framework, a comprehensive Rust-based framework designed specifically for Solana. Anchor simplifies the creation of Solana programs by providing an abstraction layer over the native Solana SDK. Despite its robustness, it is important to note that the Anchor framework does not support all transaction options or advanced features available in Solana. As a result, for certain tests where specific transaction functionalities or advanced features are required, we revert to using the bare Solana SDK.

To interact with the Solana programs and execute our benchmarks, we employ TypeScript. This choice leverages TypeScript's capabilities for easily managing and orchestrating complex asynchronous operations and its excellent support for blockchain integration.

2.2 Programs

In our testing environment, we employed a combination of custom programs and raw transactions to assess the network's capabilities comprehensively. This approach allowed us to explore both the standard functionalities provided by Solana and the custom behaviors of our developed programs.

2.2.1 Raw Transactions

For raw transactions, particularly token exchanges, we utilized Solana's native transaction procedures. These operations provide a direct way to test the basic throughput and efficiency of the network without the overhead of custom logic.

2.2.2 Custom Programs

We implemented a simple smart contract (in solana referred to as "program") that enables users to store and retrieve values on-chain. To simulate compute-intensive transactions, we introduced a "slow" version of the insert method. This method

performs dummy computations to deliberately increase the transaction processing time. The need to increase the compute unit limit is a critical aspect of our testing setup, as it directly influences the duration of the computation. We set a loop limit of 100,000 iterations for this method. This cap is crucial because the maximum number of Compute Units (CUs) that can be assigned to any transaction on Solana is limited to 1.4 million, and a larger loop would exceed this limit. Thus, this setting effectively simulates the slowest possible transaction within the confines of Solana's computational limits. This configuration not only ensures that we do not exceed the platform's computational capacity but also allows us to observe how the compute unit limit itself affects transaction processing times.

Initially, our first smart contract was designed to test non-conflicting transactions. We also developed a second contract with a shared data account to specifically assess conflict scenarios. However, we later realized that similar conflict conditions could be produced by merely issuing multiple transactions using the same user account. This revelation allowed us to streamline our test environment by removing unnecessary complexities and therefore drop the second account.

Listing 1: Exposed methods

```
// The complete program is in the appendix
#[program]
pub mod non_conflicts {
    use super::*;

    pub fn upsert_user_data(ctx: Context<UpsertUserData>,
           data: u64) -> Result<()>;

    pub fn upsert_user_data_slow(ctx: Context<UpsertUserData>,
           data: u64) -> Result<()>;

    pub fn get_user_data(ctx: Context<GetUserData>)
           -> Result<u64>
}
```

Initially, in our smart contracts, we specified a payer account different from the user's account. The purpose of this setup was to centralize transaction fees to a single funded wallet — a common practice in real-life scenarios. This method was intended to simplify the management of funds during testing by avoiding the need to fund each testing account separately. However, we soon discovered that the designated payer account must be marked as 'mutable' in the transaction, which inadvertently created a fixed point of conflict across all transactions. To accurately test non-conflicting transactions, we revised our approach by removing this centralized payer setup and began funding user wallets directly. This adjustment ensured that our tests more

accurately reflected individual transaction dynamics without unintended interference.

Listing 2: Conflict payer account

```
#[derive(Accounts)]
pub struct UpsertUserData<'info> {
    #[account(init_if_needed, payer = payer,
        space = 8 + 8 + 8, seeds = [user.key().as_ref()], bump)]
    pub user_data: Account<'info, UserData>,
    #[account(mut)]
    pub user: Signer<'info>,
    #[account(mut)]
    // The payer account needs to be marked as mutable
    pub payer: Signer<'info>,
    pub system_program: Program<'info, System>,
}
```

2.2.3 Client env

For interacting with our Solana node and executing the benchmarks, we rely on TypeScript scripts. These scripts are essential for orchestrating our test scenarios and gathering results. We predominantly utilize the SDK provided by the Anchor framework, referred to as @coral-xyz/anchor, which offers a high-level abstraction and streamlined interactions with Solana smart contracts. However, for certain advanced features that require finer control or are not supported by the Anchor SDK, we fallback to using the lower-level SDK @solana/web3.js.

In terms of data handling, we use "borsh" for deserialization of transaction return values. Borsh is a schema-based binary serialization format that is optimized for speed, efficiency, it is the de-facto standard on Rust and therefore the use used in solana.

Our benchmarking approach treats the blockchain as a black box, focusing on client-side metrics. Specifically, we measure the time it takes for transactions to be recognized in various states of confirmation by the network. Solana offers several commitment levels for transactions: "processed", "confirmed", and "finalized". Each level represents a different degree of assurance regarding the transaction's permanence and immutability on the blockchain:

- **Processed:** This is the quickest level of confirmation. A transaction is considered processed once the node has executed the transaction and kept the result in its memory. This state does not guarantee that the transaction will be committed to the blockchain.
- **Confirmed:** A transaction reaches the confirmed state once it has been validated by a supermajority of the cluster and is included in a block. However, this state still allows for the possibility of being rolled back in very rare situations.

- **Finalized:** Also known as "rooted," this is the highest level of commitment. A finalized transaction is appended to the blockchain, and the state is considered permanent. This level provides the highest security assurance and is unlikely to change except under extreme circumstances like a network-wide consensus failure.

3 Scope and Goals

The primary objective of our benchmarking project is to measure the performance differences in reaching various levels of commitment under different conditions on the Solana blockchain. By examining how transactions progress through the states of processed, confirmed, and finalized, we aim to uncover insights into the network's efficiency. Here's a breakdown of the specific areas we intend to explore:

3.1 Transaction Types

- **Raw Transactions vs. Smart Contract Calls**: We distinguish between raw transactions, such as SOL exchanges, which are straightforward and involve direct transfers between accounts, and transactions that invoke a smart contract (referred to as a program in Solana). Smart contract transactions are generally more complex, involving the execution of contract code which can affect transaction processing times. On the other side Raw transactions should be extremely optimized so we expect very good performances.

3.2 Read vs. Write Operations

- **Operational Differences**: Our tests will compare the performance of read operations, like retrieving data with `readUserData`, against write operations, such as inserting or updating data using `upsertUserData`. Write operations typically require more compute resources as they involve state changes on the blockchain, potentially leading to longer processing times.

Listing 3: Mixed transactions (no conflicts) method example

```
async banchMixed(users: anchor.web3.Keypair[], commitment: Commitment) {
    console.log("Writing␣or␣reading␣data␣from␣users");
    const s_time = Date.now();
    let txs: Promise<string>[] = [];
    for (const user of users) {
        if (Math.random() > 0.5) {
            txs.push(this.callWrite(user, commitment));
        } else {
            txs.push(this.callRead(user, commitment));
        }
    }
    console.log("All␣transactions␣sent␣in␣", Date.now() - s_time, "ms");
    await Promise.all(txs);
    console.log("Committed␣in␣", Date.now() - s_time, "ms");
}
```

3.3 Compute Intensity

- **Low vs. High Compute Intensity**: We will analyze the difference between computationally intensive operations and light ones using our `slowUpsertUserData`. High compute operations are artificially made slower by adding dummy computations, designed to test the limits of the blockchain's compute capabilities.
- **Compute Units (CUs)**: Solana measures the computational effort of transactions in Compute Units. High-intensity operations require setting a higher CU limit on transaction creation. Part of our testing involves determining whether setting a high CU limit for a light transaction unnecessarily prolongs its confirmation time.

Listing 4: Max CU limit modifier

```
const txPayload = this.program.methods[program](new anchor.BN(value))
    .accounts({
        user_data: anchor.web3.PublicKey.findProgramAddressSync(
            [user.publicKey.toBuffer()],
            this.program.programId
        )[0],
        user: user.publicKey,
        systemProgram: anchor.web3.SystemProgram.programId,
    })
    .signers([user]);
if (slow) {
    txPayload.preInstructions([
        ComputeBudgetProgram.setComputeUnitLimit({
            units: 1400000, // 1.4 M is the maximum CU limit allowed on solana.
        }),
    ]);
}
```

3.4 Scheduler Performance

- **Comparing Schedulers**: Since version 1.18, Solana has incorporated two different schedulers. Our tests aim to identify any noticeable performance differences between the legacy and the new schedulers, focusing on how each manages transaction loads and computational distribution. Since the changes primarily involve better scheduling of transactions with higher priority fees we are expecting no real differences in perfomances here.
- **Scheduler selection**: Solana offers to choose between the two schedulers when deploying a validator nodes: 'ThreadLocalMultiIterator' (the legacy scheduler)

and ‘CentralScheduler‘ (the new scheduler). The local test validator however still has harcoded the legacy scheduler. For our tests, we modified the validator code to also assess the new scheduler, allowing us to directly compare their performance under the same conditions.

3.5 Conflict Scenarios

- **Impact of Conflicting Transactions**: Another critical area of our investigation is the performance impact when sending conflicting transactions. This occurs when multiple transactions compete for the same resources or updates to the same data point. We expect that conflicts might lead to increased transaction times or higher rates of failure, or due to how transactions could put locks on accounts they using affecting parallelization.

Listing 5: Some of the conflict inducing scenarios

```
async banchWriteWithAllConflicts(
    users: anchor.web3.Keypair[],
    commitment: Commitment
) {
    console.log("Writing␣data␣to␣users␣max␣conflicts");
    let benchUsers: anchor.web3.Keypair[] = [];
    for (let i = 0; i < users.length; i++) {
        benchUsers.push(users[0]); // Always writing to the same account
    }
    const s_time = Date.now();
    let txs: Promise<string>[] = [];
    for (const user of benchUsers) {
        txs.push(this.callWrite(user, commitment, false));
    }
    console.log("All␣transactions␣sent␣in␣", Date.now() - s_time, "ms");
    await Promise.all(txs);
    console.log("Committed␣in␣", Date.now() - s_time, "ms");
}

async banchSelectiveWait2(
    users: anchor.web3.Keypair[],
    commitment: Commitment
) {
    console.log("Selective2");
    let benchUsers: anchor.web3.Keypair[] = [];
    for (let i = 0; i < users.length; i++) {
        if (i % 2 == 0) {
```

```
            benchUsers.push(users[0]);
            // Half of the time will transact to a conflicting account
        } else {
            benchUsers.push(users[i]);
        }
    }
    const s_time = Date.now();
    let txs: Promise<string>[] = [];
    for (const user of benchUsers) {
        txs.push(this.callWrite(user, commitment, false));
        // await new Promise((resolve) => setTimeout(resolve, 1));
        // Put some delay between transactions
    }

    const times: number[] = Array.from({ length: txs.length });
    console.log("All␣transactions␣sent␣in␣", Date.now() - s_time, "ms");

    for (let i = 0; i < txs.length; i++) {
        const tx = txs[i];
        tx.then((res) => {
            times[i] = Date.now() - s_time;
        });
    }

    await Promise.all(txs);
    console.log("Committed␣in␣", Date.now() - s_time, "ms");

    const avg = times.reduce((a, b) => a + b, 0) / times.length;
    console.log("Average␣time:␣", avg);
    const conflict_avg =
        times.filter((_, i) => i % 2 == 0).reduce((a, b) => a + b, 0) /
        (times.length / 2);
    console.log("conflict␣Average␣time:␣", conflict_avg);
    const free_avg =
        times.filter((_, i) => i % 2 != 0).reduce((a, b) => a + b, 0) /
        (times.length / 2);
    console.log("free␣Average␣time:␣", free_avg);
}
```

4 Results Analysis

We are going to report and analyze the obtained results in the different scenarios of interest. The environment has been reset between tests and all conditions are equivalent except when specifically stated. All the recorded times are averaged over 1000 transactions except for "slow" operations which are performed in batches of 100.

4.1 Raw Transactions vs. Smart Contract Calls

We compared the performance between simple SOL transfers (raw transactions) and transactions involving smart contract calls (Involving a write).

Transaction Type	**Processed**	**Confirmed**	**Finalized**
Raw Transactions	7.581 s	9.185 s	18.718 s
Smart Contract Calls	19.301 s	19.407 s	32.742 s

Table 1: Comparison of processing times between raw transactions and smart contract calls.

Analysis:

- Raw transactions are way faster than the program call, this implies that under the hood raw transactions follow an execution with some special optimization.
- Processing and Confirmation times are almost the same, this is expected since running a single local node the consensus is reached immediately.

4.2 Read vs. Write Operations

We assessed the performance differential between read and write operations in our smart contracts.

Operation Type	**Processed**	**Confirmed**	**Finalized**
Read Operations	18.626 s	18.977 s	31.969 s
Write Operations	20.301 s	20.407 s	32.742 s
Read or Write	19.610 s	19.836 s	31.933 s

Table 2: Operational performance for read and write operations.

Analysis:

- As expected we noticed that operations that involve onchain changes have a overhead, however the difference in performance is way smaller that expected considering the need to categorize, lock and manage used accounts when writing to them.

- This proves that solana is actually very optimized to deal with very intensive traffic.
- The test where each operation is randomly picked to be either a read or a write averages between the two as expected.

4.3 Low vs. High Compute Intensity

We compared operations having a low compute cost with ones with extremely high one, we also tried marking a low intensity operation as a high one and checking whether it would be slowed down. These tests are averaged using a sample of 100 transactions instead of 1000.

Intensity Level	**Processed**	**Confirmed**	**Finalized**
Low Intensity	1.900 s	2.185 s	14.995 s
High Intensity	6.198 s	6.394 s	19.480 s
Fake High Intensity	6.067 s	6.425 s	19.474 s

Table 3: Impact of compute intensity on processing times

Analysis:

- High compute cost transactions as expected required way more time than the low cost ones, however it can be noticed that the increase in time is non proportional to the execution cost, confirming that most of the overhead in transaction processing is outside of the execution block.
- Also the change in Finalization time is minimal confirming that the execution time is marginal with respect to block creation, this also gives us an insight that the cap of 1.4M CUs that solana puts is pretty low.
- Very interesting is on the other side that putting a high CUs limit on a cheap transaction will severely impact its processing time, therefore it is important to always calculate it properly and avoid overestimations.

4.4 Scheduler Performance

We compared the two schedulers that are available by default with different transactions both in conflicting and non conflicting environments.

Scheduler Type	Processed	Confirmed	Finalized
ThreadLocalMultiIterator	20.135 s	20.306 s	33.733 s
CentralScheduler	18.491 s	19.226 s	35.076 s

Table 4: Comparison of transaction throughput between the legacy and new schedulers.

Analysis:

- The evaluation of the two schedulers revealed no significant performance discrepancy under normal conditions, this matches our expectations since we are not using priority fees in our transactions and the main difference between the two schedulers is an improved prioritization of high fee transactions.
- One thing we noticed however is that the central scheduler has a way higher variance on benchmark results, this is most likely due to the way increased complexity of the scheduling logic adding more preprocessing overhead and therefore more volatility.

4.5 Conflict Scenarios

We also tested the impact of conflicting transactions on the network's performance.

Scenario	Processed	Confirmed	Finalized
Non-conflicting Transactions	19.421 s	19.517 s	32.634 s
All Conflicting Transactions	20.730 s	21.377 s	34.430 s
Mixed - Avg tot	20.730 s	21.377 s	34.430 s
Mixed - Avg Conflicting	20.730 s	21.377 s	34.430 s
Mixed - Avg free	20.730 s	21.377 s	34.430 s
Slow - No conflicts	6.346 s	6.791 s	20.300 s
Slow - Conflicts	6.216 s	6.816 s	19.405 s
CentralSched - No conflicts	18.711 s	19.622 s	31.034 s
CentralSched - Conflicts	18.891 s	19.342 s	33.231 s
Single Thread - No conflicts	20.678 s	21.737 s	34.155 s
Single Thread - Conflicts	20.211 s	21.874 s	33.511 s

Table 5: Effect of transaction conflicts on processing times.

Analysis:

- Despite many attempts in causing conflicting transactions to overhead (like: using both light and slow operations, mixing reads and writes to locked accounts, changing the number of threads assigned to the validator, using different schedulers

and using different metrics) we did not manage to obtain a particular drop in performance.

- This entails that the transaction scheduling is a very marginal cost in the overall transaction processing and rescheduling conflicting transactions does not have much effect.
- Also we noticed that probably the local test validator does not fully leverages all the cores available and most likely is not processing more than one transaction in parallel.

5 Appendix

Listing 6: Complete testing program -Rust

```
use anchor_lang::prelude::*;

declare_id!("8BTxUsmr5vpof3bnJJHH9isvaQKrkds7qFNNQJutnBME");

#[program]
pub mod non_conflicts {
    use super::*;
    pub fn upsert_user_data(ctx: Context<UpsertUserData>,
            data: u64) -> Result<()> {
        let user_data = &mut ctx.accounts.user_data;
        user_data.data = data;
        msg!("Added␣data");
        Ok(())
    }

    pub fn upsert_user_data_slow(ctx: Context<UpsertUserData>,
            data: u64) -> Result<()> {
        let user_data = &mut ctx.accounts.user_data;
        user_data.data = data;
        // Loop to simulate longer processing time
        let mut temp = data;
        for _ in 0..100000 {
            temp = temp.wrapping_mul(2);
        }

        msg!("Added␣data␣with␣final␣temp␣value:␣{}", temp);
        Ok(())
    }

    pub fn get_user_data(ctx: Context<GetUserData>)
            -> Result<u64> {
        let user_data = &ctx.accounts.user_data;
        Ok(user_data.data)
    }
}

#[derive(Accounts)]
pub struct UpsertUserData<'info> {
    #[account(init_if_needed, payer = user,
```

```
        space = 8 + 8 + 8, seeds = [user.key().as_ref()], bump)]
    pub user_data: Account<'info, UserData>,
    #[account(mut)]
    pub user: Signer<'info>,
    pub system_program: Program<'info, System>,
}

#[derive(Accounts)]
pub struct GetUserData<'info> {
    #[account(seeds = [user.key().as_ref()], bump)]
    pub user_data: Account<'info, UserData>,
    pub user: Signer<'info>,
}

#[account]
pub struct UserData {
    pub data: u64,
}
```

Listing 7: User Generation Procedure -TS

```
async function transferSol(
        provider: anchor.AnchorProvider,
        senderWallet: anchor.Wallet,
        recipientPublicKey: anchor.web3.PublicKey,
        amount: number
) {
        const connection = provider.connection;
        const transaction = new anchor.web3.Transaction().add(
                anchor.web3.SystemProgram.transfer({
                        fromPubkey: senderWallet.publicKey,
                        toPubkey: recipientPublicKey,
                        lamports: amount * LAMPORTS_PER_SOL,
                })
        );

        const signature = anchor.web3.sendAndConfirmTransaction(
                connection,
                transaction,
                [senderWallet.payer],
                {
                        commitment: "finalized",
                }
```

```
        );
        return signature;
 }


{
    ...
    // Generate funded users' wallets
    let s_time = Date.now();
    let users: anchor.web3.Keypair[] = [];
    let fTxs: Promise<string>[] = [];
    for (let i = 0; i < N; i++) {
        const user = anchor.web3.Keypair.generate();
        users.push(user);
        fTxs.push(
            transferSol(this.provider, this.wallet, user.publicKey, 0.1)
        );
    }
    console.log("Users generated in ", Date.now() - s_time, "ms");
    await Promise.all(fTxs);
    console.log("User funded in", Date.now() - s_time, "ms");
    ...
}
```

Listing 8: Method to send a transaction -TS

```
async callWrite(
    user: anchor.web3.Keypair,
    commitment: Commitment,
    slow = false,
    value?: number
) {
    value = value || Math.floor(Math.random() * 1000000);

    const program = slow ? "upsertUserDataSlow" : "upsertUserData";
    const txPayload = this.program.methods[program](new anchor.BN(value))
        .accounts({
            user_data: anchor.web3.PublicKey.findProgramAddressSync(
                [user.publicKey.toBuffer()],
                this.program.programId
            )[0],
            user: user.publicKey,
            systemProgram: anchor.web3.SystemProgram.programId,
        })
```

```
        .signers([user]);

    if (slow) {
        txPayload.preInstructions([
            ComputeBudgetProgram.setComputeUnitLimit({
                units: 1400000, // Need to increase CU limit for slow insert
            }),
        ]);
    }

    if (commitment === "none") { // No confirmation awaited
        return connection.sendTransaction(
            await txPayload.transaction(),
            [user, this.wallet.payer],
            { skipPreflight: true }
        );
    }

    // Returning a promise that will resolve once the desired commitment is reached
    return txPayload.rpc({
        skipPreflight: true,
        commitment: commitment,
    });
}
```