Purple: Asynchronous Smart Contracts

Octavian Oncescu
octavonce@gmail.com

Abstract

Purple is a distributed, peer-to-peer network designed for decentralized computing. Nodes participating in the network earn digital tokens by approving transactions or by executing smart contracts and then collecting their fee as reward. Purple uses a directed acyclic graph as the underlying data structure of it's ledger and transactions are processed asynchronously. Consensus is achieved by using a modern variant of Logical Clocks, called an Interval Tree Clock which enables the causal ordering of events in a decentralized network. This paper presents how interval tree clocks are used in maintaining a strict transaction order in a decentralized context without requiring sacrificing performance for providing a byzantine fault tolerant system.

1. Introduction

What enables a decentralized ledger to function is a **decentralized clock**. When Satoshi Nakamoto created Bitcoin, he used the **Proof of Work** algorithm as means of creating a decentralized clock. It works by requiring nodes who want to approve transactions to provide mathematical proof that a problem of deterministic computational complexity has been solved. The proof that is generated by the proof of work algorithm serves as the timestamp of a batch of transactions that are to be stored in the ledger and the difficulty of the proof of work algorithm is used to fine tune the time it takes for a node to approve a block of transactions. In this way, time can be quantified in a decentralized setting. However, the cost of using proof of work as a decentralized clock is the network's ability to scale and the enormous amount of electricity that is wasted in the proof generation process. In order to build a network which is both decentralized and scalable, an entirely new decentralized clock must be devised which must be able to create stamps in the same amount of time or less than it takes to verify them. Using a decentralized clock that scales well means that the entire network will do so as well.

Purple uses a decentralized clock which enables transactions to be approved in parallel while at the same time providing a strict transaction ordering, enabling the implementation of turing-complete smart contracts. Since transactions are approved by other nodes, the network's throughput increases with each node that is connected to it.

2. Ledger structure

Similarly to RaiBlocks[3], Purple's ledger is composed out of multiple accounts which have their own individual blockchains. A transaction related to an account is a block that is placed in that account's blockchain.

2.1. Accounts

Purple uses the account model for keeping track of balances. An account is a blockchain which is composed out of blocks which contain only one transaction. Accounts are opened using an Open transaction which states the account's initial balance and it's owner and is considered to be the genesis block in the account's blockchain. Each transaction made by an account must state the account's new balance. For example, if Alice has 10 tokens and Bob has 5 tokens, if Alice want to send 2 tokens to Bob she must list her new balance as being 8 and Bob receives them by listing his new balance as being 12.

2.2. Transactions

Since each account stores stores it's transactions in it's own blockchain, in order to transfer value from one account to another, two transactions have to be stored in the ledger: a **send** transaction, which is placed in the sender's blockchain and a subsequent **receive** transaction which is placed in the receiver's blockchain[Fig.1].

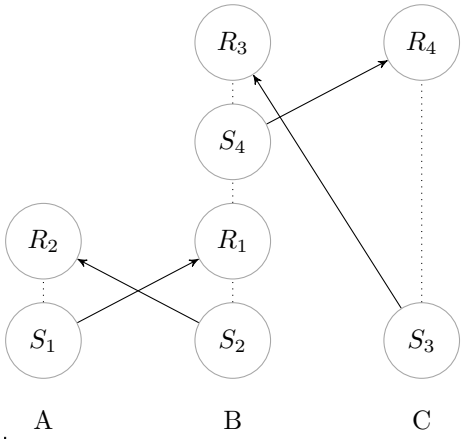

Figure 1: The structure of the ledger. Accounts A, B and C are sending and receiving funds to and from each other by storing a send transaction on the sender's chain and a receive transaction on the receiver's chain.

2.3. Ledger pruning

Since the latest balance of an account is always written on the latest transaction, devices only need to store the latest two transactions of an account in order for them to be able to verify subsequent transactions of an account. This means that the ledger can be pruned so that devices with limited storage space can use the network.

3. Decentralized clock

Inverval Tree Clocks[1], which are a modern version of Logical Clocks[2], enable the causal ordering of events inside a distributed system while at the same time accommodating a dynamic network topology. They work by incrementing a height h, and associating it with a global id i, which is a sub-interval of [0,1). The stamp of an event is the two element tuple (i, h).

In the fork-event-join model used by Interval Tree Clocks, when a node joins the network it forks the stamp of another node in the network by dividing it in two stamps with the same height but with the id interval divided between them.

$$fork((i, h)) = ((i_1, h), (i_2, h))$$

In Purple, when a node desires to join the network, it must fork a part out of each of the stamps of the latest written transactions such that the resulting stamps have their id intervals of approximately the same size.

$$join((i_1, h_1), (i_2, h_2)) = (i_3, h_3)$$

These are then joined into a new stamp which can be used by the node to approve a transaction. The node that is designated to approve a specific transaction is selected by normalizing the hash of the send transaction.

$$normalize(tx_h) = \sigma \in [0, 1)$$

The node whose id i matches the condition $\sigma \in i$ where $i \subset [0, 1)$ is the designated node for approving the transaction. Using a logical clock for ordering transactions provides several benefits, the first of which is that double spends are essentially impossible since the first received transaction by the designated node with a specific hash is always going to be approved first. The second benefit is that smart contracts can be implemented in an asynchronous context.

3.1. Unresponsive nodes

Since the designated node to approve a specific transaction can be offline or unresponsive at the time required to approve it and since no other node can approve a transaction that is designated to another node, a way of handling this case is required.

After each node has approved a transaction, other nodes are required to fork half of the stamp with the greatest height whose id interval is closest to it in clockwise direction with half of it's own stamp. The resulting stamp can then be used for approving a transaction.

The algorithm for this is the following: Take a list of all of the latest transactions in the ledger and map it to a list of their stamps. Sort the stamp list by the start points of the their id intervals and then fork the stamp at the next index after the node's stamp in that list and take the element at the first position in the resulting tuple. Then fork the node's own stamp and take the stamp at the second position in the resulting tuple and join it with the previous stamp. Since each node needs to do this, the id intervals are passed from node to node on each approved transaction.

$$getNewStamp :: [Tx] \rightarrow [Stamp] \rightarrow [Stamp] \rightarrow Stamp$$

If the new stamp of a node s follows the following property: $normalize(tx_h) \in$ i, i $\subset [0, 1)$ where tx_h is any pending send transaction's hash and i is the new stamp's id, the node is allowed to create a corresponding receive transaction for it. In this way even if a specific node that is designated to approve a send transaction is unresponsive, it will eventually be approved by other nodes who join the id responsible for approving the transaction.

3.2. Retiring nodes

Nodes are retired after they fail to approve a transaction. This is done by joining their stamp with the stamps of all of the other nodes. When a node has been retired, if it wants to approve any other transactions, it has to go through the whole process of creating a new stamp out of a part of each of the latest transactions stored in the ledger.

4. Smart contracts

The main feature of the Purple protocol is the fact that it supports smart contracts. Purple contracts are asynchronous, which means that in order to execute a contract, an account has to send a **call** transaction to the contract's address. The call transaction will be approved by a designated shard keeper who will create a **return** transaction containing the contract's output.

4.1. Payable contracts

There are two types of possible contracts: normal contracts, which requires the entity which triggered the contract to only pay the execution fee and **payable contracts** which require an amount chosen by the contract deployer to be paid in order for it to execute. This amount will be deposited in the contract's balance and the owner/s can then withdraw the funds from the contract to their own accounts.

4.2. Formal verification

Since types must be declared for all possible parameters of a contract and its state, formal verification can be performed on contracts. Contracts with type errors will simply not compile and combining this with the fact that the scripting language is immutable, the contract's code can be mathematically verified before it has been deployed. Immutability also means that most bugs found in mutable languages are simply non-existent.

4.3. Halting contract execution

In order to prevent malicious code from being executed and to maintain fast response times, each contract will have a limit on the number of opcodes it can call during it's execution. This is a hardcoded limit which cannot be changed after the launch. If the number of opcodes called by the contract exceeds the limit, it will simply stop executing. If the contract is payable, the amount paid by the executor will be refunded but, in order to prevent spam, the execution fee will not be refunded. Because a contract's call stack size is limited, doing more complex tasks requires deploying multiple contracts.

4.4. Contracts calling other contracts

Because Purple is asynchronous, a contract which requires reading the state of another contract must request that contract's state and must wait for it's reply before executing. However, this introduces a problem: What happens if the first contract gets triggered while awaiting replies from other contracts? The answer is that the second execution request must wait before the first one finishes[Fig.2], otherwise the contract's state will get mangled. Thus a "lock" must be added on the contract's state. What this means is that each contract will have an execution queue where each request must wait in line before the requests which arrived before it are handled.

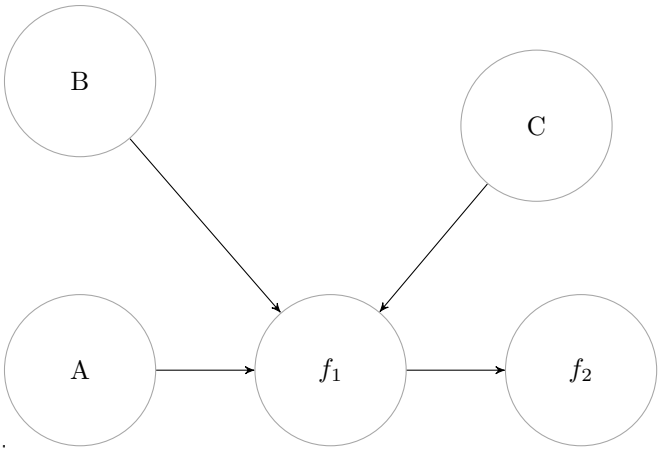

Figure 2: Account A is calling contract f_1 which must in turn call f_2 before replying. B and C are calling contract f_1 while this is happening. B and C must wait for f_1 to respond to account A before it can process their requests.

4.5. Oracles

In Purple, an oracle is also a contract. These kind of contracts are called oracle contracts and can be called by other contracts in order to obtain information about the outside world e.g. the current price of the USD. The information is pushed to the oracle's state by an outside party (e.g. a sensor running a Purple node) and in order to prevent malicious parties from pushing bad data to the contract's state, the owner of the oracle can write the code in such a way that it will only accept pushes signed by a certain account.

5. Shards

In Purple, the ledger is automatically sharded using jump consistent hashing[4]. Each account's public key is run through the jump consistent hash algorithm along with the current shard count and the output of this function is the shard number which the account is a keeper of. The account can approve transactions in the respective shard and receive transaction fees on that respective shard. If a node owner wants to approve transactions on multiple shards in parallel, he must create other accounts which get assigned to other shards. Since it is hard to determine which shard a public key will be assigned to, an owner of a node cannot create multiple accounts that approve transactions on the same shard in an easy manner thus requiring him to store multiple shards on the same machine.

6. Transaction fees

The cost for sending a transaction is 0.000000025 tokens. This serves two purposes: rewarding nodes who approve transactions and preventing spam. In the case of payable contracts, 0.2% of the contract's cost will be charged.

7. Summary

Purple is a low-latency asynchronous smart contract platform which uses a novel decentralized clock technology in order to achieve consensus. The result is a scalable and secure platform which can be used for hosting web-like applications and storing public data or for creating machine learning algorithms which run in a decentralized context and is especially suited for financial applications and the internet of things.

References

[1] A Logical Clock for Dynamic Systems. Paulo Sergio Almeida. Carlos Baquero. Victor Fonte. DI/CCTC, Universidade do Minho, Braga, Portugal. 2008. *http://gsd.di.uminho.pt/members/cbm/ps/itc2008.pdf*

[2] Time, Clocks, and the Ordering of Events in a Distributed System. Leslie Lamport. Massachusetts Computer Associates, Inc. 1978. *https://amturing.acm.org/p558-lamport.pdf*

[3] RaiBlocks, A Feeless Distributed Cryptocurrency Network, Colin LeMahieu. 2014.

[4] A Fast, Minimal Memory, Consistent Hash Algorithm, John Lamping, Eric Veach, Google. *https://arxiv.org/pdf/1406.2294.pdf*