Resource Fairness and Prioritization of Transactions in Permissioned Blockchain Systems (Industry Track)

Seep Goel
IBM Research
sgoel219@in.ibm.com

Abhishek Singh
IBM Research
abhishek.s@in.ibm.com

Rachit Garg
IIT Madras*
rachit0596@gmail.com

Mudit Verma
IBM Research
mudiverm@in.ibm.com

Praveen Jayachandran
IBM Research
praveen.j@in.ibm.com

ABSTRACT

In this paper, we consider the problem of fair scheduling of transactions of multiple types that are submitted to a permissioned blockchain system. Permissioned blockchains are being increasingly used for enterprise applications and by design are heterogeneous in nature, with different peer organizations performing different business functions. Transactions execute different smart contract operations that may have widely varying business importance. In such a setting, we argue that the typically adopted First-In-First-Out ordering mechanism for transactions in a blockchain system, which is a performance-limited resource, is inefficient and unfair. We propose a weighted fair queueing strategy for ordering transactions that can support differentiated quality of service for submitted transactions on the blockchain. The main challenge we address in this paper is to support fair allocation and differentiation in a decentralized manner, as there is no single authority that can facilitate this as in traditional systems. We demonstrate such a fair scheduling strategy and support multiple transaction types with different priorities on Hyperledger Fabric.

CCS CONCEPTS

• **Computer systems organization** → **Reliability**; **Availability**; Fault-tolerant network topologies;

KEYWORDS

Permissioned Blockchain Systems, Hyperledger Fabric, Resource Fairness, Prioritization, Weighted Fair Queueing

ACM Reference Format:
Seep Goel, Abhishek Singh, Rachit Garg, Mudit Verma, and Praveen Jayachandran. 2018. Resource Fairness and Prioritization of Transactions in Permissioned Blockchain Systems (Industry Track). In *19th International Middleware Conference Industry (Middleware '18 Industry), December 10–14, 2018, Rennes, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3284028.3284035

*This work was done when the author was at IBM Research.

1 INTRODUCTION

Over the last few years blockchain has been growing in popularity and is being applied to a variety of industries including banking and payments, insurance, supply chain and logistics, and healthcare. Blockchain, with its use of complex cryptography and distributed consensus, is a highly resource-constrained system and performance and scalability has been a topic of significant research interest and debate. One class of blockchain systems, called permissionless blockchains, permit anyone to join the network, perform transactions, participate in consensus and manage the distributed ledger. Such networks are typically homogeneous in nature, with all nodes performing the same function (called miners) and all transactions being of a single type, typically peer to peer transfer of cryptocurrency such as in the Bitcoin blockchain. Transactions include a fee in the native cryptocurrency to be paid to miners for including the transaction in the blockchain. Transactions that pay a higher fee are picked up by miners for inclusion in the blockchain, and this becomes the de facto ordering of how transactions get serviced by the network. This results in a market economy with incentive-based scheduling of transactions.

Permissioned networks, on the other hand, are highly heterogeneous in nature. They are targeted towards enterprise applications and the participants of a business ecosystem are permissioned to join the network and remains private to them. The different organizations in the ecosystem form the peers of the network, participate in consensus and maintain the distributed ledger. The organizations may perform different functions of a business process, which are codified as smart contracts. For instance, a supply chain network may comprise of manufacturers, suppliers, retailers, logistic providers, warehouses and financiers, and the ledger would include information pertaining to shipments, their status, documents such as purchase orders and invoices, and actions on each of these artifacts performed by different organizations maintained as an immutable log of transactions. It could also include periodic generation of reports, data cleansing and analytics operations, provenance tracking of supply chain artifacts for regulatory oversight. In one of our internal production system, the floods of record keeping transactions on blockchain was keeping some of the business critical transactions from going through for a very long time. Clearly not all transactions are of equal business importance and place different service requirements on the blockchain.

All permissioned blockchain platforms today consider all transactions uniformly and order them in a FIFO order, irrespective of the type of consensus algorithm they adopt. This results in suboptimal usage of blockchain's capacity to include transactions of critical business importance. Further, a single client can easily flood the network with transactions of low importance and prevent other transactions from being included in a timely manner. In this paper, we propose for the first time a fair resource allocation to different transaction types in a blockchain network. The biggest challenge we overcome is to perform such a fair allocation in a decentralized manner. We adopt a weighted fair queueing strategy [12] and support multiple priority classes for differentiated service for transactions. This permits transactions of higher business importance to be scheduled and processed faster by the blockchain, irrespective of lower priority transactions that may have flooded the network. The weighted fair queueing also supports proportionate fairness [10] for different transaction types to ensure a starvation-free schedule.

This paper makes the following contributions:

- We propose the first ever approach (3) for fair resource allocation and prioritization of transactions in a blockchain network.
- We present an implementation of our approach (4) atop Hyperledger Fabric (or Fabric) [9].
- Our evaluation (5) shows that our technique is accurate and scalable. We are able to achieve resource fairness and prioritization for transactions with minimal performance overhead.

2 BACKGROUND

Fabric is an open source modular implementation of a permissioned blockchain system. It supports running business logic as smart contracts (or chaincodes in Fabric terminology), in a decentralized fashion. Internally it implements a distributed ledger that is immutable, replicated and consistent. Fabric has different components working in tandem to provide fault resilience, security, trust and consensus to the involved non-trusting business organizations. We briefly describe the function of different Fabric components below:

Peer nodes in Fabric are operated by different organizations participating in the permissioned blockchain network. A peer node executes the chaincode and maintains a copy of the ledger. Each peer communicates with the chaincodes it executes, and provides them with an interface to operate on the underlying ledger. A peer can take the role of an Endorser, Committer or both.

- An **Endorser** simulates the transaction and collects output in the form of key-value pairs. A client application interacts with multiple endorsers (as specified and governed by endorsement policy) to agree on the results of a transaction. Apart from that, endorsers also sign their responses, which are later used to verify the authenticity of the results at commit time.
- A **Committer** that is not an endorser, does not run chaincode but maintains a full copy of the ledger. A committer receives a signed ordered block of transactions from the ordering service, which we describe next. It validates each transaction in the block and appends the local ledger with the final state changes.

The **Ordering service nodes** (hereinafter referred to as OSNs for brevity) provide an atomic broadcast delivery guarantee to the connected peers. Incoming transactions from multiple client applications are ordered by the OSN. Fabric supports a pluggable ordering service, with different implementations such as crash or byzantine fault tolerance. We utilize and modify the crash fault implementation using Apache Kafka [17] in this paper. Kafka is a pub-sub system and provides a shared concurrent queue service that implements total order. Each OSN independently cuts the blocks after a configured number of transactions or a timeout. Since Kafka queues deliver messages in a consistent order to all nodes, each OSN includes the same transactions, thus providing ledger consistency. These blocks are then delivered to the connected peers.

Channel in Fabric forms the basis for private communication between two or more participating organisations (network members) conducting confidential transactions between them. A channel typically includes a subset of network members, smart contracts (chaincodes), peers, OSNs, membership service nodes and a dedicated ledger. All the transactions in Fabric are associated with a channel, where only the authorised parties are able to transact and make modifications to the ledger.

A **Client** in Fabric is an application that interacts with the Fabric peers to perform transactions on the network. The client submits the transaction to one or more endorsers for endorsement and after receiving the sufficient number of endorsements (satisfying the endorsement policy) it sends the transaction with the collected endorsements to a OSN. The client can also subscribe to notifications from a peer for when its transactions or other transactions of interest are committed on the blockchain.

3 SYSTEM DESIGN

The main challenge we address with our design is to support prioritization for different transaction types in the system without a central controller for both determining as well as enforcing priority. Priorities are assigned by endorsers (or peers executing the transaction) independently and consolidated into a single priority value. This priority is then enforced by the ordering service which supports weighted fair queueing for the different priority classes at the time of block creation (rather than following a FIFO order). Finally, transactions in a block are validated based on their assigned priorities by the committers. Overall, the framework proposed in this paper introduces four new components in Fabric's transaction flow: (a) *Priority Calculators*, (b) *Priority Consolidator*, (c) *Multi-Queue Block Generator*, and (d) *Prioritized Validator*.

The overview of the system architecture and the transaction flow is shown in Figure 1. The client intiates a transaction by submitting it to a set of *endorsers*. Since there is no central controller and the client cannot be trusted to assign the right priority for a transaction, we leverage the endorsers to vote on the transaction priority, acting as *Priority Calculators*. While endorsers can independently assign a priority to each transaction, the criteria for priority assignment needs to be specified apriori. Different use cases may leverage different criteria and priorities may be static or dynamic. For instance, transactions pertaining to different chaincodes could statically be assigned different priorities at the time of chaincode deployment. It is also possible to have the smart contract specify a priority as

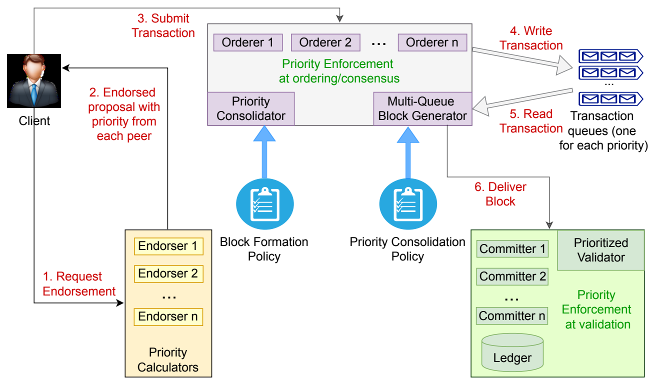

Figure 1: System Architecture and Transaction Flow

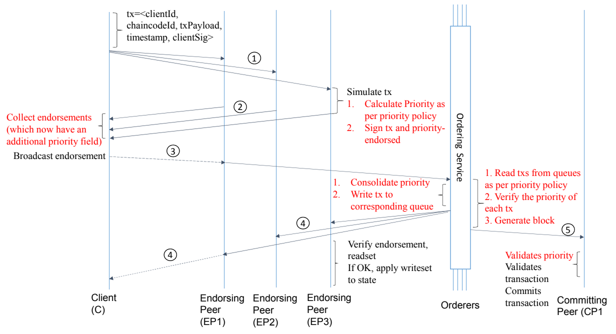
tx=<clientId, chaincodeId, txPayload, timestamp, clientSig>
1
Simulate tx
1. Calculate Priority as per priority policy
2. Sign tx and priority-endorsed
2
Collect endorsements (which now have an additional priority field)
Broadcast endorsement
3
Ordering Service
1. Consolidate priority
2. Write tx to corresponding queue
1. Read txs from queues as per priority policy
2. Verify the priority of each tx
3. Generate block
4
5
4
Verify endorsement, readset
If OK, apply writeset to state
Validates priority
Validates transaction
Commits transaction
Client (C)
Endorsing Peer (EP1)
Endorsing Peer (EP2)
Endorsing Peer (EP3)
Orderers
Committing Peer (CP1

Figure 2: Transaction Flow Diagram

part of its execution. An example of a dynamic priority assignment could be varying the priority based on load from different applications in the network perceived by different nodes. Further details are presented in 3.1.

The endorsers execute the transactions and sign the response together with the priority assigned by them. The client collects these endorsements and sends it an *OSN*, as in the regular Fabric transaction flow. Since we do not dictate any universal criteria for priority assignment, it is possible for endorsers to assign different priorities to a transaction. For example, for priorities that are static based on the chaincode, all endorsers are likely to assign the same priority. However, when priorities are assigned based on other criteria (such as load on the system for a particular application type), endorsers may assign different priorities. Hence, there is a need to consolidate the different priorities assigned by endorsers into a single priority value. This function is performed by the OSNs, based on a *priority consolidation policy*, described further in 3.2.

In Fabric using Kafka, the *OSNs* maintain only one FIFO queue of transactions per channel. In order to support weighted fair queueing for multiple priority levels, we propose to have N queues corresponding to N priority levels. In accordance with the priority decided by *priority consolidator*, the transaction is submitted to the designated queue for that priority. The *Multi-Queue Block Generator* described in detail in 3.3, reads the transactions from the priority queues based on a priority-aware *block formation policy*, which specifies the ratio of transactions from each priority level to be included in a block based on their fair share. This creates the block of transaction which is then delivered to all the committer peers.

Each committer validates transactions in the block to ensure that the endorsement policy is fulfilled and that the read-write set that was generated by the endorser peers is still valid as per the current ledger state (there are no read-write or write-write conflicts). The committer peer plays the additional role of a *prioritized validator*.

In case of a conflict between two transactions within a block, the *prioritized validator* picks the transaction with a higher priority over the transaction with a lower priority. This is another instance in the transaction flow where the priority is enforced. As a result of this exercise, each transaction in the block is either tagged as valid or invalid. The peers then append this block to their chain and commit the write sets of valid transactions to the ledger. Following this, the client is notified for the committed transactions.

The complete transaction flow is also depicted in Figure 2.

3.1 Priority Calculator

Since each endorser independently computes the priority and signs it along with the transaction response, it is impossible for the client to fake their priority. The only thing the client can do is to omit endorsements that assign it an unfavorable priority, which in our opinion is harmless as long as the endorsement policy requires endorsements from multiple participating organizations.

The client collects these endorsements and optionally verifies the signatures including the priority assignment. The endorsements will be verified later in the transaction flow as well, since the client cannot be trusted. Nevertheless, it is in the client's interest to perform the verification up front to ensure that the transaction doesn't get invalidated later. This verification involves comparing endorsements to determine if the specified endorsement policy and the priority consolidation policy (described in the next section) have been fulfilled before submitting the transaction to an OSN.

3.2 Priority Consolidator

A priority consolidation policy specified at the time of chaincode deployment, dictates how priorities assigned by different endorsers must be consolidated into a single priority value. One sample policy can be that at least k out of n endorsers should assign the same priority to the transaction, else the transaction will be considered invalid. Such a consolidation policy works well for static priority assignments. Alternatively, the policy could require aggregating the different priority values such as computing an average, which is then rounded off to the nearest integer value. The priority consolidation policy could also be modified using a channel configuration transaction, although we did not implement this in our prototype.

The OSN performs the priority consolidation based on the policy and places the transaction in the corresponding priority queue to be included in a block.

3.3 Multi-Queue Block Generator

The objective of the multi-queue block generator is to support fair queueing for multiple priorities and transaction types. The weights or the ratio in which transactions of different priority levels should be included in a block is specified as a *block formation policy*, which is part of the channel configuration. There can be scenarios where there is a need to modify the *block formation policy* during the course of operation of the channel, such as:

(1) The system designer realizes that the *block formation policy* defined at the beginning is not the best policy for the system.
(2) The system designer might want to have an online learning mechanism in place to determine the best policy and adjust the *block formation policy* accordingly.

This could also be modified using a channel configuration transaction, although we did not implement this in our prototype.

We propose a novel block generation algorithm for prioritized weighted fair queueing based on the block formation policy, where the *OSNs* generate the block by reading transactions from multiple transaction queues (see Algorithm 1). The number of priority levels (N) and the number of transactions of each priority ($\mathbb{TR}$) to be included in a block are available to the *OSNs* from the block formation policy. As an example, if there were 3 priority levels and the block formation policy was $\mathbb{TR} =< 100 : 0 : 0 >$, this essentially implies that the highest priority transactions would always be picked first, and the medium and low priority transactions are served as best effort (the 0 signifies best effort), based on whether there is space in the block to include them after all highest priority transactions have been included. Further, the lowest priority transactions will be picked as part of a block only after all medium priority transactions have been picked. If the block formation policy was $\mathbb{TR} =< 50 : 40 : 30 >$, this would try to form the block with transactions in this ratio. In case a priority level does not receive sufficient transactions before *timeout*, the remaining transactions are taken from the highest priority level with surplus transactions (see lines 17- 23 in Algorithm 1).

In the existing design of Fabric, *OSNs* cut a block if one of the below mentioned two conditions is satisfied:

(1) When the *OSN* has read the maximum number of transactions to be included in a block (*BS*, for block size).
(2) When the timer for the block being generated expires at *timeout*.

OSNs maintain only one queue of transactions. This queueing service supported by Kafka, delivers a totally ordered sequence of transactions to all *OSNs*. However, in the architecture proposed in this paper, we maintain multiple kafka queues, one for each priority level, and ensuring a total order amongst the transactions read from these multiple queues is a challenge we address in this paper.

Consider a scenario, where we have two *OSNs*, *OSN1* and *OSN2*, and two priority levels (high and low). Suppose that $\mathbb{TR} =< 100 : 0 >$. Further, lets assume at time T, there are 99 high priority transactions and 1 low priority transaction in kafka. At time $T + 1$, a high priority transaction arrives. As both the *OSNs* run a local timer which may not be in sync, it is possible that *OSN1's* timer expires at T, whereas *OSN2's* timer expires at $T + 1$. In this case, the resultant block at *OSN1* will have 99 high priority transactions and 1 low priority transaction, and the block at *OSN2* will have 100 high priority transactions and 0 low priority transactions. The two *OSNs* in this scenario have diverged and produced different sequences of transactions, which is unacceptable.

Algorithm 1: Multi-Queue Block Generator

```
1  MULTI_QUEUE_BLOCK_GENERATOR($N$, $\mathbb{TR}$, $BS$, $BN$, $timeout$)
2  Input: $N$: Number of priority levels
3  $\mathbb{TR}$: Array containing the number of transactions of each priority level
   to include in a block
4  $BS$: Maximum block size in terms of total number of transactions
5  $BN$: Block number of the block to be generated
6  $timeout$: Clock time to cut this block
7  Output: $\mathbb{B}$: Generated block of transactions
8  Global: $\mathbb{QUEUE}$: Array of queues of transactions corresponding to
   each priority level, from highest to lowest priority
9  Assumption: $sum_i(\mathbb{TR}[i]) = BS$ (the number of transactions to
   include of each priority level is normalized to the block size)
10 Initialize: $\mathbb{B}$: {}, $\forall_{i \le N}$ $\mathbb{TTCFLAG}[i] = false$
11 while ($exists_i(\mathbb{TTCFLAG}[i] = false)$ AND $sum_i(\mathbb{TR}[i]) \neq 0$) do
       /* Conditions on which a block should be formed */
12     for ($i$ = 1 to $N$) do
           /* Parse queues in decreasing priority order */
13         if ($\neg$ $\mathbb{TTCFLAG}[i]$ AND $\mathbb{TR}[i] \neq 0$) then
               /* TTC message not seen and limit of
                  transactions not reached, so read more
                  transactions from this queue */
14             [$\mathbb{TX}$, $\mathbb{TTCFLAG}[i]$] := READ_QUEUE($i$, $\mathbb{TR}[i]$, $BN$);
15             $\mathbb{B} := \mathbb{B} \cup \mathbb{TX}$;
16             $\mathbb{TR}[i] = \mathbb{TR}[i]$ - $\mathbb{TX}.length$;
17             if ($\mathbb{TTCFLAG}[i]$) then
                   /* Transfer remaining transaction buffer
                      to the highest priority level which has
                      not seen $TTC_{BN}$ yet */
18                 $h := find(\mathbb{TTCFLAG})$;
                   /* The call to find method returns the
                      first index in $\mathbb{TTCFLAG}$ array with
                      value as false, if all entries are true
                      the method returns $-1$ */
19                 if ($h \neq -1$) then
20                     $\mathbb{TR}[h] := \mathbb{TR}[h] + \mathbb{TR}[i]$;
21                     $\mathbb{TR}[i] = 0$;
22                 end
23             end
24         end
25     end
26     if ($System.currentTime \ge timeout$) then
27         Enqueue $TTC_{BN}$ in all queues;
28     end
29 end
30 return $\mathbb{B}$
```

This implies that whenever an *OSN* generates block on *timeout*, it needs to send an explicit coordination signal to the other *OSNs*. This coordination signal is a **time to cut block (TTC_{BN})** message, where *BN* is the block number being generated. When any *OSN* reaches *timeout*, it sends a TTC_{BN} message to all queues. In the design proposed in this paper, an *OSN* cuts a block only if one of the below mentioned two conditions is satisfied:

Algorithm 2: Read Transactions From Queue

```
READ_QUEUE($i$, $TN$, $BN$)
Input: $i$: Queue number to read from
$TN$: Number of transactions to read
$BN$: Block number of block to be generated
Output: $\mathbb{TX}$: Array of transactions read from $\mathbb{QUEUE}[i]$
$TTC$: Boolean set to true if $TTC_{BN}$ is received on $\mathbb{QUEUE}[i]$
Initialize: $\mathbb{TX}$: { }, $TR = 0$
while (($TR < TN$) AND ($\mathbb{QUEUE}[i].length \neq 0$)) do
    $\tau$ := DEQUE A TRANSACTION FROM $\mathbb{QUEUE}[i]$;
    if $\tau$ is $TTC_{BN}$ then
        return [$\mathbb{TX}$, $true$];
    end
    $\mathbb{TX}$ := $\mathbb{TX} \cup \tau$;
    $TR$ := $TR + 1$;
end
return [$\mathbb{TX}$, $false$];
```

(1) When the *OSN* has seen the maximum number of transactions for each priority level as per the block formation policy.
(2) When it has received the TTC_{BN} message for all the priority levels (see line 11 in Algorithm 1).

As each *OSN* can submit TTC_{BN} message, there can be multiple TTC_{BN} messages in a queue. To handle this, while generating block BN, *OSNs* stop reading from a queue on seeing the first TTC_{BN} message. Subsequent TTC_{BN} messages for that queue are ignored.

Also, note that the ordering service implemented in Fabric provides crash fault tolerance. Therefore, we assume that an orderer, if alive, shall always consolidate priorities correctly. However, in an alternate byzantine fault tolerant implementation, each orderer, while consuming the transactions from the queue, can independently verify if the priority consolidation was done correctly by the orderer that pushed the transaction to the corresponding queue.

3.4 Prioritized Validator

As blockchain is a complex distributed system, it is not sufficient to enforce priorities at *OSNs* alone. It also needs to be enforced at the time of validation and commit to the ledger. Specifically, if there is a read-write or a write-write conflict between two transactions of different priorities in a block, the transaction with higher priority should be accepted while the transaction of lower priority should be invalidated. This functionality is provided by the prioritized validator, executed by the committing peers. Observe that the ordering provided by our multi-queue block generator preserves FIFO order within each priority level, so if there is a conflict between two transactions of the same priority, the transaction that was ordered earlier will be deemed valid and the other will be invalidated.

4 IMPLEMENTATION DETAILS

The code was implemented and experimented on a fork of Fabric v1.0 [3] using Kafka consensus. The client used the officially released java SDK for making requests. We included support for prioritization at the transaction level, including a priority field in the transaction data structure and modifying the peer and client code to support this. The priority consolidation policy and the block formation policy are included in the channel configuration at the time of channel creation. While it is possible to implement the ability to modify these using a channel configuration transaction at run time, we did not implement this feature in our prototype.

When setting up the channel for communication, each orderer creates a priority queue for each priority level to be supported. In order to create multiple queues in Kafka corresponding to different priorities, we created Kafka queues with separate topic. Messages in a topic in Kafka are sequentially ordered and are associated with separate producers and read by separate consumers (the producers and consumers being part of the *OSNs*). Thus, different producers and consumers representing different priority topics were set up in the ordering service on channel creation. After the channel setup and once chaincodes are installed and instantiated, client applications can perform transactions on blockchain. All special transactions such as install, instantiate, and other chaincode lifecycle transactions and channel configuration transactions are by default executed at the highest priority level.

5 EVALUATION

5.1 Experimental Setup

Our experiments were performed with Fabric components deployed as Docker containers running atop Soft-Layer [6] servers. Each component was provisioned a separate server with 32 cores, 64GB RAM and ran Ubuntu16.04. We use Hyperledger Caliper (or Caliper) [2] as the benchmarking tool. Caliper allows users to measure the performance of a blockchain implementation with a set of predefined use cases and produces reports containing a number of performance indicators, such as tps (Transactions Per Second), transaction latency, etc. We forked and modified Caliper to be able to generate load for different priority levels and also report performance indicators across these levels.

Note that the number of priority levels is a configurable parameter but we restrict ourselves to three priority levels (high, medium, low) for operational reasons alone. There wasn't a significant overhead change with increasing the number of priority levels. Unless specified, we assume the incoming rate of transactions across the three priority levels to be in 1:2:1 ratio. This was done to model a real world scenario where only selected transactions are prioritized or de-prioritized. The block size for all our experiments was 500 and the block timeout was 1s. The default block formation policy was considered as 2:3:1. The default transaction submission rate was 500tps. We ran each experiment 10 times and in each run a total of 15000 transactions were submitted. We report the average across all the runs.

5.2 Analysis of *Block Formation Policy*

Figure 3 shows the effect of *block formation policy* on the transaction latency. To highlight the effects of prioritization, all the latency numbers have been normalized with respect to the average latency for a system without priority and the black horizontal line at $y = 1$ shows this baseline latency. We observe that when the *block formation policy* was in sync with the incoming transaction ratio (1:2:1), the transaction latency for all the priority levels is

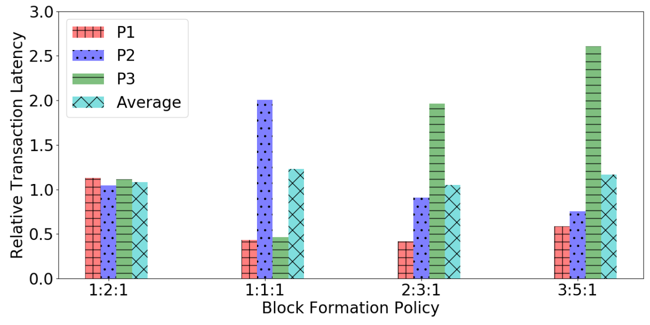

Figure 3: Effect of Block Formation Policy on relative transaction latency

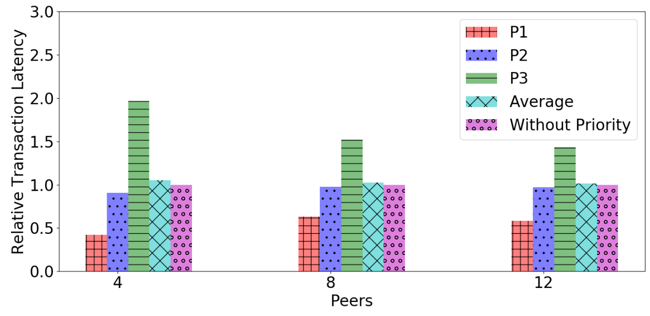

Figure 4: Effect of Increasing number of peers on relative transaction latency

roughly the same as baseline latency. The marginal increase can be construed as the overhead of our system in enabling prioritization and fairness. If for a system, it is important to achieve low latency for high priority transactions without impacting the performance of medium priority transactions, the system designer can choose to configure the *block formation policy* to be 2:3:1 or 3:5:1. This will provide the required speedup for high priority transactions but at the cost of increased latency for low priority transactions. Also, as observed for 1:1:1 and 3:5:1, the farther we skew the policy from the incoming rate, the overall average transaction latency for the system also increases. This demonstrates that the *block formation policy* enables the system designer to have fine grained control over the transaction latencies.

5.3 Effect of Increasing number of Peers on Performance

We next studied the effect of increasing the number of peers in Figure 4. As our objective was to measure the overhead introduced by our system compared to one that doesn't enforce priorities, for each blockchain size in terms of number of peers, we have normalized the latencies with respect to the average latency of the same blockchain size without priorities (the 'without priority' baseline measurements were normalized to 1). We observe that the gap between the average transaction latency of our system with priority enforcement and the average transaction latency of the system without priority is small and doesn't increase with the

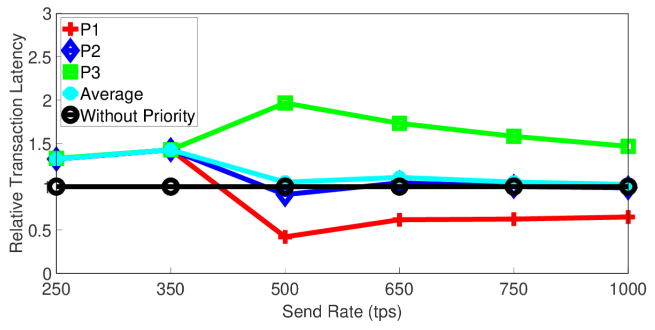

Figure 5: Relative latency with increase in send rate

blockchain size. This demonstrates that our system can easily scale and support large blockchain setups. It is noteworthy that as the number of peers increase, the number of endorsements collected and validated also increases. This causes the absolute values of the transaction latencies to increase with the number of peers. The average latency of a system with 8 peers was roughly 2.7 times the average latency of a system with 4 peers. For 12 peers, the average latency was approximately 4.3 times that of the system with 4 peers. However, our above experiment demonstrates that this increase in latency with the number of peers is not exacerbated because of introducing priorities with our system. [9] presents a more detailed analysis of scalability of Fabric with increasing number of peers.

5.4 Effect of Increasing Send Rate

In this experiment shown in Figure 5, we study the impact of increasing the send rate or load on transaction latency. Latency numbers at each send rate have been normalized with respect to the average latency of a system without priority operating at that send rate. Note that below 500tps, enabling priorities does not really help because the system was operating well under capacity and low priority transactions were not affecting the performance of higher priority transactions. From a send rate of 500tps onwards, higher priority transactions are benefited due to prioritization. An important observation here is that with the increase in send rate the gap between the average latency of a system with and without priority decreases. This implies that the performance overhead incurred due to introduction of priorities at higher send rates is lower.

5.5 Resource Fairness

In the current Fabric implementation, a single client can flood the network with its transactions, which might lead to poor performance for the other clients in the system. One of the important contributions of this paper is to enable a system design that can ensure resource fairness to all clients. This can be done by tweaking the block formation policy to provide a weighted fair share to all clients, specifically an equal weight if equality is desired.

To demonstrate this we conduct an experiment where we observe the latency numbers as one of the client starts flooding the system. We assume that the system has three clients (C1, C2, C3) corresponding to the three priority levels. We start with all clients having the same send rate of 100tps. All the latency numbers have been normalized with respect to the average latency of the system

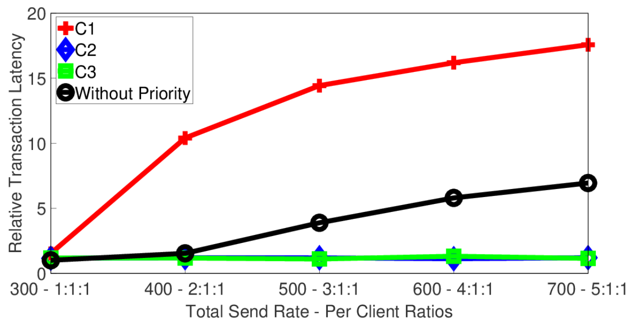

Figure 6: Relative latency with increase in send rate of one client

without priority for this rate. In each of the subsequent runs, we increase the number of transactions submitted by C1 by 100tps. As observed in Figure 6, in subsequent runs, the average latency for a system without priority starts increasing. This implies that although it is just C1 who is submitting additional transactions, clients C2 and C3 also observe increased latency, which is unfair to them. In contrast, with our system providing resource fairness, the latency for clients C2 and C3 remain completely unaffected. Only client C1 observes increased latency. Such a design protects the system against transaction flooding, resource hijacking and delay or denial of service.

6 RELATED WORK

Prioritization of tasks, jobs or transactions in distributed systems dealing with data, compute or network is a well studied problem [14] [13] [11] [20] [18]. In traditional distributed systems this is generally incorporated with a scheduling module where a scheduler determines how and what resources should be allocated to an incoming request. In such systems, a scheduler is tuned to achieve one of the many guarantees with regards to fairness, importance, resource optimization or performance [17] [21] [7] [24] [22]. Specifically, in transactional distributed databases dealing with real time traffic, where not every incoming transaction has the same importance, and many requests are associated with time bound constrains, it becomes difficult to manage the system [16] [19] [15]. A blockchain or a distributed ledger is one such transactional system.

Public blockchain implementations such as Bitcoin, Ethereum, and Litecoin [4] [23] [14] use transaction fees to determine transaction priorities. Higher the fees a client pays on a transaction, greater is its chance of inclusion in the distributed ledger. Miners try to pick transactions with higher transaction fees in order to maximise their returns for the work they do to advance the distributed ledger.

Bitcoin historically used transaction size and age of previous confirmed transactions (unspent transaction outputs or UTXOs) that are being spent in this transaction, in deciding priorities [8]. The age is determined in terms of the number of committed blocks since the block in which an unspent output was created. The formula for deciding priority was,

$$\texttt{priority} = sum_{inputs}(\texttt{inputValue*inputAge})/\texttt{TxSizeInBytes}$$

Higher the size of the transaction, lower would be its priority. Furthermore, higher the inputAge of input transactions (older UTXOs), higher the priority. However as bitcoin grew, miners have preferred to simply use transaction fees to determine whether to include a transaction or not. Transactions which pay very low or no fees have no realistic chance of being accepted.

Similarly, miners in the Ethereum network work on the notion of *Gas* and *GasPrice* for deciding a transaction priority [1]. Every transaction can specify the maximum Gas a transaction can consume and the unit price for Gas expressed in terms of Ether (the native cryptocurrency of Ethereum). Both of these fields are decided by the transaction sender. Each operation performed as part of the transaction execution consumes a fixed amount of Gas as specified by the Ethereum protocol. When a transaction is submitted and executed by a miner, `Gas consumed*GasPrice` worth of Ether are withdrawn from the sender's account, which acts as the fee paid by the transaction. If the gas consumed exceeds the maximum gas limit set by the sender, then the transaction is invalidated and the sender loses the equivalent Ether (this is to prevent spamming). As in Bitcoin, miners prioritize and select transactions that have the highest payoff in terms of the GasPrice specified by the sender. The main difference between Bitcoin and Ethereum in terms of transaction fees, is that in Bitcoin all transactions are similar while in Ethereum different contracts may have different execution times. Ethereum expects transactors to pay proportional to the resource cost of their transactions.

In contrast to such permissionless systems that prioritize based on transaction fees, permissioned blockchain systems targetting enterprise use cases are designed to not have any transaction fees. Even the least important transactions, if they are valid, are expected to eventually be included in the blockchain. Further, clients should not be burdened with having to make micropayments for every transaction they submit to the system (the scale of permissioned blockchain systems serve orders of magnitude more transactions than permissionless systems). Current implementations of permissioned blockchain systems such as Fabric [3] and Corda [5] simply order the transactions on a first come first served basis. This leaves the system vulnerable to transaction flooding, resource highjacking and even denial or delay of service. In this paper, we address these issues by providing the ability to support prioritization and resource fairness in a decentralized manner to different transaction types in a permissioned blockchain system.

7 CONCLUSION

In this paper, we presented a novel way to incorporate fairness and prioritisation to different classes of transactions in a permissioned blockchain network. We achieve this in a decentralized manner without compromising any guarantees provided by blockchain and provide a reference implementation on Fabric. Our experiments demonstrate the efficacy of the system by providing fairness and flexibility, with little overhead, and prevents individual malicious clients from overwhelming the system.

REFERENCES

[1] [n. d.]. Account Types, Gas, and Transactions âĂŤ Ethereum Homestead 0.1 documentation. http://www.ethdocs.org/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html. (Accessed on 08/14/2018).

[2] [n. d.]. Hyperledger Caliper. https://www.hyperledger.org/projects/caliper.

[3] [n. d.]. Hyperledger Fabric. https://www.hyperledger.org/projects/fabric.

[4] [n. d.]. Litecoin - Open source P2P digital currency. https://litecoin.org/. (Accessed on 08/14/2018).

[5] [n. d.]. R3 Corda. https://docs.corda.net.

[6] [n. d.]. SoftLayer Cloud Platform. ttp://www.softlayer.com.

[7] [n. d.]. Swarm mode overview | Docker Documentation. https://docs.docker.com/engine/swarm/. (Accessed on 08/14/2018).

[8] [n. d.]. Transaction fees - Bitcoin Wiki. https://en.bitcoin.it/wiki/Transaction_fees. (Accessed on 08/14/2018).

[9] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 30.

[10] Sanjoy K Baruah, Neil K Cohen, C Greg Plaxton, and Donald A Varvel. 1996. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica* 15, 6 (1996), 600–625.

[11] Ananda Basu, Saddek Bensalem, Doron Peled, and Joseph Sifakis. 2011. Priority scheduling of distributed systems based on model checking. *Formal Methods in System Design* 39, 3 (2011), 229–245.

[12] A. Demers, S. Keshav, and S. Shenker. 1989. Analysis and Simulation of a Fair Queueing Algorithm. In *Symposium Proceedings on Communications Architectures &Amp; Protocols (SIGCOMM '89)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/75246.75248

[13] Derek L Eager, Edward D Lazowska, and John Zahorjan. 1986. Adaptive load sharing in homogeneous distributed systems. *IEEE transactions on software engineering* 5 (1986), 662–675.

[14] JJ Gutiérrez García and M González Harbour. 1995. Optimized priority assignment for tasks and messages in distributed hard real-time systems. In *Parallel and Distributed Real-Time Systems, 1995. Proceedings of the Third Workshop on*. IEEE, 124–132.

[15] Jayant Ramaswamy Haritsa. 1991. *Transaction Scheduling in Firm Real-time Database Systems*. Ph.D. Dissertation. Madison, WI, USA. UMI Order No. GAX91-34322.

[16] Jayant R Haritsa, Miron Livny, and Michael J Carey. 1991. Earliest deadline scheduling for real-time database systems. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*. IEEE, 232–242.

[17] Apache Kafka. 2014. A high-throughput, distributed messaging system. *URL: kafka. apache. org as of* 5, 1 (2014).

[18] Aloysius Ka-Lau Mok. 1983. *Fundamental design problems of distributed systems for the hard-real-time environment*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[19] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. 2014. Phase Reconciliation for Contended In-Memory Transactions.. In *OSDI*, Vol. 14. 511–524.

[20] Krithi Ramamritham and John A Stankovic. 1994. Scheduling algorithms and operating systems support for real-time systems. *Proc. IEEE* 82, 1 (1994), 55–67.

[21] David K Rensin. 2015. Kubernetes-scheduling the future at cloud scale. (2015).

[22] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 5.

[23] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151 (2014), 1–32.

[24] Matei Zaharia. 2009. Job scheduling with the fair and capacity schedulers. *Hadoop Summit* 9 (2009).