Ordered HotStuff

Suyan Qu
squ27@wisc.edu
UW–Madison, USA

Shawn Zhong
shawn.zhong@wisc.edu
UW–Madison, USA

ABSTRACT

Most state-of-art Byantine fault tolerence (BFT) consensus protocols only require that all nodes process the agreed set of messages in the same order. The ordering within a batch is determined solely by the primary or through some global ordering. However, in many critical tasks that rely on BFT, such ordering is necessary since different ordering can lead to different results.

Existing work on Byzantine ordered consensus performs ordering based on the median of timestamps proposed by each node [7]. It relies on a physical clock to achieve consensus, which inherently requires all-to-all communication to achieve synchronization, leading to quadratic network traffic.

We propose Ordered HotStuff, a new consensus protocol that achieves the same linear network communication as HotStuff [6] and does not require any real-world timestamps. In HotStuff, during each round of proposal, the replicas reply to leader the order in which it sees the commands in the current proposal message from the leader. The leader then combines these proposed orders into its quorum certificate, which will be sent to all the replicas. The replicas then locally run a deterministic sorting algorithm based on the orders proposed by different replicas. We note that if each replica receives the same quorum certificate, a deterministic sorting algorithm running on different replicas should produce the same result. Ordered HotStuff ensures that, in the final execution, command 1 is ordered before command 2 only if at least $f + 1$ replicas agree in that order.

1 INTRODUCTION

In many decentralized applications, different command execution orders can lead to different results. One such example is if A wants to send 5 Bitcoins to B and B wants to send 3 Bitcoins to C, if the second transaction happens before the first transaction, it may be aborted if B does not have 3 Bitcoins.

Byzantine consensus protocols are the heart of decentralized applications. In a distributed system with n replicas, they ensure that the $n - f$ non-faulty replicas can agree on the order of command execution with the presence of f Byzantine replicas with arbitrary behavior. Batching is a common technique to group multiple commands from the clients and execute them together, thus achieving higher throughput.

In traditional Byzantine consensus protocols, the leader dictates the ordering of commands within a batch. The specification of the consensus protocol only enforces that the commands are executed in the same order, but does not specify how the order is decided or what is considered as a correct order. Byzantine ordered consensus [7] is a new specification for BFT protocols that takes ordering constraints into consideration. Specifically, replicas in Byzantine ordered consensus protocol should indicate the ordering of commands in addition to the commands themselves.

Existing work on Byzantine ordered consensus ensures ordering based on the median of timestamps proposed by each replica [7]. It relies on a physical clock to achieve consensus, which inherently requires all-to-all communication to achieve synchronization, leading to quadratic network traffic.

In this paper, we propose Ordered HotStuff, which is a new Byzantine ordered consensus protocol based on HotStuff [6]. It achieves the same linear network communication complexity as HotStuff, and does not require any real-world timestamps. Ordered HotStuff ensures that, in the final execution, command 1 is ordered before command 2 only if at least $f + 1$ replicas agree in that order.

2 RELATED WORK

2.1 HotStuff

HotStuff [6] is a quorum-based Byzantine fault-tolerant replication protocol that achieves consensus in 4 phases:

(1) **Prepare**: a leader batches a list of client messages in a block and sends it to all the replicas in a prepare message. With each block the leader also sends a hash of the parent block that this block should append to (and hence form a chain of blocks). On receiving the proposal, if the block is valid and its parent is the current locked tail of the chain, the replica will vote yes on it by sending its signature.
(2) **Pre-commit**: upon receiving $n - f$ votes from the replicas, the leader will combine these signatures into a quorum certificate. It will broadcast it to all replicas in a pre-commit message, and replicas, after verifying that this quorum certificate is valid, will set the current block as a pending tail, and respond with a pre-commit vote including its signature.
(3) **Commit**: upon receiving $n - f$ pre-commit votes from the replicas, the leader will build another quorum certificate from the signatures in these pre-commit votes. It will broadcast this quorum certificate to all the replicas in a commit message, and replicas, after verifying that it is valid, will set this block to the current tail of the chain and lock on this decision, after which it will reply with a commit vote.
(4) **Decide**: upon receiving $n - f$ commit votes, the leader combines the commit votes into yet another quorum certificate, which is broadcasted to all replicas in a decide message. After verifying that this message is valid, replicas will finally commit the messages and can respond to the client.

HotStuff achieves linear communication complexity since quorum certificates (QC) only requires star-shaped communication with the current leader at the center, and thus avoids all-to-all communications. HotStuff also allows frequent change of leaders by changing the leader before each round.

In each phase, the network communication pattern is the same: the leader broadcasts a message to all replicas, and each replica replies to the leader with a vote. Chained HotStuff is developed

based on this observation. The basic idea is that we can batch different phases from consecutive rounds in each round of communication. Suppose the current leader is proposing a block of commands b, in addition to the prepare message for block b, it also sends the pre-commit message for block $b-1$ (the block proposed in the previous round), the commit message for block $b-2$ (the block proposed right before block $b-1$), and the decide message for block $b-3$ (the block proposed right before block $b-2$). Similarly, each replica will respond with a prepare vote for block b, pre-commit vote for block $b-1$, commit vote for block $b-2$, and the decide vote for block $b-3$. This allows for concurrent processing of different blocks, improving overall throughput.

2.2 Byzantine Ordered Consensus

Byzantine ordered consensus [7] augments the correctness specification of traditional Byzantine consensus with the command ordering constraints. The authors implemented a new consensus protocol, Pompē which instantiated Byzantine ordered consensus. In Pompē, each nodes only proposes the commands, but also provides a ordering indicators on how the commands should be ordered. Pompē satisfies ordering linearizability, which is a generalization of linearizability, which ensures that if the highest timestamp for a command cmd_1 is lower than the lowest timestamp for a command cmd_2, then cmd_1 is ordered before cmd_2 for all non-faulty nodes. Pompē works in two phases. The first phase is the ordering phase, where each replica collects $2f+1$ timestamps for a command, and use the median timestamp as the assigned timestamp to determine the relative order of commands. In the second phase, consensus phase, the replicas periodically decides a prefix of the ordered commands to execute.

Pompē relies on synchronized clock across benign nodes, so it's not robust in a distributed setup where network delay can be arbitrary. Synchronizing clock on each replica inherently requires all-to-all communication. Pompē requires all-to-all comm., though it is built on HotStuff with linear communication.

3 DESIGN & IMPLEMENTATION

In this section, we present Ordered HotStuff, a byzantine ordered consensus based on HotStuff. We leverage the basic ideas in [7] to have each replica attaching an ordering indicator to each command in the block when it votes for that block. We differ from Pompē [7] in that, instead of proposing a clock time synchronized over all replicas for each command, we propose its relative order in this block (from 0 to $k-1$ where k is the number of commands in the block). This relative order is then signed by the replica and sent to the leader in the prepare phase. The leader will combine all the relative orders together into its quorum certificate (QC) and broadcast it to all the replicas in the pre-commit phase. We refer to this QC the order QC. To ensure that the leader sends the same QC to all the replicas, each replica will respond to the leader a signed set of replicas the QC contains. The leader will combine the sets from replicas into the QC for the commit phase. We refer to this QC the check QC. Finally, each replica will locally run a deterministic sorting algorithm based on the relative orders in the order QC to decide the execution order during the decide phase. Since Ordered HotStuff is based on HotStuff, our safety and liveness guarantees rely on that of HotStuff. We do note that, although our implementation is based on HotStuff, the general idea applies to any other consensus (e.g. PBFT [2], Kauri [5]).

3.1 Order QC

In HotStuff, clients send their commands to all replicas, and each proposal contains a hash for each command in this block. In Ordered HotStuff, upon receiving each command from the clients, each replica records the time at which it receives the command. When it receives a proposal from the current leader in the prepare phase, it will try to fetch the recorded time for each command in this block, and sort the commands based on this recorded time to get the relative order of the commands. If the replica has not seen a command yet, it will skip that command in its relative order. The replicas will then send the sorted commands, in the form of its index in the proposed block, to the leader in its vote. Upon receiving $n-f$ votes from the replicas, the leader will combine all these votes into an order QC that will be sent to all replicas in the pre-commit phase. The relative order indicators listed in order QC from all replicas will be used by each replica to determine the order of execution.

3.2 Check QC

However, due to network delays, even the benign nodes may not agree on a single execution order. Since the order QC only contains the set of relative order indicators from $n-f$ different replias, it is possible that a malicious leader can generate multiple order QC with different sets of $n-f$ replicas and send them to different replicas (e.g. it can send one order QC with votes from replicas 0, 1 and 2 to replica 1, but send another order QC with votes from replicas 1, 2 and 3 to replica 2 instead). Since each replica will locally determine the execution order based on order QC, if they received different order QCs, they may end up executing commands in different order, breaking the safety guarantee.

To avoid this, upon receiving an order QC in the pre-commit phase, each replica parses the order QC to find the set of replicas whose votes are included in the order QC. It then respond the leader with its set of replicas along with its signature. Upon receiving $n-f$ responds who proposed the same set, the leader will put this set, along with $n-f$ signatures from the replicas, into a check QC. It will broadcast the check QC to all the replicas in the commit phase. If a benign replica sees that the set of replicas in its order QC is the same as this set, it will lock on the current block and respond with its signature as in HotStuff. Otherwise it knows that the current replica is faulty, so it will freeze this round (skip the decide phase) and will not respond to the leader. These responds will be broadcasted to the replicas in the decide phase.

3.3 Sorting Algorithm

During the decide phase, each replica will locally run a deterministic sorting algorithm to decide the execution order. Since all replicas reaching the decide phase get the same order QC and the algorithm is deterministic, they will be able to reach a consensus on the execution order.

Another property that this sorting algorithm should satisfy is that, for each pair of commands cmd_1 and cmd_2, cmd_1 can be ordered before cmd_2 in the final execution order if at least $f+1$

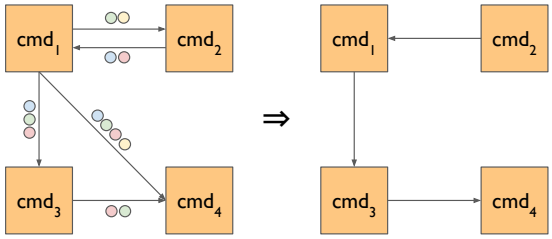

Figure 1: The sorting algorithm. Each colored circle represents a vote on the ordering of commands.

replicas think cmd_1 can be ordered before cmd_2. This is because having $f+1$ replicas who agree that cmd_1 should be ordered before cmd_2 implies that at least one benign replica agrees with this order. We define $\leq$ and $\leq_i$ such that $\text{cmd}_1 \leq \text{cmd}_2$ means cmd_1 can be ordered before cmd_2, and $\text{cmd}_1 \leq_i \text{cmd}_2$ means cmd_1 is ordered before cmd_2 in the relative order proposed by replica i. Hence, let R be the set of all replicas, this property can be rephrased as $\text{cmd}_1 \leq \text{cmd}_2$ if exist a subset $S \subset R$ such that $|S| = f+1$ and $\text{cmd}_1 \leq_s \text{cmd}_2$ for all $s \in S$.

It is important to note that $\leq_i$ and $\leq$ both satisfies transitivity. it is trivial that $\leq_i$ is transitive since if replica i orders command cmd_1 before cmd_2 and cmd_2 before cmd_3, then obviously it orders cmd_1 before cmd_3. As for $\leq$, for each pair of commands cmd_1 and cmd_2, a replica i either can determine the order between these two commands since it has seen both, or it cannot determine the order since it has not seen one or both of the two commands. Nonetheless, if it has only seen cmd_1 but not cmd_2, it can reasonably infer that $\text{cmd}_1 \leq_i \text{cmd}_2$ since cmd_2 can only arrive at a later time. As a result, a replica cannot determine the order between two commands if and only if it has seen neither commands. In this case, we assume that such a replica i would have $\text{cmd}_1 \leq_i \text{cmd}_2$ and $\text{cmd}_2 \leq_i \text{cmd}_2\text{cmd}_1$, which we will denote by $\text{cmd}_1 =_i \text{cmd}_2$ for short. Then, we have that for each replica i and any pair of commands cmd_1 and cmd_2, we either have $\text{cmd}_1 \leq_i \text{cmd}_2$ or $\text{cmd}_2 \leq_i \text{cmd}_1$, or both. Then, with $n \geq 3f+1$, we have $n - f \geq 2f+1$. By pigeon hole theorem, at least $f+1$ nodes would reach a consensus on either $\text{cmd}_1 \leq_i \text{cmd}_2$ or $\text{cmd}_2 \leq_i \text{cmd}_1$, in which case we have either $\text{cmd}_1 \leq \text{cmd}_2$ or $\text{cmd}_2 \leq \text{cmd}_1$, or both. Now, assume that we have $\text{cmd}_1 \leq \text{cmd}_2$ and $\text{cmd}_2 \leq \text{cmd}_3$. If $\text{cmd}_1 \leq \text{cmd}_3$ is not true, then $\text{cmd}_3 \leq \text{cmd}_1$ must be true. This means, by definition, that cmd_1 can be ordered before cmd_2, cmd_2 can be ordered before cmd_3, and cmd_3 can be ordered before cmd_1, implying that they happen roughly at the same time and they can be ordered in any way. Hence $\text{cmd}_1 \leq \text{cmd}_3$ should also be true.

The sorting algorithm we choose proceeds in two steps. In the first step, the replica will parse the order QC it gets and build a directed graph G as shown in Figure 1 as following: for each command cmd_j, we add a vertex v_j to the graph. Then we add an edge from v_k to v_j if $\text{cmd}_k \leq \text{cmd}_j$. In the second step, we run a variance of Kahn's algorithm on G to get a total order. Kahn's algorithm is a classic topological sort algorithm for directed acyclic diagrams. It keeps a queue Q that initially contains only vertices with no incoming edges. Then, while Q is not empty, it dequeues the first element E in Q and put it into the output list L, remove all edges from E, and enqueue all childrens of E who now has no incoming edges. To compensate that G is often times cyclic, we do the following: we first add all vertices to a set Q, then we always remove the element v_j with the least number of incoming edges and put its corresponding command cmd_j at the end of the order of execution L, and remove all its outgoing edges from G.

Correctness. To show the correctness of this algorithm, it suffices to show that at any point we add cmd_j to L if and only if $\text{cmd}_j \leq \text{cmd}_k$ for all $k \neq j$ such that $v_k \in Q$. Suppose that v_j we are removing from Q does not have any incoming edges, then $\text{cmd}_k \leq \text{cmd}_j$ is not true for all $k \neq j$ such that $v_k \in Q$. Since we have shown earlier that for any cmd_a and cmd_b, we must have either $\text{cmd}_a \leq \text{cmd}_b$ or $\text{cmd}_b \leq \text{cmd}_a$. This implies that $\text{cmd}_j \leq \text{cmd}_k$ for all k such that $v_k \in Q$. Suppose v_j has at least one incoming edge, and assume without loss of generality that it has exactly one incoming edge. Let v_p denote the source of that edge, then $\text{cmd}_p \leq \text{cmd}_j$. Since v_j is the node with the least number of incoming edges, v_p must also have at least one incoming edge. If v_j is the source of that edge, then $\text{cmd}_j \leq \text{cmd}_p$ holds. Otherwise, for each v_q with an outgoing edge to v_p, it satisfies that $\text{cmd}_q \leq \text{cmd}_p$. By transitivity, $\text{cmd}_q \leq \text{cmd}_j$. This implies that v_p must have less number of incoming edges than v_j, contradicting to our assumption that v_j is the vertex with the least number of incoming edges. ■

Complexity. The first step of this sorting algorithm requires building the graph G by parsing the relative order indicators from each replicas. Since we can have an order indicator for each command, the total complexity of this parsing is of $n \cdot O(k) = O(kn)$. Then, we can run a BFS or DFS on each node to apply the transitivity rule and get the complete graph ($O(k^2)$). In the second step, we will run k rounds to add all commands into the list L. During each round, we need to traverse Q to find the command with least indegree and update the indegree for its children in Q, the complexity is bounded by $|Q| \leq k$. Hence the total time complexity is

$$O(kn) + O(k^2) + k \cdot O(k) = O(kn + k^2)$$

Space complexity is bounded by the size of G, which has k vertices and a maximum of $O(k^2)$ edges. The total space complexity (excluding the relative order indicators) is then $O(k^2)$.

3.4 Implementation

We implemented Ordered HotStuff on top of Chained HotStuff [6]. It differs from vanilla HotStuff in that each round i of communication between the leader and replicas can serve simultaneously as the prepare phase for block i, pre-commit phase for block $i-1$, commit phase for block $i-2$, and decide phase for block $i-3$. In our implementation, during each round of communication, the leader will include block i to be proposed, order QC for block $i-1$, and check QC for block $i-2$. Similarly, each vote message will contain the vote with relative order indicators and the set of replicas in order QC. The commands in a block will only be committed if it has received and verified both order QC and check QC.

3.5 Complexity

The complexity of Chained HotStuff and Ordered HotStuff is shown in Table 1. In HotStuff, each proposal message contains k commands, and each QC consists $n-f$ votes from different replicas. Hence

	Communication	Computation
Chained HotStuff	$O(k+n)$	$O(k+n)$
Ordered HotStuff	$O(kn)$	$O(kn+k^2)$

Table 1: Communication complexity and computation complexity. k is the number of commands within a batch, and n is the total number of replicas. See Section 3.5 for complexity analysis.

the propose message if of size $O(k+n)$, and each vote message is of size $O(1)$. As for the time complexity, during each round, the replicas need to parse the proposal from the leader and verify its integrity ($O(n+k)$), update head QC for pre-commit phase of block $i-1$ ($O(1)$), update lock QC for commit phase of block $i-2$ ($O(1)$), find and commit the commands in block $i-3$ ($O(k)$), and finally reply to the leader $O(1)$. Hence the total time complexity is

$$O(k+n)+O(1)+O(1)+O(k)+O(1)=O(n+k)$$

The leader will additionally compose the proposal for block i ($O(k)$) and parse the replies from all replicas $O(n)$, leading to a total time complexity of

$$O(n+1)+O(k)+O(n)=O(n+k)$$

Ordered HotStuff, the leader adds to each proposal message an order QC composed of the relative order indicators (each of size $O(k)$) from $n-f$ different replicas, and a check QC containing votes (each of size $O(n)$) from each replica. Since each vote in check QC is represented by a bitmask where a bit is set if and only if order QC contains the relative order indicator from the corresponding replica, we expect this overhead to be negligible. Hence each proposal message from leader is of size

$$O(k+n)+k\cdot O(n)=O(kn)$$

Vote messages from each replica similarly include its vote with relative order indicators and the set of replicas in order QC, summing up to

$$O(1)+O(k)+O(1)=O(k)$$

During each round of communication, Ordered HotStuff will similarly parse the proposal and verify its integrity ($O(kn)$), update head QC for pre-commit phase of block $i-1$ ($O(1)$), update lock QC for commit phase of block $i-2$ ($O(1)$), find and commit the commands in block $i-3$ ($O(k)$), and finally reply to the leader $O(1)$. It additionally needs to run the sorting algorithm, which has a complexity of $O(kn+k^2)$. Hence the total time complexity for a replica is

$$O(k+n)+O(kn+k^2)=O(kn+k^2)$$

The leader will additionally compose the proposal for block i ($O(k)$), the order QC ($O(kn)$), and check QC ($O(n)$), leading to a total time complexity of

$$O(kn+k^2)+O(kn)+O(n)=O(kn+k^2)$$

4 EVALUATION

In this section, we present the evaluation results of Ordered HotStuff. We aim to answer the following questions:

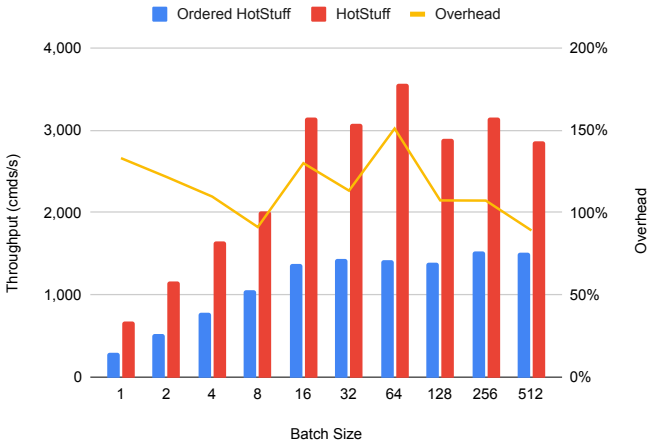

Figure 2: Results of batch size experiment.

- How well does Ordered HotStuff perform under different batch sizes? (Section 4.2)
- Can Ordered HotStuff scale with the number of replicas? (Section 4.3)

4.1 Experimental Setup

All experiments presented in this section are performed on a cluster of 20 Intel x86 server machines on CloudLab [3]. Each machine is equipped with an Intel Xeon 8-core D-1548 CPU at 2.0GHz, and 64GB of DDR4 memory, and a 10 Gbps Mellanox ConnectX-3 NIC. The network latency between nodes is 0.15ms.

We measure the end-to-end throughput from the client side for both Ordered HotStuff and Chained HotStuff (which we will refer to as HotStuff for the rest of this section). We also report how much overhead is added by maintaining ordering within each batch. The client keeps a total of 6,400 outlying requests at any point of time: whenever it receives $f+1$ responses from different server replicas, it sends a new request. Throughout the experiment, we use one client since a single client is sufficient to generate the workload to saturate the server replicas.

Both HotStuff and Ordered HotStuff use a single thread to process client requests and another thread to respond to clients. The protocol itself is event-based, but since each round of proposal relies on the QC from the previous round, the protocol proceeds in a synchronized manner. For a more direct comparison, we remove signatures and integrity checks from both HotStuff and Ordered HotStuff.

4.2 Batch size

In this experiment, 4 replicas and 1 client run on 5 different nodes. We vary the batch size from 1 to 512. We notice that for both HotStuff and Ordered HotStuff, the throughput increases linearly as we increase the batch size exponentially until the batch size reaches 16, after which the throughput remains relatively stable. We suspect that this is because we reach the bottleneck for committing the client requests and responding to the client. To our surprise, regardless of the batch sizes we use, the extra overhead we introduce is relatively fixed between 100% and 150%. This is not expected since

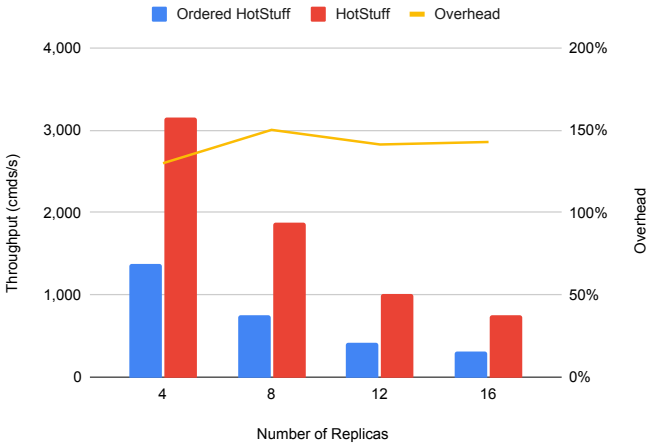

Figure 3: Results of scalability experiment.

for fixed number of replicas, the compute complexity of HotStuff is linear while that of Ordered HotStuff is exponential as shown in Table 1. We deduce that the relatively constant overhead is likely since, even though all the replicas are running on the same cluster, the network communication cost is still the main bottleneck. With 4 replicas, the communication cost for Ordered HotStuff would be a constant of 4 times the communication cost of HotStuff. Hence an overhead of 100% 150% is reasonable. In real-world deployment, we expect the machines to be geo-distributed all over the world, in which case the network latency would increase significantly. In that case the performance of HotStuff and Ordered HotStuff would be more restricted by network communication cost, and the compute cost would seem negligible. Conversely, if we run both protocols on replicas connected by faster networks, we may see the overhead increase linearly with the batch size.

4.3 Scalability

In this experiment, we fix a batch size of 16 and vary the number of replicas from 4 to 16. We similarly run 1 client in the same cluster as the server replicas. We notice that the throughput for both HotStuff and Ordered HotStuff drops exponentially as the number of replicas increase linearly. It makes intuitive sense that the performance degrades when the number of replicas increases since more replicas means that the protocol needs to keep more number of nodes synchronized, leading to dramatic downgrade in overall performance. Similar to the previous experiment, we also noticed Ordered HotStuff introduces a relative constant overhead of slightly below 150% compared to HotStuff. This is because, when we keep the block size constant, both communication cost and computation cost for both HotStuff and Ordered HotStuff should increase linearly as shown in Table 1. We expect to see a similar trend if we further increase the number of replicas.

5 DISCUSSION

Although we implement our Ordered HotStuff on Chained HotStuff, the general idea of passing a relative order indicator along with each command in the block and run a deterministic sorting algorithm on this ordering applies to all other BFT protocols as well. It is capable of avoid byzantine oligarchy without changing the communication pattern of the original protocol. The Ordered HotStuff we implemented is not optimized and has a few limitations. In this section, we discuss some main limitations and how they can be resolved.

5.1 Order across Batches

Ordered HotStuff only ensures ordering between commands within the same batch. A leader can still manipulate the order in which the commands execute by intentionally censoring some command and postpone it to the next block, in which case it will be executed after all commands in the first batch. To avoid this, we can have each replica vote on which commands to be executed as well, and postpone the commands not included in the current block to the next block.

One approach is that, during the decide phase when we generate graph G, we can leave out the commands that only appear in less than $2f+1$ relative order indicators from $2f+1$ different replicas. This also means that we no longer need to assume that if a replica i has not received cmd_1 and cmd_2, then $\text{cmd}_1 = \text{cmd}_2$. This is because for any vertex v_j in the graph, at least $2f+1$ replicas has seen cmd_j, and they all can determine the ordering relationship of cmd_j with any other command cmd_k either because it has seen both commands or because it can infer from the fact that it has seen cmd_j but not cmd_k. By pigeonhole principle, at least $f+1$ of them would reach an consensus on which way they should be ordered, which we assume without loss of generality that $\text{cmd}_j \leq_i \text{cmd}_k$. Then this, by definition, indicates that $\text{cmd}_j \leq \text{cmd}_k$.

Similarly, we can choose to include a command if it has been seen by at least $\frac{1}{2}(n+f+1)$ different replicas. In this case, since each QC can hold up to the relative order indicators from $n-f$ different replicas, for any pair of commands cmd_1 and cmd_2, we will have at least $2 \cdot \frac{1}{2}(n+f+1) - (n-f) = 2f+1$ who has seen both cmd_1 and cmd_2, in which case they can determine the order between these two commands by the order it receives the two commands. Then similar to the previous case, at least $f+1$ replicas can reach a consensus on their ordering relationship, and replicas do not have to infer about the order between two commands in the case that it has only seen one but not the other. It is also worth noting that the this approach only requires that $\frac{1}{2}(n+f+1) \leq n-f$, which we can resolve to $n \geq 3f+1$, the same fault tolerance provided by HotStuff and the optimal fault tolerance for any BFT protocol.

5.2 Independent Commands

In most real-world applications, most commands do not exhibit strong dependency relationship and hence can commute. For example, in a payment network such as Ethereum [1], transactions only show dependencies when the they share the same sender/receiver, and they can be executed in any order without affecting correctness. In such workloads, we can significantly reduce the overhead by only including the information necessary to summarize all dependency relationships between any two commands in the current batch instead of proposing a total order, and we only need to resolve the ordering between these commands and execute all other commands in any order (even asynchronously by multiple threads).

To achieve this, the application running HotStuff needs to supply a function to indicate dependency relationship between any two commands. Upon receiving a command from the client, each replica can use this dependency function to resolve any dependency between this command and a previously seen command. Although this check seems to expensive ($O(K^2)$ where K is the number of all pending commands), in most cases this dependency can be checked in an efficient way using indexing.

5.3 Improvements

Our implementation for Ordered HotStuff is not optimized. One example is that the relative order indicators from each replica is indeed a permutation of consecutive numbers from 0 to k, which can be represented in a more compressed and efficient manner to reduce network overhead. As for computation cost, the sorting algorithm we choose is not very efficient and incurs a time complexity of $O(kn + k^2)$. Alternatively, one potential alternative is to assign a score to each command based on its order proposed by some replica i. Then we sum the score up for across all replicas to get a final score for each command, and sort the commands based on the final score. This will reduce the overhead to $O(kn + k \cdot \log k)$. At this point we have not yet figure out a way to assign the values so that the basic properties discussed in Section 3.3, and we defer this to future work.

6 CONCLUSION

We present Ordered HotStuff, new protocol based on Chained HotStuff that avoids Byzantine oligarchy in execution order of commands. After leader sends a proposal, Ordered HotStuff requires each replica to propose an execution order for the commands in this proposal. The leader will collect the execution orders proposed by $n - f$ replicas into an order QC and broadcast it to all replicas. Each replica will run a deterministic sorting algorithm to decide the final order in which the commands should be executed in based on the order QC it receives. Our experiment show that Ordered HotStuff introduces an overhead of 100% 150% on throughput regardless of the batch size or the number of replicas. We deduce that this is because the performance of both HotStuff and Ordered HotStuff is bounded by network communication. Finally, although we implement Ordered HotStuff on Chained HotStuff, the same idea can be applied to all BFT protocols to avoid Byzantine Oligarchy on execution orders, without modifying the network communication pattern.

[4]

REFERENCES

[1] Vitalik Buterin. 2014. Ethereum: A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper Accessed: 2016-08-22.

[2] Miguel Castro, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.

[3] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. 2019. The design and operation of CloudLab. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 1–14.

[4] John H Howard, Michael L Kazar, Sherri G Menees, David A Nichols, Mahadev Satyanarayanan, Robert N Sidebotham, and Michael J West. 1988. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)* 6, 1 (1988), 51–81.

[5] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. 2021. Kauri: Scalable BFT Consensus with Pipelined Tree-Based Dissemination and Aggregation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 35–48. https://doi.org/10.1145/3477132.3483584

[6] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 347–356.

[7] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. 2020. Byzantine ordered consensus without Byzantine oligarchy. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 633–649.