Fault-Tolerant and Deterministic Mechanism for Maintaining Dependency Data in Hyperledger Fabric

Karan Gupta Pranjal Prajapati

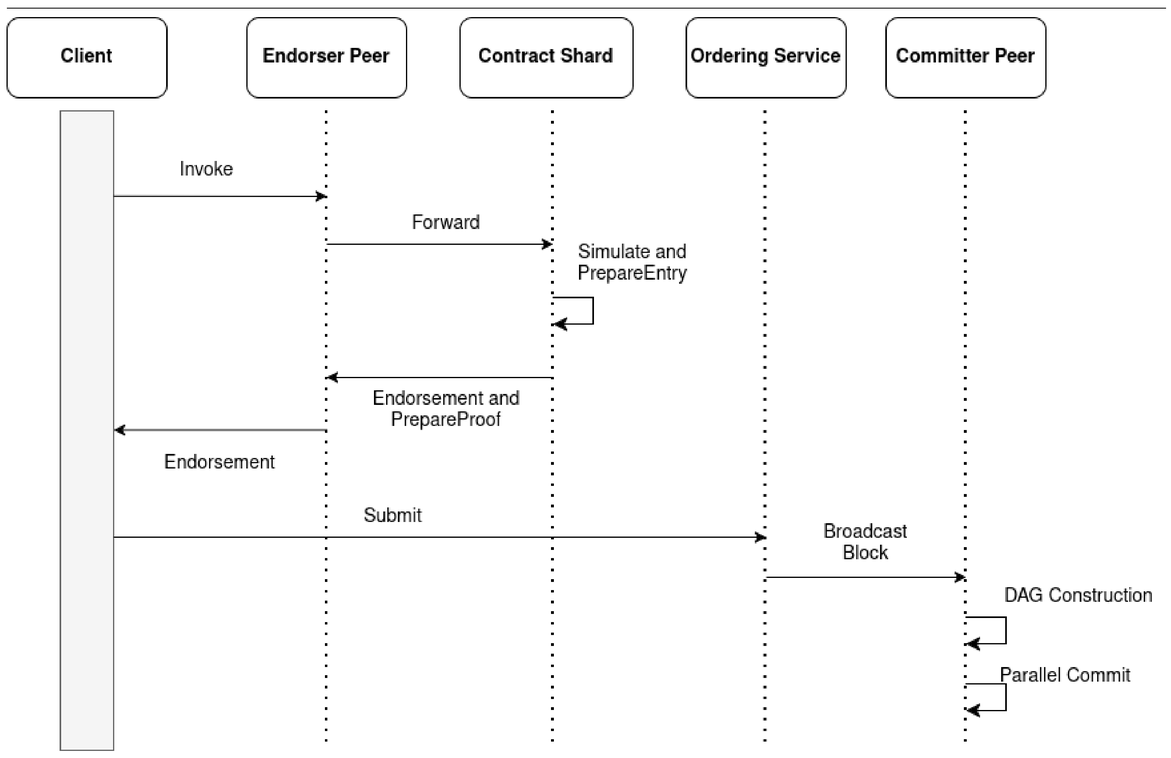
Indian Institute of Technology, Hyderabad

Introduction

Hyperledger Fabric uses an execute–order–validate pipeline with endorsement policies and MVCC checks. Under contention (i.e. hot keys and popular contracts), late validation causes wasted work and low parallelism.

- **Problem:** Proposed global sequencing of dependency information creates a bottleneck when many transactions touch the same keys or contracts.
- **Goal:** Eliminate the single-leader protocol in a crash-fault-tolerant (CFT) setting while keeping the design simple.
- **Our idea:** Shard dependency tracking by smart contract: treat each contract as a shard served by a small Raft group (with one leader per shard).

Proposed flow



Proposed algorithm

The following step-by-step methodology was followed to design and evaluate the algorithm.

1. System model:

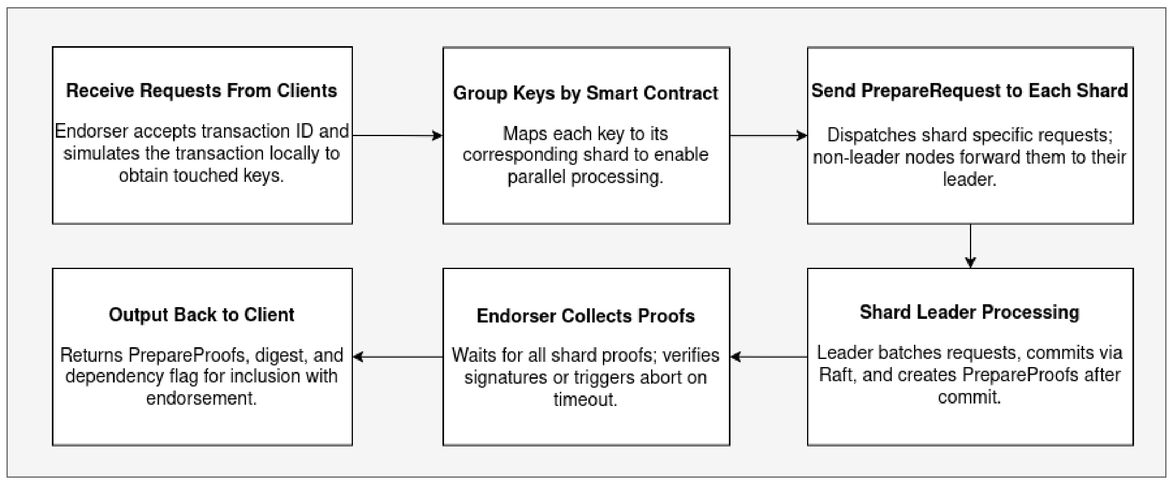
- A permissioned blockchain similar to Fabric: endorsers, orderers, and committing peers.
- Supports crash fault tolerance (CFT) with Raft-style leader replication.
- Each smart contract (chaincode) is modelled as a separate shard.

2. Single-leader baseline:

- All the transactions go through a single global leader.
- There is no explicit per-contract dependency establishment, and ordering is purely global.
- This models the current non-sharded behaviour.

3. Sharded Raft-based design:

- Each contract C has a Raft group of N nodes (typically 3 or 5, for 1 and 2 faults respectively).
- Endorser simulates the transaction, collects read/write sets, and sends a `PrepareRequest` per involved shard.
- Shard leaders batch multiple prepare requests into a `PrepareEntry` in the Raft log, replicate, and once committed, respond with a `PrepareProof` containing:
  - Transaction ID,
  - Shard ID,
  - Commit index,
  - Leader signature / identifier.
- Endorser either waits for proofs from all relevant shards or times out and aborts.



Experimental setup

We showcase a small experiment that was run on 3 different VMs running Ubuntu-20.04:

Abstraction:

- Simulator run using MPI with 8 processes (1 endorser node + 6 shard nodes + 1 committer node).
- Each process corresponds to a logical node, and physical mapping to VMs is flexible.

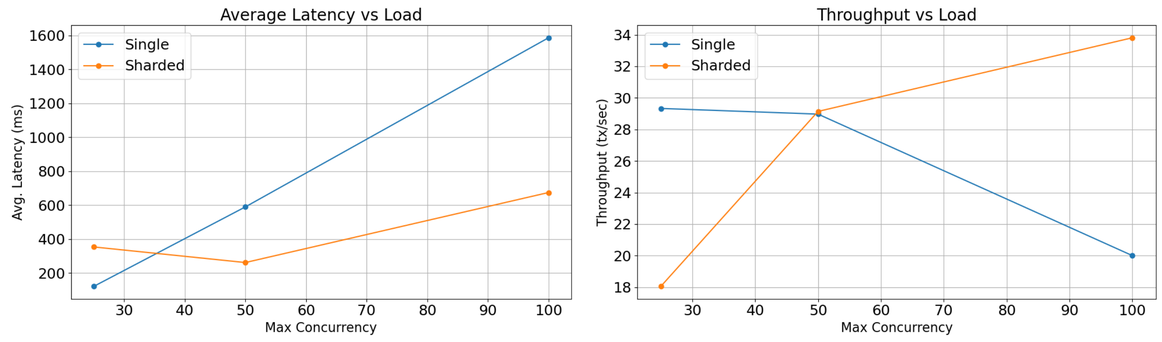
Workload generation:

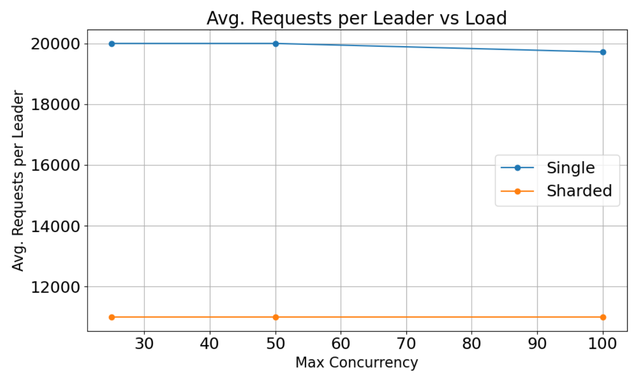
- Transactions choose a smart contract uniformly at random.
- With probability p_{cross}, they involve one additional shard (i.e. cross-contract access).
- Each transaction touches a set of pre-determined keys.

Simulation parameters

- **Total transactions:** 20000 to ensure that steady-state behaviour is attained.
- **Total processes:** 8 (1 endorser node + 6 shard nodes + 1 committer node).
- **Total shards:** 2 for sharding evaluation, and 1 for single-leader evaluation.
- **Batching:** 20 being the maximum batch size and 300 ms being the batch timeout.
- **Prepare timeout:** 2000 ms after which it aborts.
- **Cross-shard probability (p_{cross}) :** 0.1 which controls how often transactions depend on multiple contracts.

Results and discussion





The carried out experiment shows the following trends:

- **Throughput:** As we increase the contention, total throughput in the sharded design increases consistently, while it falls off under high contention for the single design.
- **Latency:** For low to moderate load, the sharded design shows latency outputs comparable or less than those shown by the single design. This changes quickly with the increase in load, as shown in the figure.
- **Load on Leader:** As the load increases, the average number of requests handled per leader remains almost constant, and in the sharded design it stays close to little more than half of what the single design demonstrates. The reason for the slight decline in value for the single design is the global leader becoming saturated under high load.

Conclusions

- Treating each smart contract as a separate Raft shard is a simple yet effective way to introduce sharding in a permissioned blockchain like Hyperledger Fabric under a CFT model.
- The proposed protocol lets endorsers collect per-shard dependency information before ordering, which reduces late aborts and wasted execution.
- Our simulation shows that the sharding approach improves throughput, latency and reduces contention on the global leader, especially when most transactions are single-shard.
- Overall the approach offers quite a clean middle-ground: more concurrency than a single-leader architecture, with considerably less complexity than Byzantine fault tolerance (BFT).

References

[1] H. Dang, T. Dinh, M. Thai, et al.
On sharding permissioned blockchains.
In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2019.

[2] A. Gupta, P. Kumar, et al.
Thunderbolt: Concurrent smart contract execution with non-blocking reconfiguration for sharded dags.
arXiv preprint arXiv:2407.09409, 2024.

[3] Y. Jiao, X. Wang, Y. Long, et al.
Reinshard: An optimally sharded dual-blockchain for concurrency resolution.
arXiv preprint arXiv:2109.07316, 2021.

[4] Diego Ongaro and John Ousterhout.
In search of an understandable consensus algorithm (raft).
In *USENIX Annual Technical Conference (USENIX ATC)*, pages 305–319, 2014.

[5] Mengyao Wang, S. Lin, et al.
Smart contract parallel execution with fine-grained state accesses.
In *IEEE International Conference on Blockchain*, 2023.

[6] X. Wu, H. Zhang, et al.
An efficient sharding consensus algorithm for consortium chains.
Nature Scientific Reports, 2023.