M.Tech Thesis Defense
Dept: CSE

HoneyBadgerBFT as an ordering service in Hyperledger Fabric

Superviser: Prof. Sandeep K. Shukla

Presenter: Deepak Yadav (18111015)

Outline

- Background
  - Blockchain
  - Hyperledger Fabric
  - Consensus
- Problem
- Solution
  - HoneyBadgerBFT
  - Integration
- Result
- Conclusion

1. Background

Blockchain

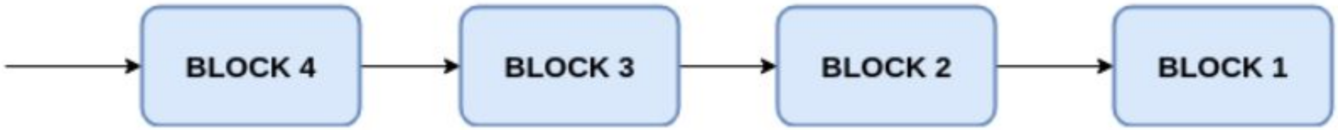
- **Bitcoin**, Satoshi Nakamoto, 2008.

  What is Blockchain?

- Append-only immutable distributed ledger of transactions.
- Shared and maintained between network participants.

  How it works?

- Cryptology : digital signatures, merkle tree, hash links
- Consensus



Types of Blockchain

<u>Blockchains categorized into two broad classes:</u>

- Permissionless blockchain
  - Any node can participate in the system
  - No verified identity required for the node
  - Allow equal and open rights to all of its participants
  - e.g. Bitcoin and Ethereum networks
- Permissioned blockchain
  - Nodes in the system are identifiable to each other
  - Need special permissions to read, access and write data into the blockchain
  - e.g. Hyperledger Fabric

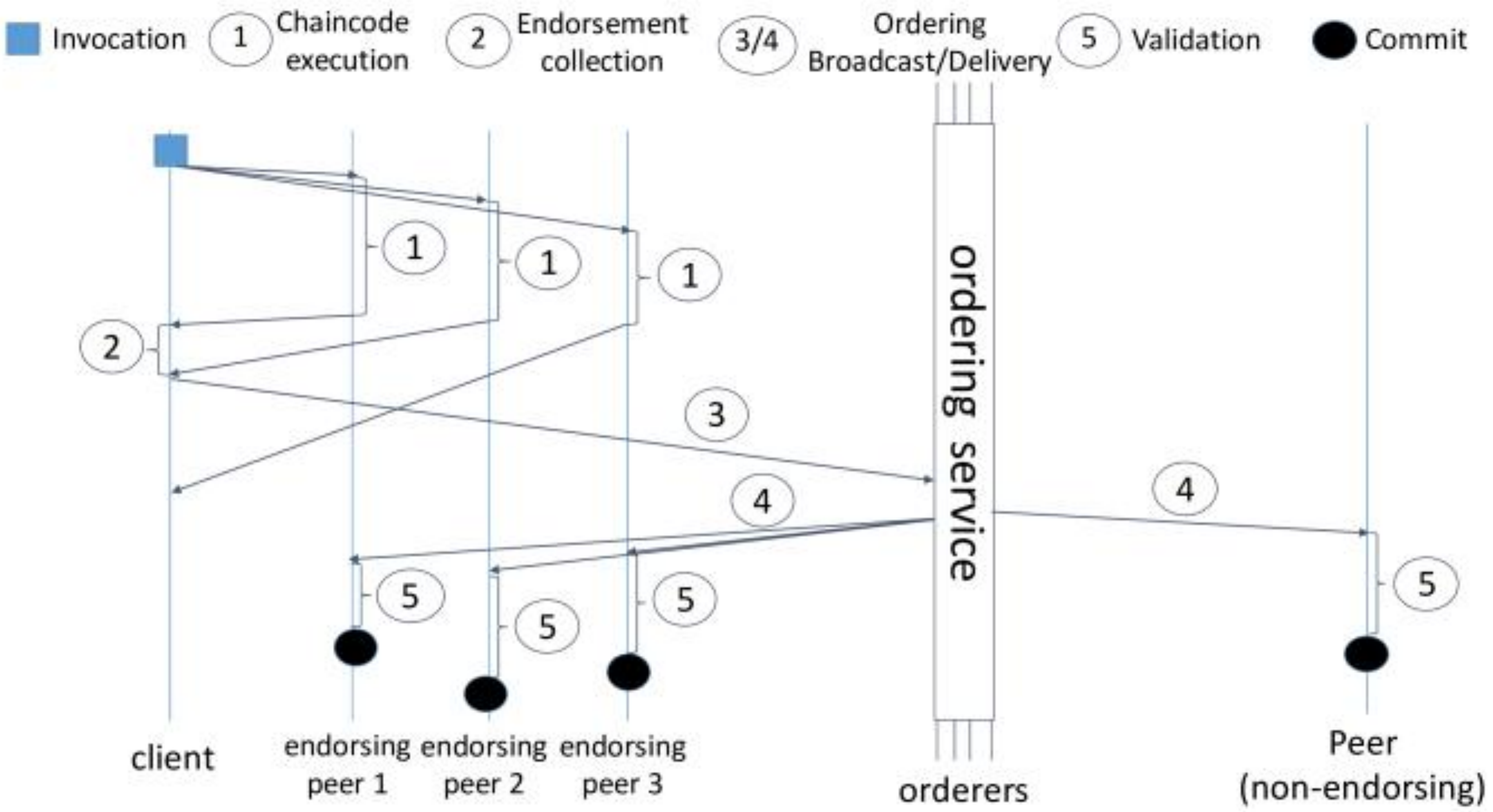
Hyperledger Fabric

- Fabric is the most promising open-source **permissioned** blockchain for running distributed applications.
- It is designed for **enterprises** and has modular design to support multiple use cases of industry.
- First blockchain to allow developers to write smart contracts/ applications in general-purpose programming languages like Go, Java, Nodejs etc.
- It introduces the novel **execute-order-validate** blockchain architecture.
- First blockchain to **allow plug-able consensus** protocol in ordering service, So users can choose and plug consensus according to threat model and application.

Roles in Fabric

- Clients
  - Orchestrate the execution phase by **submitting transaction proposals** for execution
  - Broadcasts the transactions for ordering
- Peers
  - Peer has two roles: **Endorser** and **Committer**
  - Special peers called endorsers execute transaction proposals
  - Committers validate the transactions and update the ledger
- Ordering Service Nodes
  - Nodes (orderers) that collectively form the ordering service
  - Do not participate in the execution and the validation phases
  - Due to this design, Fabric is able to deliver pluggable consensus

Transaction flow in Hyperledger Fabric



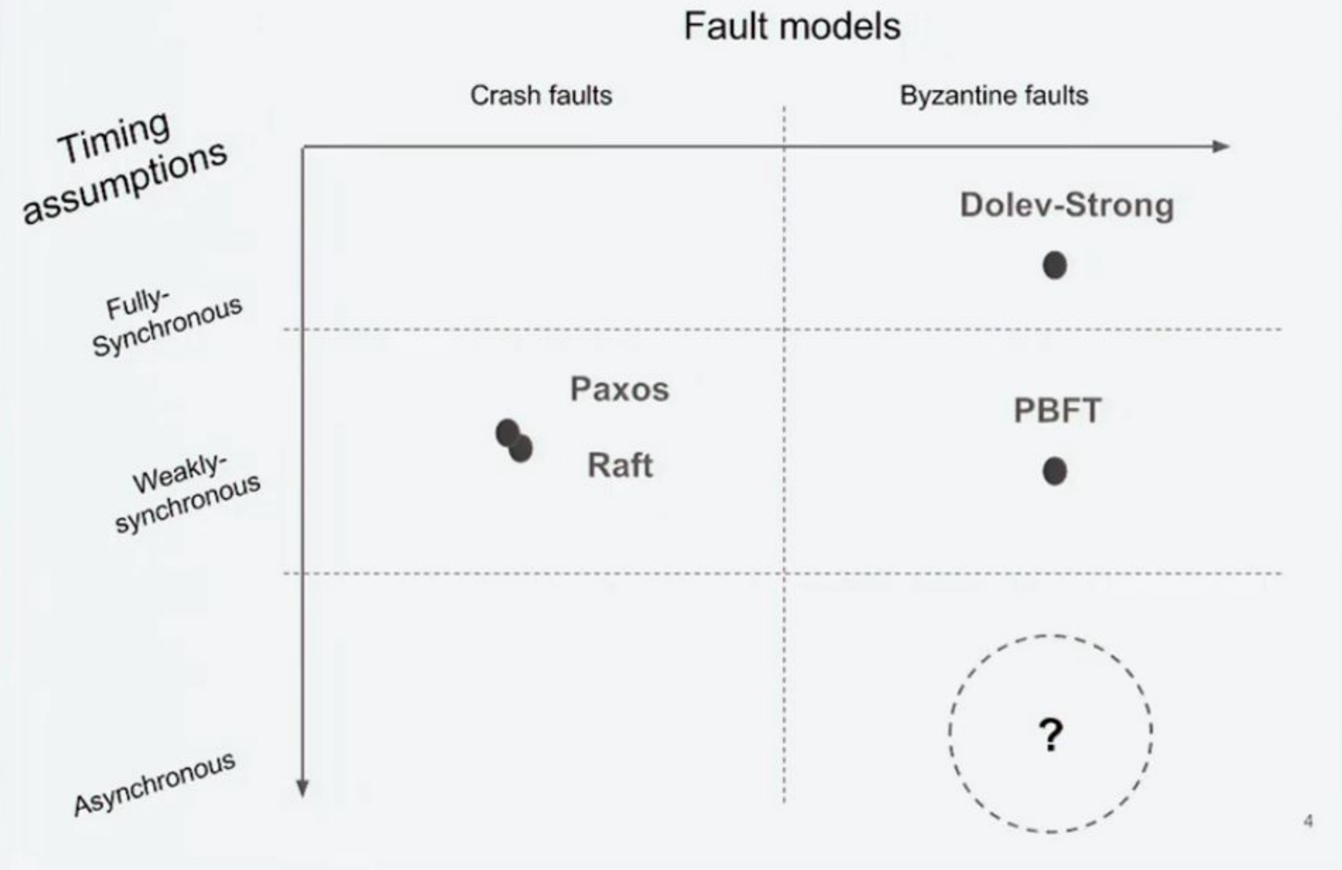
Consensus protocols in Fabric

- Solo (official)
  - Simple consensus protocol which consists of only one orderer.
  - No consensus is required in Solo as there is only one central authority.
  - Not meant to be used in production.
- Raft (official)
  - Consists of only multiple orderer nodes
  - Raft is a crash fault tolerant (CFT) ordering service.
  - Raft follows a “leader and follower” model, where a leader node is elected (per channel) and its decisions are replicated by the followers.
- PBFT

Components of the Fabric may be controlled by different organizations with different goals, they may act maliciously as well

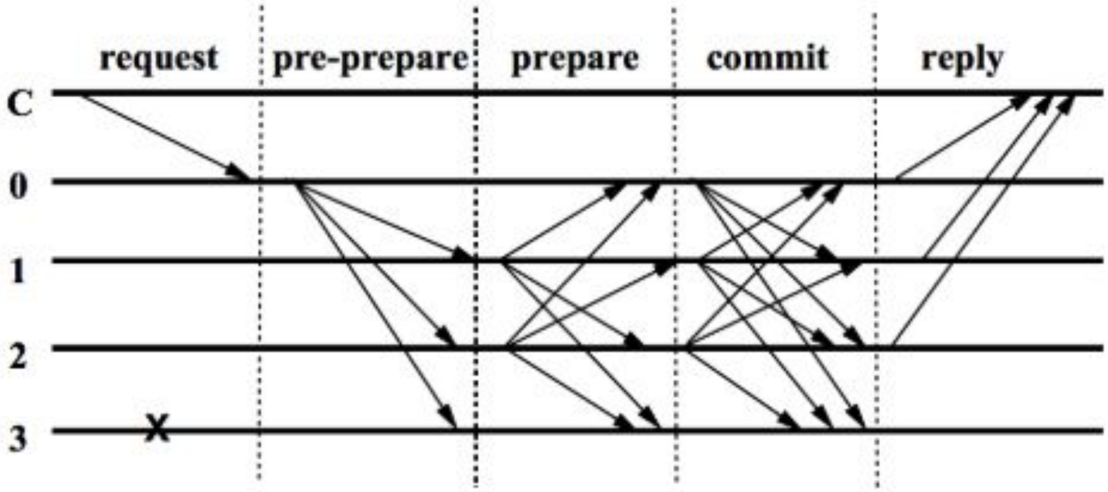
2. Problem

Popular protocols for Private Blockchain



Flow of typical synchronous protocol

- Starts a leader with Δ timing assumption
- If leader is uncorrupted and delivers messages within Δ, then the “fast path” portion of protocol succeeds
- Otherwise, elect a new leader, and increase the timeout



Problem in current protocols

- Synchronous : all messages are delivered within Δ time.
- Asynchronous : all messages are delivered eventually.

- All of them have some timing assumption, if that breaks.
  - Liveness property is at stake.
  - Performance degrades significantly.
- Tuning timing parameter is a task
  - If timing parameter is small, then we may have lot of recoveries/ view changes
  - If timing parameter is large, then we have slow recoveries after partition

N_n	New View with view n
V_n	View change for view n
PP_n	Pre-prepare from replica n
Req	Client request
■ (red)	Send withheld
■ (pink)	Delayed receive

*	Start timer
**	Increment view
●	Message delivered
○	Message sent
X	Message ignored by protocol
■ (green)	Rapid message delivery between all honest parties

Time ↓

	0 (faulty)	1	2	3
0Δ	●Req* ○PP_0 view:0	●Req* view:0	●Req* view:0	●Req* view:0
1Δ	○V_1 ●V_1* view:1	○V_1 ●PP_0X view:1	○V_1 ●PP_0X ●V_1* view:1	○V_1 ●PP_0X ●V_1* view:1
2Δ	○V_2 ●N_1,PP_1X ●V_2* view:2	●V_1 ○N_1,PP_1* ●V_2** ○V_2 view:1/2	○V_2 ●N_1,PP_1X view:2	○V_2 ●N_1,PP_1X ●V_2* view:2

2. Solution

HoneyBadgerBFT protocol

The first practical asynchronous BFT protocol. It can handle upto n/3 malicious node in the network

- HoneyBadgerBFT ensures liveness of the protocol without depending on any timing assumptions.
- It provides throughput which closely track the network's performance.
- It is robust against network attacks.

- Efficiency : O(N)*

*asymptotic communication complexity (bits per transaction, expected) for atomic broadcast protocol

Our contribution

We have added **HoneyBadgerBFT as an ordering service in Hyperledger Fabric** first asynchronous BFT consensus protocol implementation for Fabric

- Byzantine Fault Tolerant - now Fabric can be used where malicious nodes are present in the network.
- Asynchronous protocol - makes progress even when the underlying network is not stable.
- Liveness is always guaranteed.

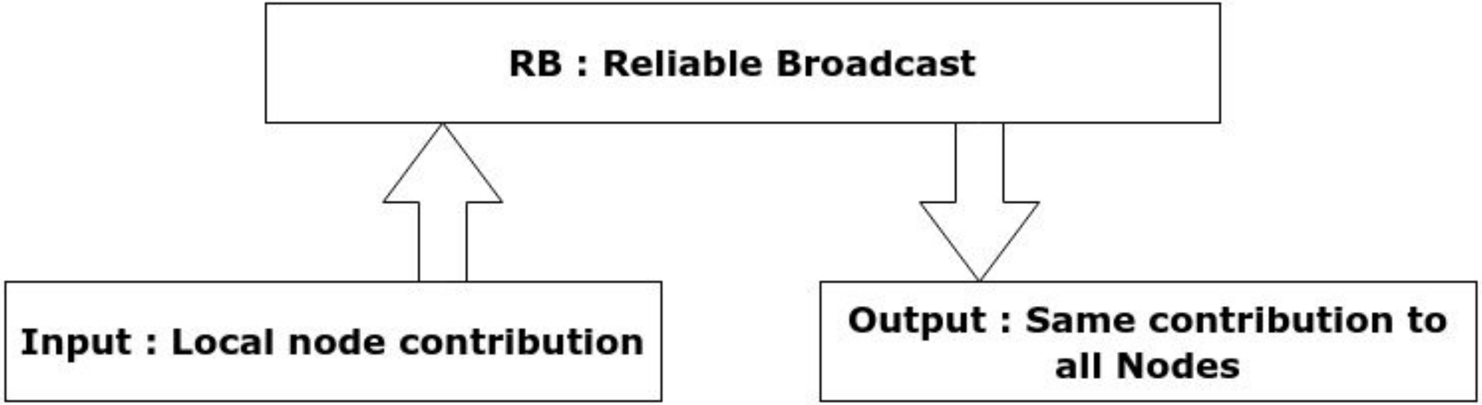
We also provide **stand-alone implementation of HoneyBadgerBFT** in Golang. It can used directly to add HoneyBadgerBFT as consensus in any permissioned blockchain.

Details of HoneyBadgerBFT

- We will explain protocol in a modular style, where each protocol may run several instances of other(sub)protocols.
- Protocol works in **Epochs**.
- Protocol have following modules:
  - Reliable Broadcast (RB)
  - Byzantine Binary Agreement (BBA)
  - Asynchronous Common Subset (ACS)
  - HoneyBadger (HB)
- In HoneyBagderBFT every node provides contribution for each epoch and out of these contribution final result of the epoch is selected.

Reliable Broadcast (RB) :

- **Erasure code** : transforms a message of k symbols into a longer message (code word) with n symbols such that the original message can be recovered from a subset of the n symbols.
- **Merkle tree :** allow efficient and secure verification of the contents of large data structures

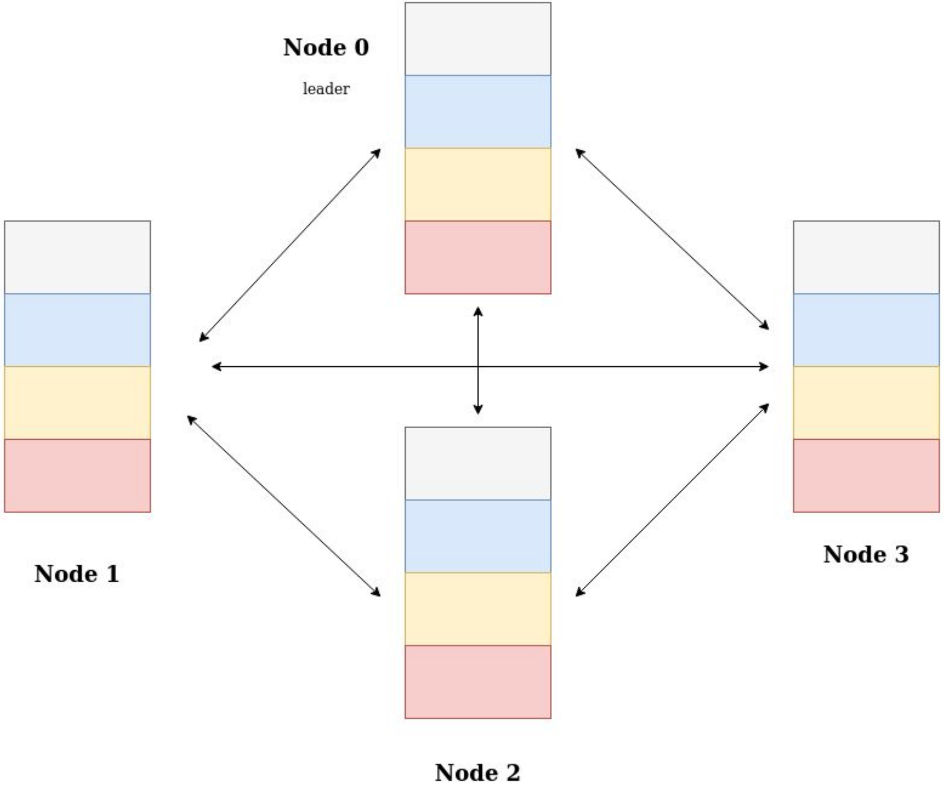


Algorithm Reliable Broadcast

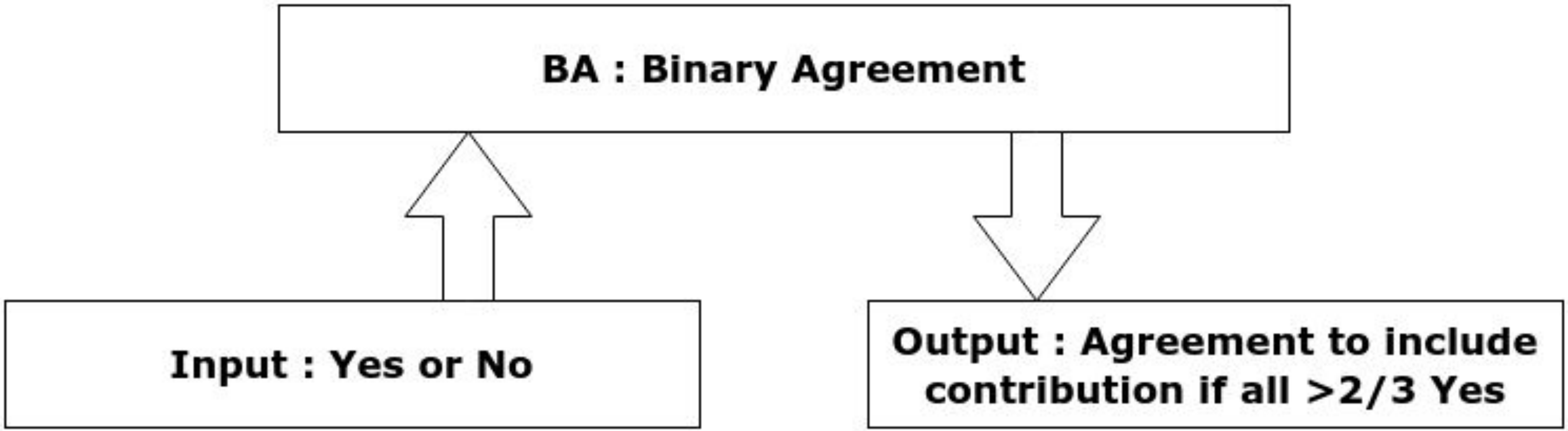
Let N be the total number of nodes
Let f be the number of faulty nodes
Let P_i be i[th] RB instance

// Step 1: input to Reliable Broadcast

- upon receiving input v if $P_i = N_i$
  - let $\{s_j\}_{j \in |N|}$ be the block of an $(parity, N)$ - erasure coding scheme applied to v
  - let h be a Merkle tree root computed over $\{s_j\}$
  - send $VAL(h,\ b_j,\ s_j)$ to each party P_j, where b_j is the j^{th} Merkle tree branch
- upon receiving $VAL(h,\ b_i,\ s_i)$ from P_{sender}
  - Multicast $ECHO(h, b_i, s_i)$
- upon receiving $ECHO(h,\ b_j,\ s_j)$
  - Check that b_j is a valid Merkle branch for root and leaf s_j, and otherwise discard
- upon receiving valid $ECHO(h,\ \cdot,\ \cdot)$ messages from $N - f$ distinct parties,
  - Interpolate $\{s'_j\}$ from any $N - 2f$ leaves received
  - recompute Merkle root h' and if $h' \neq h$ then abort
  - if $READY(h)$ has not yet been sent, multicast $READY(h)$
- upon receiving $f + 1$ matching $READY(h)$ messages, if $READY(h)$ has not yet been sent, multicast $READY(h)$
- upon receiving $2f + 1$ matching $READY(h)$ messages, wait for $N - 2f$ $ECHO$ messages, then decode v



Byzantine Binary Agreement (BBA)



Algorithm Binary Agreement (for node P_i)

Let r be the epoch number for BA instance
Let est_r be the estimated value for r^{th} epoch
Let $BVAL_r$ be a Binary value type message
Let AUX_r be an Auxiliary value type message
Let bin_value_r be a set of binary values

// Step 1: input to binary agreement

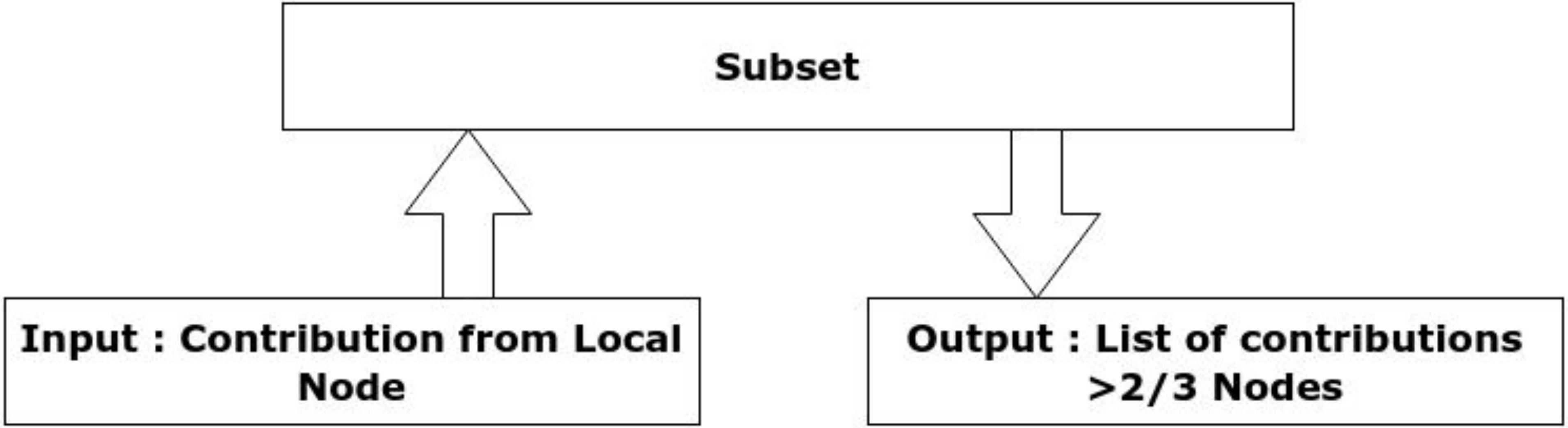
- upon receiving input b_{input} set $est_0 := b_{input}$

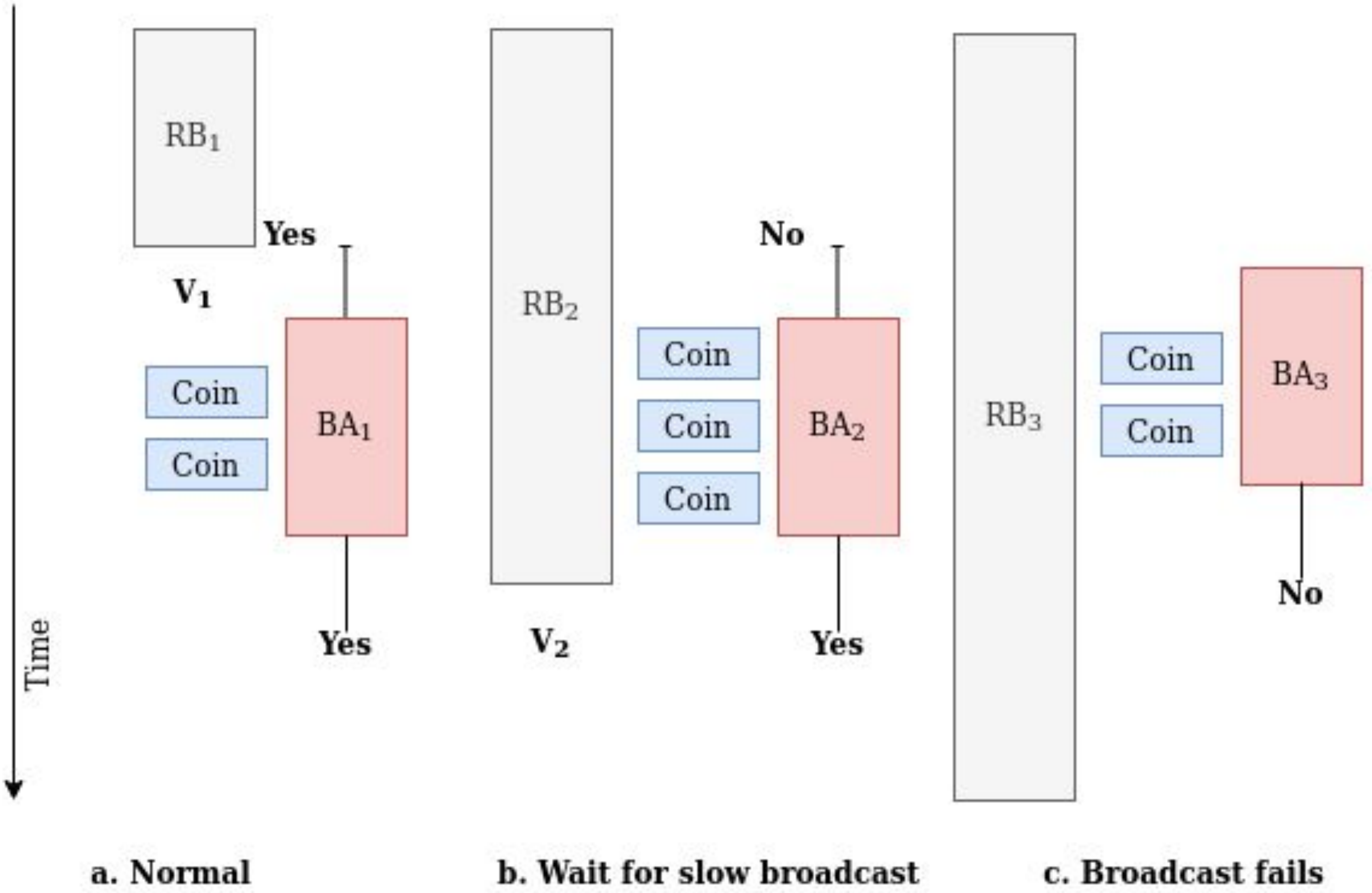
// Now algorithm proceeds in epochs

- multicast $BVAL_r(est_r)$
- set $bin\_value_r := \{\}$
- upon receiving $BVAL_r(b)$ message from $f+1$ nodes, if $BVAL_r(b)$ has not been sent, multicast $BVAL_r(b)$
- upon receiving $BVAL_r(b)$ message from $2f+1$ nodes, $bin\_value_r := bin\_value_r \cup \{b\}$

- wait until $bin\_value_r \neq \phi$, then
  - multicast $AUX_r(w)$, where $w \in bin\_values_r$
  - Wait until at least $(N-f)$ AUX_r messages have been received, such that the set of values carried by these messages, $vals$ are a subset of bin_value_r
  - $s \leftarrow r\%2$
  - If $vals = \{b\}$, then
    - $est_{r+1} := b$
    - If $(b = s)$ then output b
  - else $est_{r+1} := s$
- loop continues until both the value b is output in round r, and the value $s = b$.

Asynchronous Common Subset (ACS)





Time
RB1
Yes
V1
Coin
Coin
BA1
Yes
RB2
V2
Coin
Coin
Coin
No
BA2
Yes
RB3
Coin
Coin
BA3
No
a. Normal
b. Wait for slow broadcast
c. Broadcast fails

Algorithm Subset (for node P_i)

Let $\{RB_i\}_N$ refer to N instances of the reliable broadcast protocol, where P_i is the sender of RB_i.
Let $\{BA_i\}_N$ refer to N instances of the binary agreement protocol, where P_i is the sender of BA_i.
Let total number of nodes be N.
Let total number of fault nodes be f.

// Step 1: input to reliable broadcast and binary agreement

- upon receiving input proposal v_i from QHB, input v_i to RB_i.
- upon completion of RB_j, if input has not been provided to BA_j, then provide input 1 to BA_j

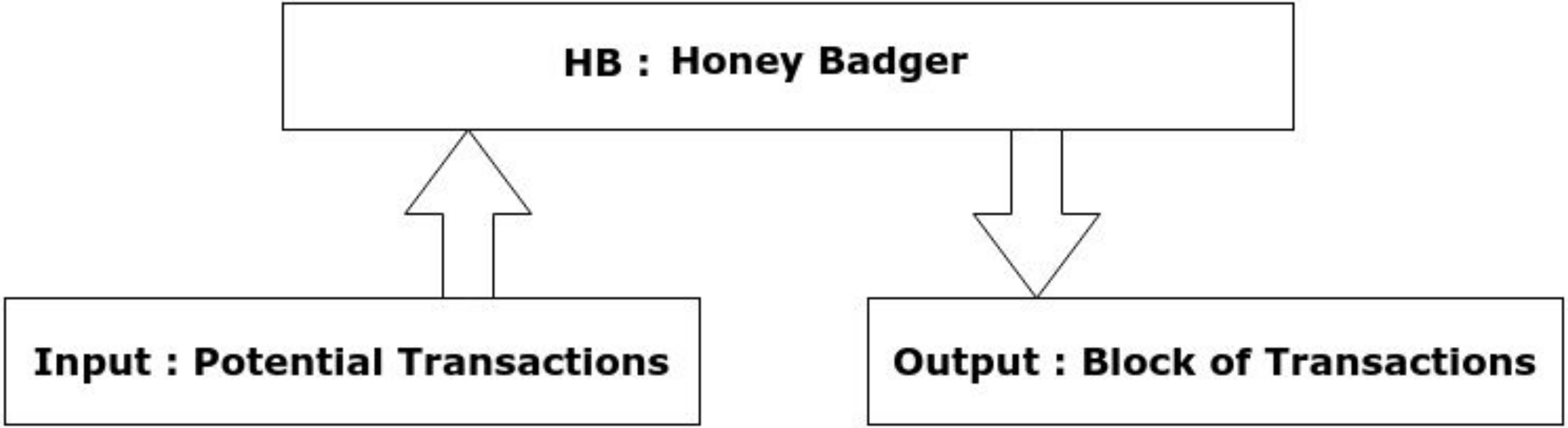
// Step 2: input to pending binary agreement

- Upon delivery of value 1 from at least $N - f$ instance of BA, provide input 0 to each instance of BA that has not yet been provided input.

//step 3: gather outputs and return

- once all instances of BA have completed, wait for the completion of $RB's$, if the corresponding $BA's$ output 1.
- Finally return the union of all RBs output if the corresponding BA output is 1.

HoneyBadger (HB)



Algorithm Queuing HoneyBadgerBFT (for node P_i)

Let batch size be B
Let total number of nodes N
Let buf be a FIFO queue of input transactions.

Algorithm proceeds in epochs numbered k:

//Step 1: Random selection of transaction

- For this epoch, this nodes proposes random selection of
- $proposal = B/N$ transaction from the first B of buf

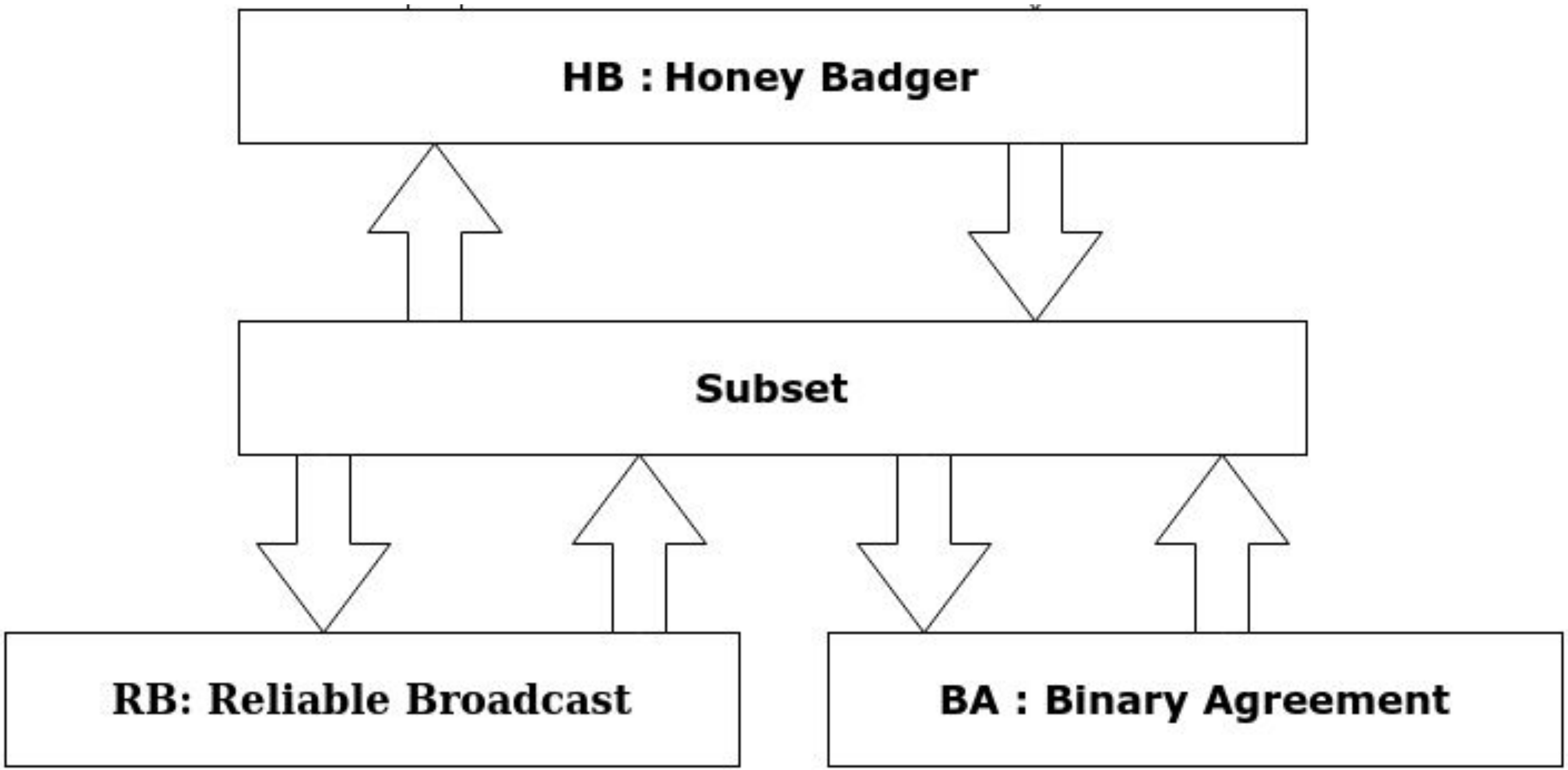
//Step 2: Agreement on proposal

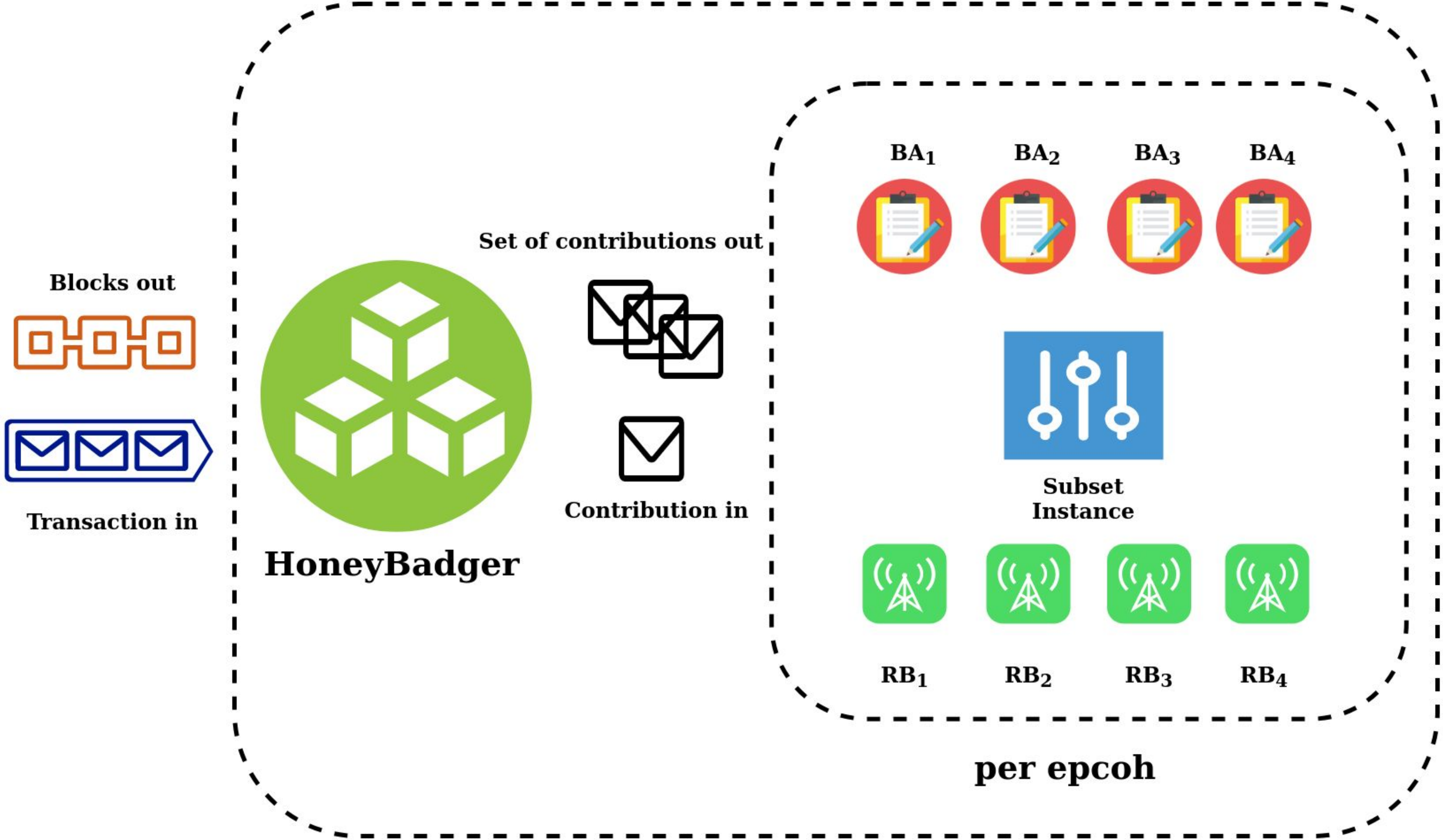
- pass $proposal$ as input to $Subset\,[k]$
- receive all the agreed up proposal from all the nodes, from $Subset\,[k]$

//Step 3: Making block from the proposals

- Let $block_r$:= sorted(agreed up proposals)
- set buf := $buf - block_r$

Overall


HB : Honey Badger
Subset
RB: Reliable Broadcast
BA : Binary Agreement



Blocks out
Transaction in
HoneyBadger
Set of contributions out
Contribution in
BA_1
BA_2
BA_3
BA_4
Subset Instance
RB_1
RB_2
RB_3
RB_4
per epcoh

Implementation Details:

1. We have implemented stand-alone version in **Golang.**
2. **2900** Lines of Code.
3. We achieved throughput of **2100 txn/sec** with 4 nodes.
4. The implementation is available on Github. Anyone can use the implementation as library.
5. API's Available are:
    a. **Enqueue(Transaction)** : to input the transaction
    b. **Output** : channel to get the output blocks
6. Experiment results are presented in detail at the end of the presentation

HoneyBadgerBFT in Fabric

- HoneyBadgerBFT is both Crash fault tolerant(CFT) and Byzantine fault tolerant (BFT).
- We introduce HoneyBadgerBFT as a new consensus model in Fabric v1.4
- Hyperledger Fabric supports pluggable consensus. So, other consensus models may be implemented in the future.
- We implement HoneyBadgerBFT in the consensus package of Fabric’s source code.

Integration details:

To add a new consensus protocol, need to implement the following interfaces functions:

- Consenter interface:–
  - HandleChain(ConsenterSupport, *cb.Metadata) (Chain, error) : This function creates and return a reference to a Chain.
- Chain interface:–
  - Order(env *cb.Envelope,configSeq uint64) (error) : This function is called by a chain to submit a normal transaction to order.
  - Configure(config *cb.Envelope, configSeq uint64) (error) : This function is called by a chain to submit a configuration transaction to order. This is used to make configuration changes in the orderer nodes.–
  - Start() : This is used to start the Chain.
  - Halt() : This is used to halt a running C

3. Result

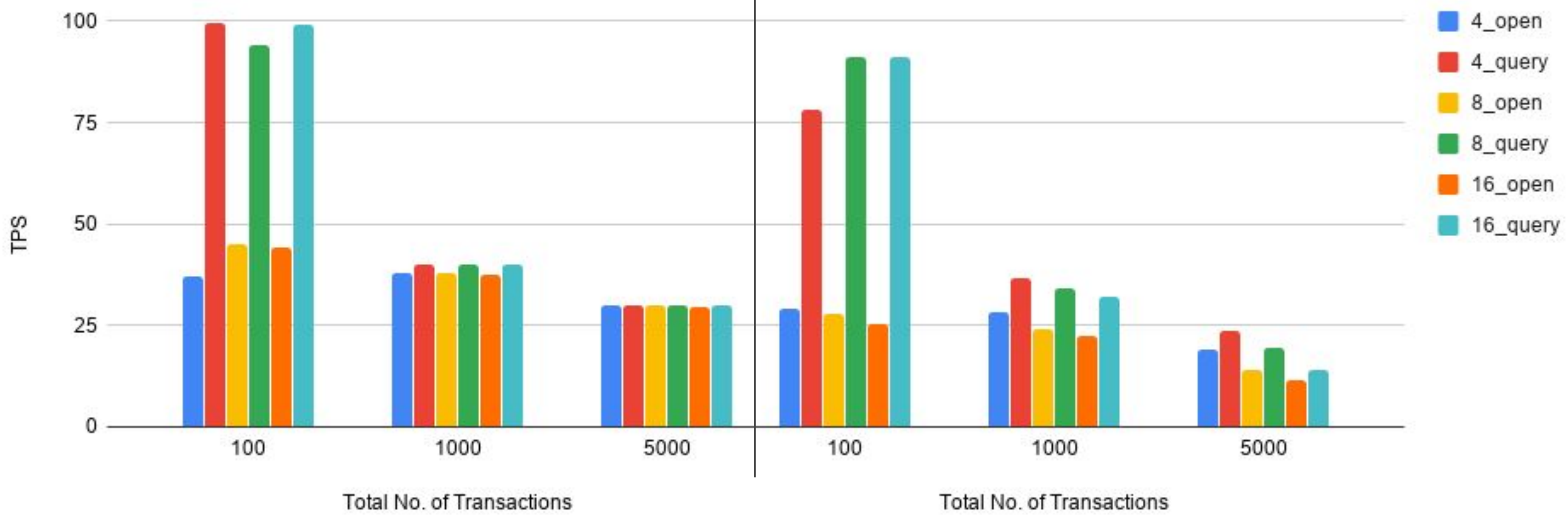
Results

For testing and benchmarking we have used:

- Hyperledger Caliper
    - A blockchain performance benchmark framework
    - Allows users to test different blockchain solutions with predefined use cases, and get a set of performance test results
- 4 VMs each with following configuration:
    - OS : Ubuntu 18.02
    - CPU : Intel Xeon v2 processor 2 cores
    - Memory : 8 GB

RAFT

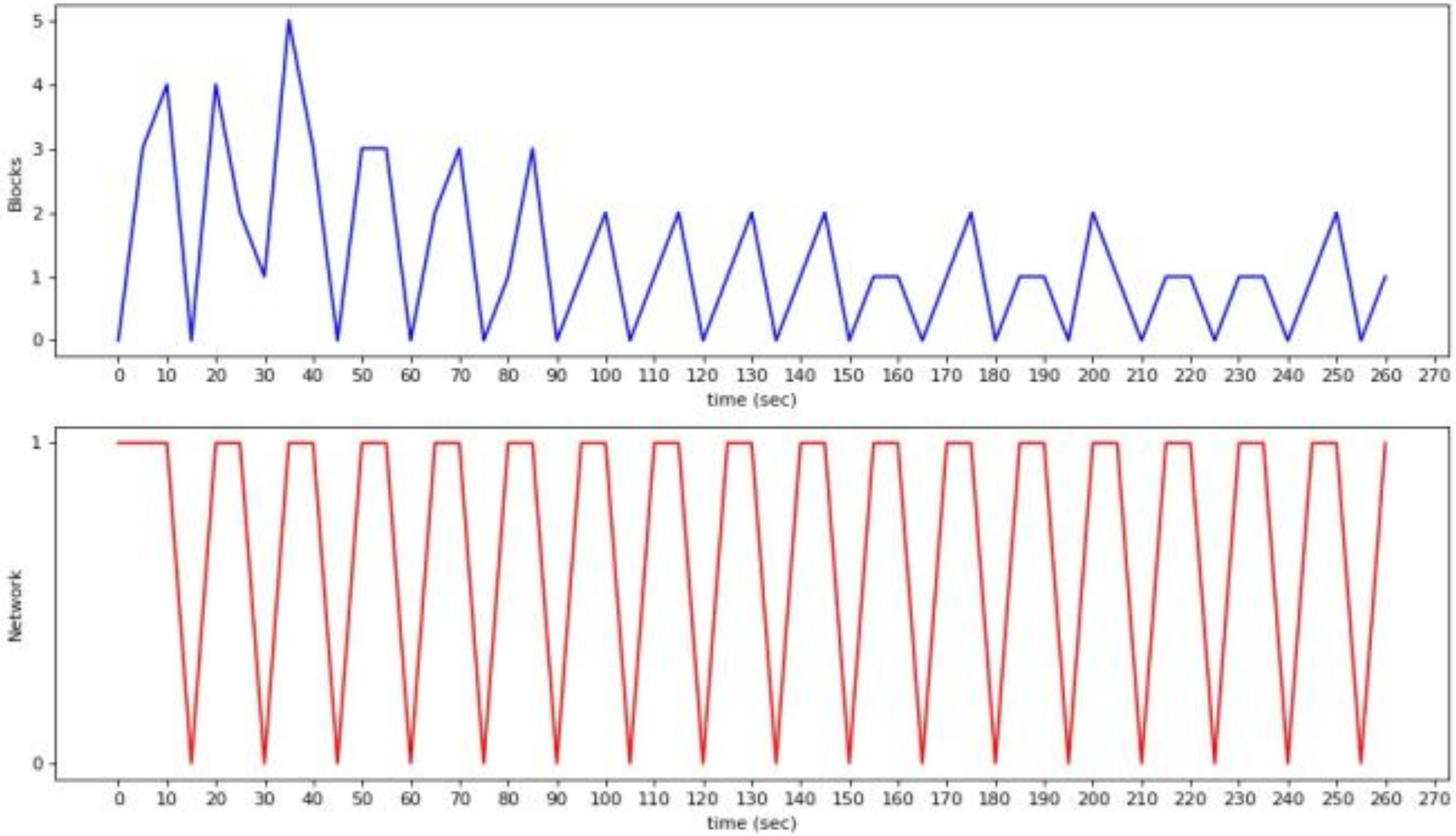
HoneyBadgerBFT

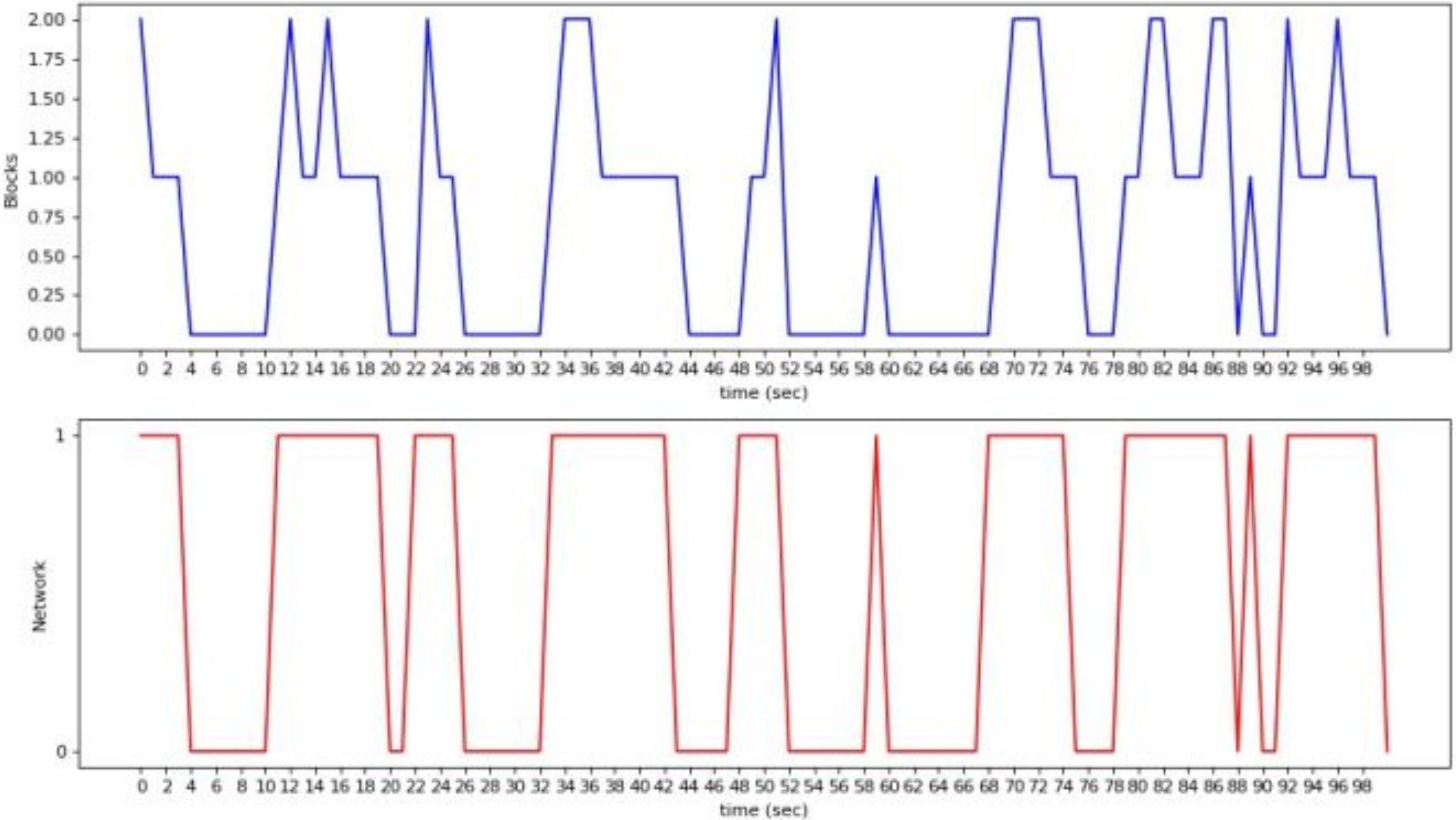


Resource utilization

No of Nodes	Orderer	Avg. Latency	Avg. CPU(max.) usages	Avg. RAM usages
4	Raft HoneyBadgerBFT	1.72 3.43	6.40 8.24	15.23 46.4
8	Raft HoneyBadgerBFT	1.61 5.42	3.99 8.13	29.05 44.3
16	Raft HoneyBadgerBFT	1.76 8.26	2.33 12.13	32.06 88.67

Test results of protocol : when network is unstable





2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Blocks
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98
time (sec)
1
0
Network
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98
time (sec)

4. Conclusion

Conclusion

- We provide implementation of HoneyBadgerBFT in Golang and HoneyBadgerBFT as an ordering service in Hyperladger Fabric v1.4.

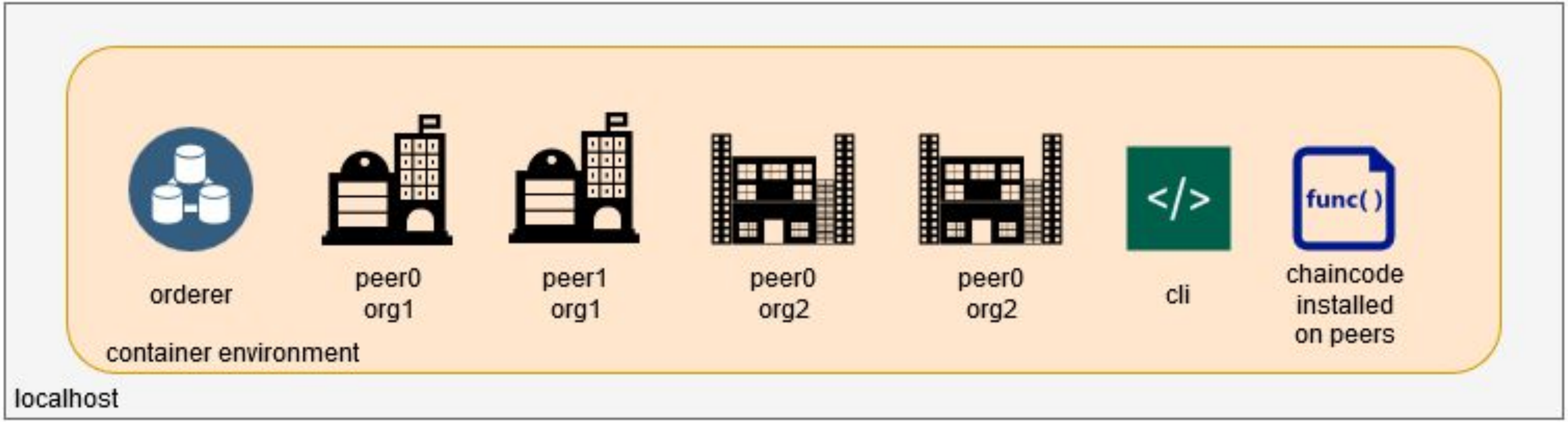
From the result of the benchmarks can conclude that:

- Throughput of HoneyBadgerBFT` is comparable (around 60% on average) to Raft.
- Which is acceptable as HoneyBadgerBFT is Byzantine Fault Tolerant.
- Resource utilizations are also comparable and are not too high.
- Which make HoneyBadger a suitable choice as an ordering service for Fabric when network may have malicious nodes and user can not provide guarantees for the underlying network

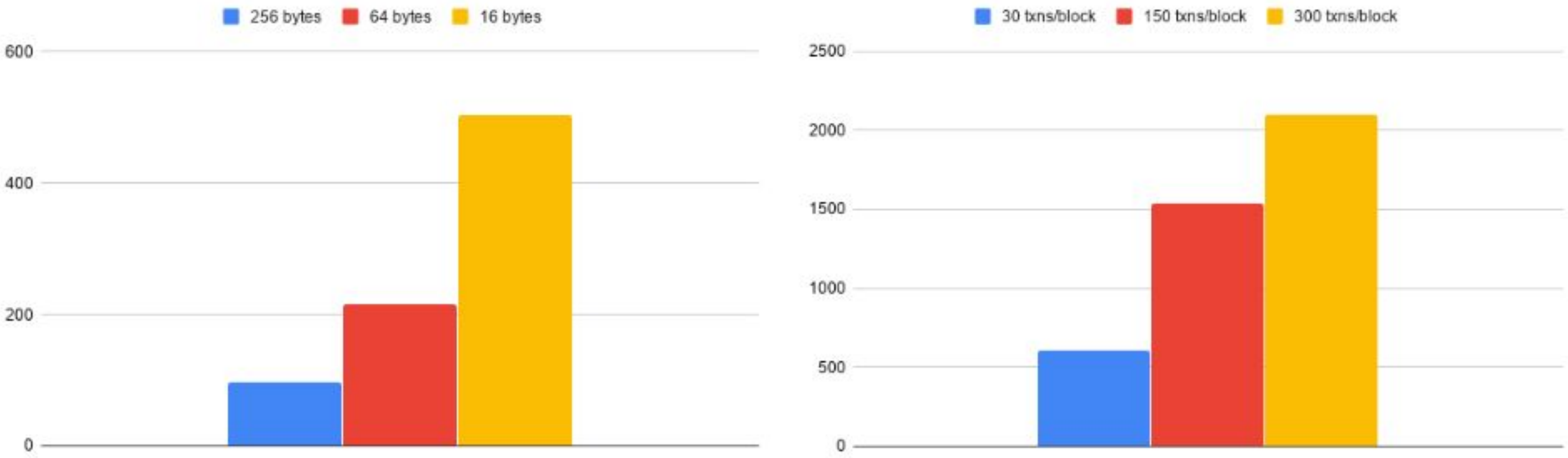
Questions?

Thank You

Container environment



Throughput for diff. Txn. sizes and Block size



(a) Different Transaction Sizes (b) Different Block Sizes

Consensus

- Decisions are generally taken by the leader or board of decision makers.
- no “leader” for decision making.
- For blockchain to make decisions, we need to come to a consensus using consensus mechanisms
- Set of N computational Nodes
- Agree on a block
- Two Properties:
    - Agreement
    - Validity: All transactions satisfy some validity conditions