ECS 265: The Honey Badger of BFT Protocols

By Kaiyue Li , Yang Yu, Ziqi Cheng, Zili Wang

Overview

- Introduction
- Background
- Protocol
- Experiment & evaluation

The New Challenge: The Crypto Environment

- **New Deployment Environment**
  - Cryptocurrencies have created a new, challenging environment for BFT.
  - Key Characteristics:
    - Wide-Area Network
    - Many Distrustful Nodes
    - Hostile & Unpredictable Network
      - Network links can be unreliable
      - Network speeds can change rapidly
      - Network delays may be adversarially induced
- **New Requirement for BFT protocol**
  - **Robustness** under adversarial, decentralized conditions, not speed
  - Favor **throughput** over latency

Network Models

- **Network Assumptions**
  - ***synchrony***
    - Every message sent is delivered at most a delay of Δ
  - ***partial synchrony***
    - Unknown-Δ: the protocol is unable to use the delay bound as a parameter
    - Eventually synchronous: Δ is only guaranteed to hold after some GST
  - ***weak synchrony (PBFT)***
    - Δ is varying, but eventually not grow faster than a poly. function of time
  - ***Asynchronous (HoneyBadgerBFT)***
    - Assumes message will eventually be delivered
    - It does not care when it will be delivered.
    - It must be correct even if they are delayed for a long time.
- **"In terms of feasibility**, timing-assumption protocols are equivalent…"

The “SOTA” & Its Hidden Flaw

- Before, almost all “practical” BFT protocols are *partial* or *weak synchrony*
  - Rely on timing assumptions to guarantee **liveness**
- They are **unsuitable** for the cryptocurrency-oriented application scenarios
- The paper presents two counter arguments that refute the premise below:
  - There’s a theoretical separation between the async and weakly scyn network models
  - Even the assumptions are met, weak synchrony protocol is slow to recover from a network partition, whereas the async one can make progress

[X] *Weak synchrony assumptions are unavoidable, since in any network that violates these assumptions, even asynchronous protocols would provide unacceptable performance.*

The GAP: The “Network Scheduler” Attack

- **The attacker: a Network Scheduler**
  - An adversary that controls the timing of all network messages.
  - Attacks on both PBFT and HoneyBadgerBFT
- **The attacker breaks the assumptions and halt PBFT (Liveness Failure)**
  - First, assume a single **crashed node**
  - When a **correct node** is the leader
    - Delay its messages just past the timeout -> repeated view changes
    - For each failed leader, PBFT *increases* its timeout value (and the scheduler delay messages longer and longer to force failures).
  - When the **crashed node** is the leader
    - The network scheduler “heals” the network
    - However, PBFT still cannot make any progress during this good period.

The GAP: Slow Recovery from Network Partitions

- **What if assumptions holds eventually?**
  - PBFT can be very slow to recover
  - Current Situation
    - One crashed node
    - Network partition lasts for $2^D\Delta$ (just like last slide)
    - Timeouts have been exponentially increased by earlier failures
  - Scheduler heals the network right when the crashed node becomes leader
    - Though network is already synchronous
    - PBFT still stuck waiting for a full extra $2^{D+1}\Delta$
- **Timeout-based weak synchrony protocols can waste long "good" periods after a partition**

The GAP: the tradeoff

- **Must balance the timeouts**
  - Extreme 1: Fixed Timeout (Eventual Synchrony)
    - If estimated timeout **too low** → frequent view changes and make no progress
    - If estimated timeout **too high** → wastes bandwidth / throughput
  - Extreme 2 Adaptive Timeout (Weak Synchrony):
    - Don’t fix an absolute Δ
    - There’s a need to tune a “gain” parameter
- **Solution (HoneyBadgerBFT):**
  - Do not depend on timeouts for liveness
  - Avoid the need to tune Δ or gain parameters, sidestepping this tradeoff.

Optimal asymptotic efficiency

- **HoneyBadgerBFT is the first practical BFT protocol to achieve:**
  - Asynchronous atomic broadcast protocol
  - Optimal asymptotic efficiency
  - Censorship-resistant

Table 1: Asymptotic communication complexity (bits per transaction, expected) for atomic broadcast protocols

	Async?	Comm. compl.	
		Optim.	Worst
PBFT	no	$O(N)$	∞
KS02 [34]	yes	$O(N^2)$	$O(N^3)$
RC05 [47]	yes	$O(N)$	$O(N^3)$
CKPS01 [15]	yes	$O(N^3)$	$O(N^3)$
CKPS01 [15]+ [9, 18]	yes	$O(N^2)$	$O(N^2)$
HoneyBadgerBFT (this work)	**yes**	**O(N)**	**O(N)**

4. Protocols

4.1 Atomic Broadcast: Problem Definition

a) Network model and the atomic broadcast problem

i) Goal: reach common agreement on an ordering of transactions between N nodes
ii) Atomic broadcast: allows us to abstract away any application-specific details
iii) No model clients, transactions are chosen by the adversary
iv) Committed after sent

4. Protocols

4.1 Atomic Broadcast

Assumptions

- N replicas
- Up to f < N/3 Byzantine nodes
- Asynchronous network
    - NO timing assumptions
    - Messages can be arbitrarily delayed
    - But messages between honest nodes are eventually delivered

4. Protocols

4.1 Atomic Broadcast:

Properties

- Agreement

If any honest node outputs transaction tx → all honest nodes must output tx.

- Total Order

All nodes output transactions in the same global order.

If Node A outputs:[tx1, tx2, tx3]. Then Node B cannot output:[tx1, tx3, tx2].

Protocols

4.1 Atomic Broadcast

Properties

- Agreement
- Total Order
- Censorship Resilience

If any transaction is submitted to ≥ N – f honest nodes,it must eventually be included in the log.

This is stronger than PBFT’s standard liveness.

4.1 Atomic Broadcast

Metrics(Efficiency)

- The communication cost per node over all transactions committed in an epoch.
- Formally:

  If each honest node has at least $\Omega(\text{poly}(N, \lambda))$ transactions queued, the efficiency is the expected communication cost per committed transaction per node.

4.1 Atomic Broadcast

Metrics(Transaction Delay)

- Transaction delay = $O(T/B + \lambda)$ epochs
- Expected number of asynchronous rounds required for a transaction to be committed, as a function of the backlog T.

Protocols

Challenge 1 — Fair batching without censorship

To achieve high throughput, nodes must propose *batches* of transactions.
But if nodes simply choose batches and broadcast them:

- Byzantine nodes (or the adversary) can see **which node proposed which transactions**
- ACS may choose which proposals to keep/drop
- **Adversarial nodes can strategically censor** specific transactions
  (e.g., exclude any proposal containing a targeted tx)

This breaks the **censorship resilience** property of atomic broadcast.

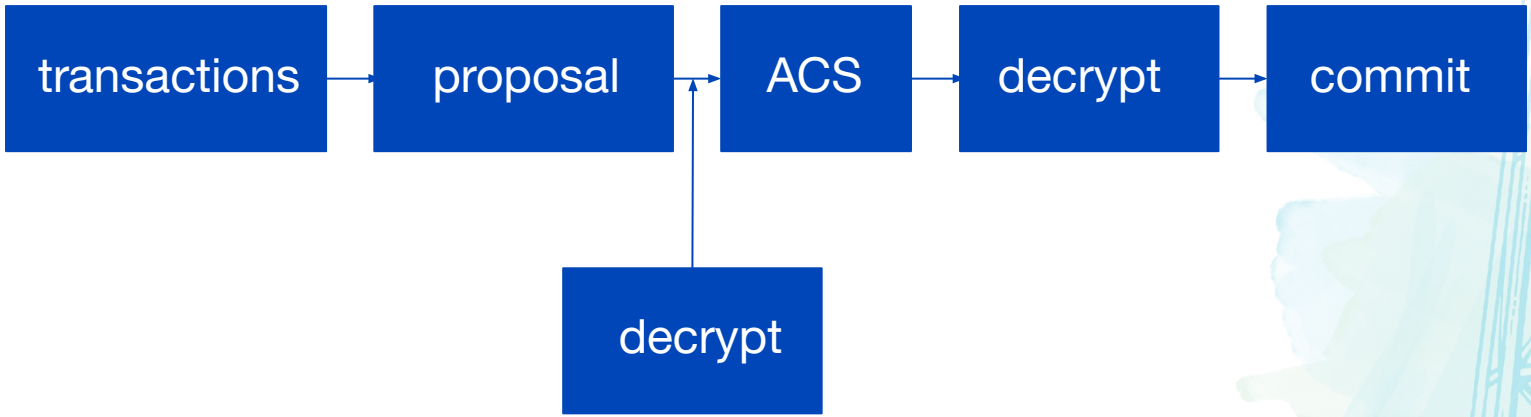
Challenge 2 — Practical throughput

If each node includes the first B/N transactions in its queue:

- Many nodes will choose **highly overlapping sets**
- Leads to **redundancy**, reducing throughput
- Some proposals may become identical
- But if nodes choose disjoint sets deterministically → adversary learns patterns → censoring possible

Protocols

Big Picture



Protocols

1. Each node chooses a random batch of transactions

Each replica:

- Looks at its local transaction pool.
- Selects ~B/N random transactions from the first B.

  -Random selection, not FIFO

- Encrypts them using threshold public-key encryption (TPKE).

Protocols

Challenge 2 solved

Random Sampling from the First B Transactions

Each node: Takes the first **B transactions** in its incoming queue. Randomly samples **B/N** items. Then encrypts the sample

- Randomization reduces overlap (“spread out” the proposals)
- Expected number of distinct transactions per epoch ≈ B/4
- Maintains fairness: every transaction among the first B has similar probability of being proposed

Protocols

Challenge 1 solved

Threshold Encryption (TPKE)

Randomly selects ~B/N transactions. **Encrypts** the batch under a threshold public key. Inputs the ciphertext into ACS. Only **after** ACS agrees on all ciphertexts do nodes reveal decryption shares.

- During ACS, no one knows the content of the proposals
- The adversary *cannot distinguish* which proposals contain a target tx
- Cannot censor selectively
- The outcome of ACS is predetermined before any plaintext is visible

Protocols

2.Each node proposes its encrypted batch into ACS

ACS = Asynchronous Common Subset protocol

- Every node inputs a ciphertext
- ACS ensures every node outputs the same set of ciphertexts
- Contains at least N – f correct proposals

This eliminates the need for a single leader, unlike PBFT.

Protocols

3. Once ACS is done, everyone decrypts together

- Using threshold decryption
- Needs f + 1 valid decryption shares
- Ensures that ciphertexts cannot be selectively decrypted early
- Prevents censorship

After decryption:

- Everyone merges the resulting transaction sets
- Sorts them deterministically
- Commits this as the next block

What makes PBFT work?

PBFT relies on waiting for time: If a message doesn't arrive within expected time, it assumes the leader failed

In real networks, delays are unpredictable. You can't know how long to wait for a message

Set timeout too short → Frequently false alarms → Constant leader re-elections → Throughput crashes

Set timeout too short → Frequently false alarms → Constant leader re-elections → Throughput crashes

yangyu

A giant leap : from partially asynchrony to fully asynchrony

- *Stop waiting for time. Wait for messages instead. As long as messages eventually arrive, timing doesn't matter.*

PBFT vs HoneyBadgerBFT

Aspect	PBFT	HoneyBadgerBFT
Analogy	Restaurant: customers wait a fixed time for dishes	Email system: you don't know when it arrives, but it eventually does
Waiting Mode	Wait based on a clock (Δ time units)	Wait for messages to arrive (no time-based waiting)
Fault Detection	If no message arrives within the "dish time window," treat it as a fault	Judged by message content itself, not by timeouts
Worst Case	If the network is slow → frequent false timeouts; timeout parameter is hard to tune	No matter how slow the network is, as long as messages eventually arrive, it works

Using an example

Step	How PBFT Does It	How HoneyBadgerBFT Does It
Step 1	The leader says: “I decide the order of these transactions.”	The leader and the other 99 nodes each propose a batch of transactions.
Step 2	Everyone waits 20 ms to see whether the messages arrive.	Messages are encrypted and processed through ACS (I’ll explain ACS shortly).
Step 3	If all required messages arrive, the system commits.	The protocol eventually produces a total order (a full ordering of all batches).
Step 4	If 20 ms passes with no message, leader is marked faulty → trigger view change.	No timeout → no leader-fault step here.
When network is bad	20 ms too short? Increase to 200 ms, but everything slows down → impossible to tune!	System naturally adapts to network speed; throughput adjusts automatically.

Innovation:Decentralized Proposals + Encryption

PBFT's Bottleneck:

If the leader is slow ,the whole system is slow.And pbft wait and change view from time to time.

HoneyBadgerBFT improvements: every node choose proposals from

Its own pool and do encryption ,and ACS .

Image a bad leader : I saw your proposal and I choose to delay it forever .

Techinical Details

- Each node proposes its batch using threshold encryption.
- As long as f + 1 nodes (where *f* is the maximum number of faulty nodes)
- provide their shares, the ciphertext can be decrypted.
- A malicious node cannot prevent the system from decrypting on its own.

HoneyBadger's Approach: Reduce Message Overhead with Erasure Coding

Take data block and break it into small fragments

Node A splits the 256KB block into **100 fragments**

Each fragment is encoded using **erasure coding**

A sends **one small fragment** to each of the 100 nodes

Every node receives only a **512-byte piece**

Any 67 fragments ($\geq 2f + 1$) are enough to reconstruct the full original data

Result: message count drops from 9,801 → 100!

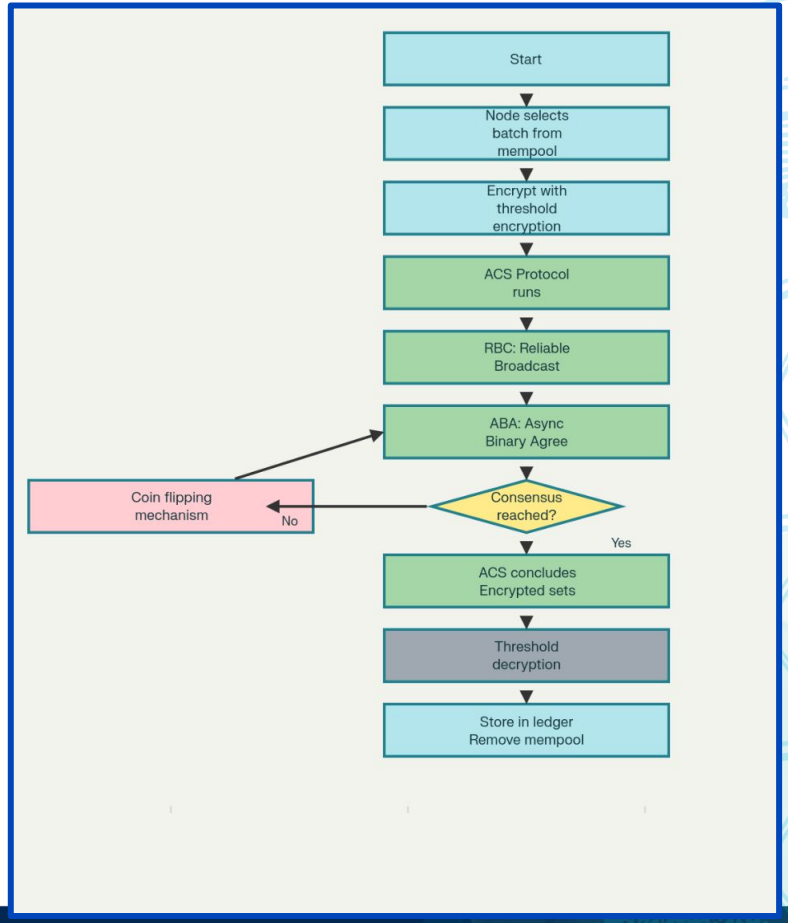
FLP assumptions recap

- **Asynchronous network** (no clocks, no time bounds) ✓
- **Deterministic algorithm** ✗ ← HoneyBadger drops this
- **Crash failures allowed** ✓

yangyu

flowChart

- RBC = broadcast, everyone gets the same message from a sender
- Atomic Broadcast = broadcast, everyone gets all messages, and in the exact same order



Step 1: Initial Broadcast

One person (let's call them Alice) wants to share a message with the whole group. Alice sends her message directly to everyone in the group (this is the "initial send").

Step 2: Relay (Echo) Messages

Everyone who receives Alice's message sends the exact same message to the whole group again.

This way, even if someone missed Alice's first message, they can still get it from someone else's echo—just like in a group chat where people forward an important notice to make sure no one misses it.

Step 3: Threshold Collection

When you've received the same message from a certain number of people (for example, from more than half or two-thirds of the group—whatever the system requires), you treat the message as valid and are ready to process it.

In other words: you only believe "Alice says there's no class tomorrow" once enough people in the group have said they got the same message from Alice.

Summary

Protocol	First-Class Citizen	Supports Node Recovery?	Recovery Mechanism
Raft	Log Entry	✅ Yes	Leader uses AppendEntries to resend missing logs
PBFT	Checkpoint	✅ Yes	Recovers from checkpoint + view-change
HoneyBadgerBFT	Epoch/Batch	❌ No	Paper provides no mid-epoch recovery mechanism

Core components

ACS: Asynchronous common subset

- (Validity) If a correct node outputs a set v, then |v|≥N–f and v contains the inputs of at least N–2 f correct nodes.
- (Agreement) If a correct node outputs v, then every node outputs v.
- (Totality) If N–f correct nodes receive an input, then all correct nodes produce an output.

After ACS, we have determined the set of transactions to process in this epoch, now decrypt the resort to get stable ordering

Algorithm HoneyBadgerBFT (for node $\mathcal{P}_i$)

Let $B = \Omega(\lambda N^2 \log N)$ be the batch size parameter.
Let PK be the public key received from TPKE.Setup (executed by a dealer), and let SK_i be the secret key for $\mathcal{P}_i$.
Let buf := [ ] be a FIFO queue of input transactions.
Proceed in consecutive epochs numbered r:

// Step 1: Random selection and encryption

- let proposed be a random selection of $\lfloor B/N \rfloor$ transactions from the first B elements of buf
- encrypt $x := \mathsf{TPKE.Enc}(\mathsf{PK}, \mathsf{proposed})$

// Step 2: Agreement on ciphertexts

- pass x as input to $\mathsf{ACS}[r]$ *//see Figure 4*
- receive $\{v_j\}_{j \in S}$, where $S \subset [1..N]$, from $\mathsf{ACS}[r]$

// Step 3: Decryption

- for each $j \in S$:
  - let $e_j := \mathsf{TPKE.DecShare}(\mathsf{SK}_i, v_j)$
  - multicast $\mathtt{DEC}(r, j, i, e_j)$
  - wait to receive at least $f + 1$ messages of the form $\mathtt{DEC}(r, j, k, e_{j,k})$
  - decode $y_j := \mathsf{TPKE.Dec}(\mathsf{PK}, \{(k, e_{j,k})\})$
- let $\mathsf{block}_r := \mathsf{sorted}(\cup_{j \in S}\{y_j\})$, such that block_r is sorted in a canonical order (e.g., lexicographically)
- set $\mathsf{buf} := \mathsf{buf} - \mathsf{block}_r$

ACS = RBC + ABA

- RBC (Reliable broadcast)
  - Broadcasts honest proposal
- ABA
  - Decide whether they should wait for a RBC to complete (no idea of time-out in asynchronous model)

Algorithm ACS (for party $\mathcal{P}_i$)

Let $\{\mathsf{RBC}_i\}_N$ refer to N instances of the reliable broadcast protocol, where $\mathcal{P}_i$ is the sender of RBC_i. Let $\{\mathsf{BA}_i\}_N$ refer to N instances of the binary byzantine agreement protocol.

- upon receiving input v_i, input v_i to RBC_i // *See Figure 2*
- upon delivery of v_j from RBC_j, if input has not yet been provided to BA_j, then provide input 1 to BA_j. *See Figure 11*
- upon delivery of value 1 from at least $N - f$ instances of BA, provide input 0 to each instance of BA that has not yet been provided input.
- once all instances of BA have completed, let $C \subset [1..N]$ be the indexes of each BA that delivered 1. Wait for the output v_j for each RBC_j such that $j \in C$. Finally output $\cup_{j \in C} v_j$.

RBC: reliable broadcast

- VAL
  - Share the blocks
- Echo
  - Ensures the same h (Byzantine node could not broadcast different content)
  - Ensures that the majority has seen the same h by waiting for N - f
- Ready
  - Shares that consensus with others

Algorithm RBC (for party $\mathcal{P}_i$, with sender $\mathcal{P}_{\text{Sender}}$)

- upon input(v) (if $\mathcal{P}_i = \mathcal{P}_{\text{Sender}}$):
  - let $\{s_j\}_{j\in[N]}$ be the blocks of an $(N-2f, N)$-erasure coding scheme applied to v
  - let h be a Merkle tree root computed over $\{s_j\}$
  - send $\texttt{VAL}(h, b_j, s_j)$ to each party $\mathcal{P}_j$, where b_j is the j^{th} Merkle tree branch
- upon receiving $\texttt{VAL}(h, b_i, s_i)$ from $\mathcal{P}_{\text{Sender}}$,
  - multicast $\texttt{ECHO}(h, b_i, s_i)$
- upon receiving $\texttt{ECHO}(h, b_j, s_j)$ from party $\mathcal{P}_j$,
  - check that b_j is a valid Merkle branch for root h and leaf s_j, and otherwise discard
- upon receiving valid $\texttt{ECHO}(h, \cdot, \cdot)$ 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 $\texttt{READY}(h)$ has not yet been sent, multicast $\texttt{READY}(h)$
- upon receiving $f+1$ matching $\texttt{READY}(h)$ messages, if READY has not yet been sent, multicast $\texttt{READY}(h)$
- upon receiving $2f+1$ matching $\texttt{READY}(h)$ messages, wait for $N-2f$ ECHO messages, then decode v

ABA: Asynchronous binary agreement

- BVAL
  - Broadcast current decision
  - Get consensus
- AUX
  - Share consensus
- Coin
  - If consensus on both {0, 1}
  - Both is acceptable
  - Next round estimate the same one
- If output with b, don't stop until the next coin = b. First b ensures that the vals = {b}. Next one ensures output

Algorithm BA (for party $\mathcal{P}_i$)

- upon receiving input b_{input}, set $\text{est}_0 := b_{\text{input}}$ and proceed as follows in consecutive epochs, with increasing labels r:
  - multicast $\texttt{BVAL}_r(\text{est}_r)$
  - $\text{bin\_values}_r := \{\}$
  - upon receiving $\texttt{BVAL}_r(b)$ messages from $f+1$ nodes, if $\texttt{BVAL}_r(b)$ has not been sent, multicast $\texttt{BVAL}_r(b)$
  - upon receiving $\texttt{BVAL}_r(b)$ messages from $2f+1$ nodes, $\text{bin\_values}_r := \text{bin\_values}_r \cup \{b\}$
  - wait until $\text{bin\_values}_r \neq \emptyset$, then
    - multicast $\texttt{AUX}_r(w)$ where $w \in \text{bin\_values}_r$
    - wait until at least $(N-f)$ $\texttt{AUX}_r$ messages have been received, such that the set of values carried by these messages, vals are a subset of bin_values_r (note that bin_values_r may continue to change as $\texttt{BVAL}_r$ messages are received, thus this condition may be triggered upon arrival of either an $\texttt{AUX}_r$ or a $\texttt{BVAL}_r$ message)
    - $s \leftarrow \text{Coin}_r.\text{GetCoin}()$ *// See Figure 12*
    - if $\text{vals} = \{b\}$, then
      - $\text{est}_{r+1} := b$
      - if $(b = s\%2)$ then output b
    - else $\text{est}_{r+1} := s\%2$
- continue looping until both a value b is output in some round r, and the value $\text{Coin}_{r'} = b$ for some round $r' > r$

Coin

- Any f+1 group should see the same output
- Threshold signature ensures that at each iteration r, they would see the same signature, thus same coin result.
- Ensures O(1) expected time for ABA consensus

Algorithm $\mathsf{Coin}_{\mathsf{sid}}$ **for party** $\mathcal{P}_i$

sid is assumed to be a unique nonce that serves as "name" of this common coin

- (Trusted Setup Phase): A trusted dealer runs $\mathsf{pk}, \{sk_i\} \leftarrow \mathsf{ThresholdSetup}$ to generate a common public key, as well as secret key shares $\{\mathsf{sk}_i\}$, one for each party (secret key sk_i is distributed to party $\mathcal{P}_i$). Note that a single setup can be used to support a family of Coins indexed by arbitrary sid strings.
- on input GetCoin, multicast $\mathsf{ThresholdSign}_{\mathsf{pk}}(\mathsf{sk}_i, \mathsf{sid})$
- upon receiving at least $f+1$ shares, attempt to combine them into a signature:

  $\mathsf{sig} \leftarrow \mathsf{ThresholdCombine}_{\mathsf{pk}}(\{j, s_j\})$

  if $\mathsf{ThresholdVerify}_{\mathsf{pk}}(\mathsf{sid})$ then deliver sig

Experiments

- Implementation details
  - Threshold encryption: 256 bit ephemeral key + AES 256 for the actual payload. (Faster the computation by using symmetric encryption for the actual payload)
  - Original assumptions about unbounded buffer could be achieved by adding a watermark once 75% of the buffer is full
  - N = 4f & each propose B/N
  - Running time:initial time to (N-f)th node outputs a value

Experiments

- When batch size is small, the additive overhead dominates, but once batch size reach 16384, the transaction dependent portion begins to dominate.

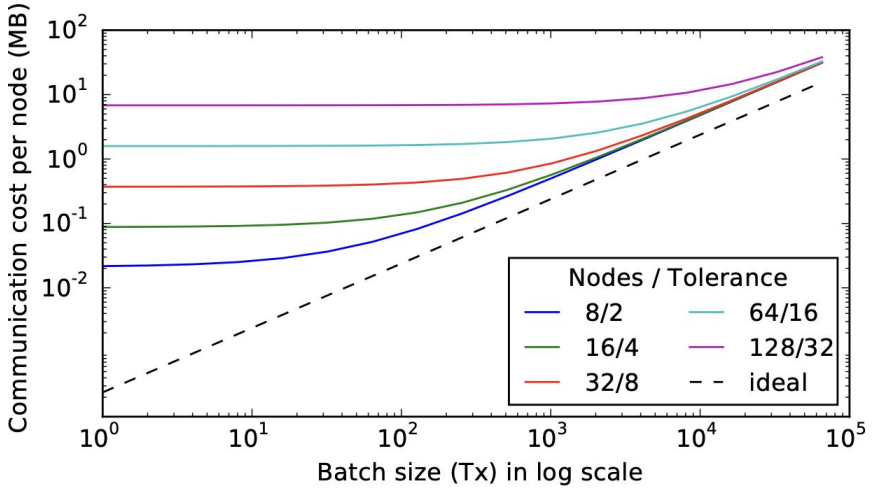


Figure 5: Estimated communication cost in megabytes (per node) for varying batch sizes. For small batch sizes, the fixed cost grows with $O(N^2 \log N)$. At saturation, the overhead factor approaches $\frac{N}{N-2f} < 3$.

Experiments

- Experiments settings
  - 32, 40, 48, 56, 64, 104 Amazon EC2 instance in 8 regions across 5 continents
  - Batch size: 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, or 131072
  - Measure unit: confirmed transactions per second
  - Fault tolerance f = N / 4

Experiments

The scalability of nodes is still an issue as adding nodes adds latency with O(N^2 logN)

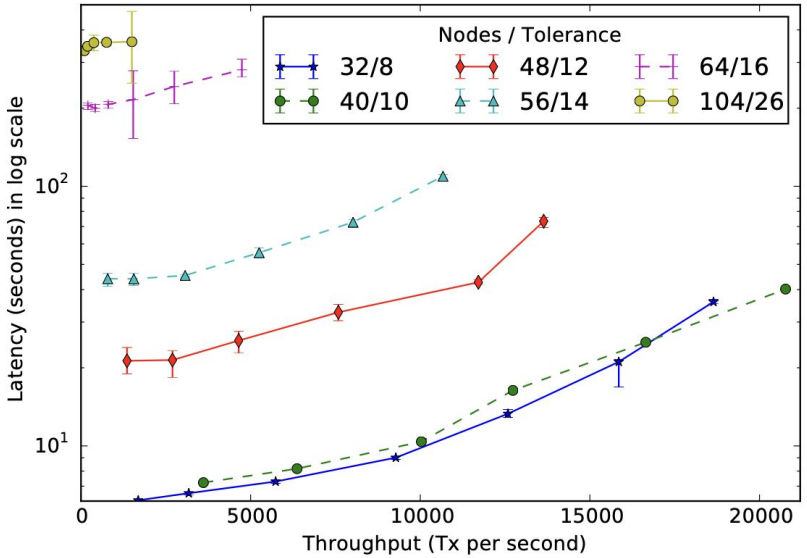


Figure 7: Latency vs. throughput for experiments over wide area networks. Error bars indicate 95% confidence intervals.

Experiments

- Tor experiment set up
  - Single machine with N hidden services
  - 5 random relays and N^2 tor networks
  - Significant, high variable latency

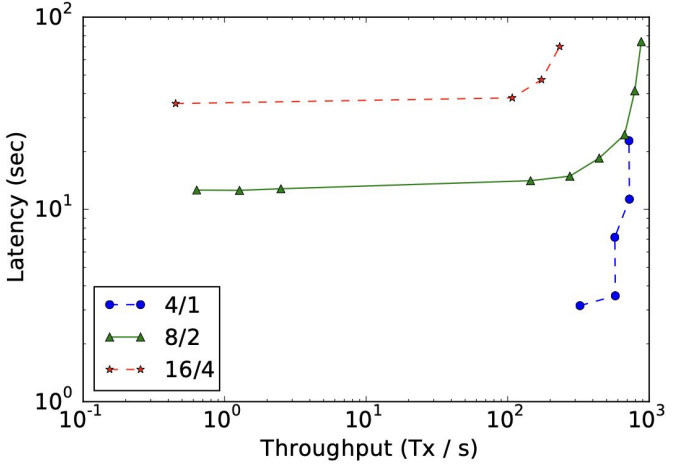


Figure 9: Latency vs throughput for experiments running HoneyBadgerBFT over Tor.

Thank you!