Caroline Berney, Shibali Mishra, and Ulemj Munkhtur
Sean Barker
CSCI 3325
18 December 2025

honeycomb

INTRODUCTION

Natural disasters such as hurricanes, earthquakes, tornadoes, and wildfires can disrupt reliable communication. Cell towers can be damaged, power can fail, and surviving networks often become congested as demand spikes. That failure is not just inconvenient—it can be life-threatening. People need a way to request medical help, food, and shelter, to coordinate with neighbors, and to reach loved ones. At the same time, responders and community groups need a way to broadcast time-sensitive updates like evacuation routes, road closures, or where supplies are available.

honeycomb is our response to that gap: a lightweight peer-to-peer communication platform designed for disaster environments. Instead of relying on a central server, honeycomb runs as a distributed network where every participant is both a client and a relay, avoiding a single point of failure or bottleneck. Users broadcast short, text-only messages that propagate through the network even when nodes join, leave, or crash. Our core priorities are availability during high churn, low overhead suitable for unreliable, low-bandwidth conditions, and scalability as the number of users grows. In this paper, we describe honeycomb's Chord-based ring architecture, the design choices that make it self-healing and fault-tolerant, and experiments that evaluate recovery time and delivery reliability.

RELATED WORK

honeycomb is most directly inspired by Chord[1], which organizes nodes into a logical ring using consistent hashing and keeps the ring intact under churn with periodic maintenance like stabilize, fix_fingers, and check_predecessor. We borrow Chord's ring architecture and the basic "repair in the background" approach because it provides a lightweight structure that can survive failures without any central coordinator. At the same time, honeycomb is not a full DHT: we are not doing general key lookups or storing (key, value) data. The only lookup technique we rely on is find_first_successor, mainly to place joining nodes and to maintain a small amount of routing structure. We also simplify Chord's routing state by keeping a single finger pointer rather than a full finger table, since our goal is lightweight, resilient broadcast rather than guaranteed logarithmic lookup performance.

honeycomb also has similarities to Gnutella[2], an early peer-to-peer system where information spreads through broad forwarding rather than a carefully structured routing layer. We do not use classic flooding because it scales poorly and wastes bandwidth, but we do adopt the same core idea that, under failures, relying on only one path is fragile. That is why honeycomb forwards messages probabilistically to multiple successors and deterministically to the finger, creating redundant paths that ensure dissemination even when neighboring nodes have crashed.

SYSTEM DESIGN

Ring Topology

honeycomb is organized as a Chord-style logical ring built using consistent hashing. Each node in the network has an ID, created by hashing (via SHA-1) its IP and port number and modding by a predefined constant m (discussed in implementation section). The ring is defined by clockwise successor order: a node's successor is the first node encountered moving clockwise in the identifier space. This ordering is maintained using local pointers rather than global membership.

To keep the state small, each node stores only a predecessor, a 5-entry successor list (its five immediate clockwise neighbors), and a finger. The successor list provides fault tolerance by allowing multiple immediate fallbacks when a successor fails. The finger is the first successor of the ID equal to ((node's ID) + 2^{m-1}) mod 2^m, which is halfway across the ID space. Therefore, the finger is a long-range contact which is used to accelerate placement/routing and reduce reliance on purely local links, without the overhead of a full finger table. The predecessor enables honeycomb to maintain successor lists via stabilization.

A node's successor list, predecessor, and finger are maintained through frequent periodic calls to stabilize, notify, check_predecessor, and fix_finger. Therefore, this data can become temporarily stale. We accept this tradeoff because eventual consistency allows us to maintain the network despite churn. In particular, as long as a node has at least one alive successor, it can broadcast messages to the network.

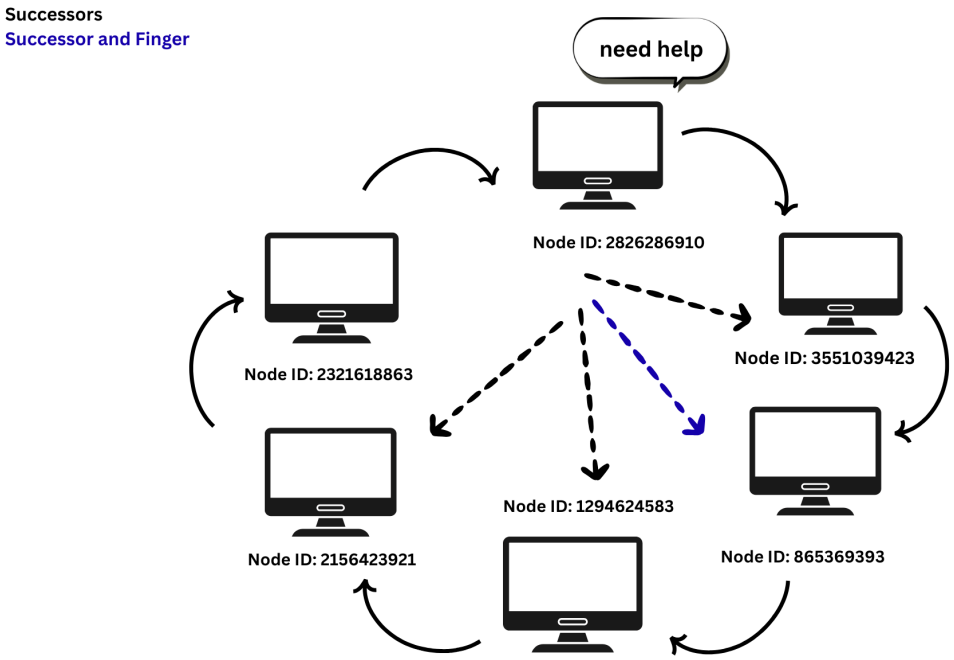

Figure: honeycomb's ring structure

IMPLEMENTATION

Ring Maintenance

In honeycomb, the ring topology is preserved through periodic updates to successor lists, predecessors, and fingers. These updates run approximately every second to ensure the network continues to function smoothly, even when nodes join, leave, or fail.

Create

A node creates its own honeycomb network using create, sets itself as its only successor and both predecessor and finger to None.

Join

A node can join an existing honeycomb network via an existing node using join. The node creates its ID via hash and asks the existing node to find its first successor. It sets its successor_list to contain only that node, and then contacts it to obtain any honeycomb messages that it has in memory.

Periodic Updates:

- **Stabilize** updates the successor list, ensuring the ring stays intact even if nodes fail or new ones join. First, it checks for a new possible successor by getting its first successor's

predecessor. If that node's ID is between its own ID and its successor's ID, then it becomes the node's new first successor. Second, it reconciles the node's successor list with that of its first successor, regardless of whether it was updated in step 1, falling back to other successors, finger, or predecessor in case of failure. If a node is no longer connected to any of these, it is killed, and the user is prompted to rejoin the network. Finally, it notifies its first successor of its existence, which keeps the successor's predecessor updated.

- **Notify** is called from within stabilize, and lets a node know that it may have a new predecessor. If a node doesn't have a predecessor, or the calling node is between its predecessor and itself, it sets that calling node as its new predecessor.
- **Fix Fingers** keeps the finger pointer updated by finding the successor for the ID located halfway around the ring, as mentioned above.
- **Check Predecessors** pings the predecessor and sets it to None if it has failed.

Reconciling Successor Lists

Reconcile(n) is called from within stabilize, and updates a node's successor list to be the first four elements of n's successor list, prepended with n.

Lookups

Lookups are only performed in the form of finding the appropriate successor to a given ID in the find_first_successor function. Given node_id, find_first_successor checks if node_id is between this node's ID and its first successor's ID. If it is, then its appropriate successor is this node's successor. Otherwise, the call is recursively forwarded to its first successor, or its other successors, finger, or predecessor in case of failure. We use the finger here as a potential speedup: if node_id is after our finger but before our ID in the ID space, then we forward the call to finger, approximately halving the runtime.

Message Sending & Receiving

honeycomb represents each broadcast as a single serialized record: (messageID, UTC timestamp, senderID, content). When a node receives a message, it checks whether the message ID has been seen—that is, if the message ID is in its messages_set. If yes, it drops the message immediately; if no, it stores the ID, inserts the message into its time-ordered display list, and forwards it onward.

Trade-off: This design enables deduplication while avoiding global coordination (no central server or global ordering protocol) and remains available under churn, but delivery is best-effort: during instability, a node may temporarily miss some broadcasts.

Probabilistic Sending

To account for the possibility that the first successor may have failed, instead of sending the message to only one node, it is deterministically forwarded to the first successor and the finger as

well as probabilistically forwarded to subsequent successors with logarithmically decreasing probability:

- **Successor 1:** 100%
- **Successor 2:** 50%
- **Successor 3:** 25%
- **Successor 4:** 12.5%
- **Successor 5:** 6.25%
- **Finger:** 100%

The successor list provides short, redundant forwarding paths even when nearby nodes fail, while finger forwarding provides long-range jumps that accelerate dissemination across the ring and bypass ring-section failures.

Trade-off: honeycomb trades additional bandwidth and slight latency for significantly higher delivery reliability under failure. It relies on deduplication to prevent unbounded forwarding.

Concurrency
honeycomb prioritizes availability: a node must receive and forward messages while also repairing ring pointers and interacting with the user. Each node therefore runs as a concurrent process with (i) a multi-threaded RPC server and (ii) background maintenance threads.

Each node creates the following threads:

- **RPC server thread:** runs the XML-RPC server loop. The server spawns a new worker thread per incoming RPC, so one slow or stalled request does not block other message deliveries.
- **Ring maintenance threads:** periodic background loops for stabilize, check_predecessor, and fix_fingers. These run roughly once per second with small randomized jitter to avoid synchronized maintenance bursts across the network.
- **UI watcher thread:** monitors newly stored messages and refreshes the terminal output without blocking network handling.
- **Main thread:** keeps the process alive and handles user commands (send/list/info/ring).

honeycomb uses a class-level lock to protect shared state (the successors, predecessor, finger, message list, message set, and flags) by locking their getter and setter methods.

Trade-off: Threading improves availability and reduces latency, but it increases synchronization complexity and the risk of races.

TCP vs UDP
honeycomb uses XML-RPC, which runs over TCP.

Trade-off: While TCP introduces some additional overhead compared to UDP, we made this decision because implementing our own transport layer over UDP would significantly increase system complexity.

IDs via SHA-1 Hashing
honeycomb maps each node's {IP:port} to an ID by SHA1 hashing and modulo into an m-bit space. We use M_BIT = 32 ($N = 2^{32}$) to keep identifiers small while making collisions negligible. The Birthday Paradox[3] estimates the following chance of collision:

$$P[\text{any collision}] \approx 1 - \exp\left(-\frac{n(n-1)}{2N}\right) \approx \frac{n(n-1)}{2 \cdot 2^{32}}.$$

For $n = 70$ nodes, $P \approx 5.6 \times 10^{-7}$ (≈ **1 in 1.8 million**).

Flushing
Nodes keep (i) an in-memory, time-ordered message list for display and (ii) a messageID set for deduplication. To prevent unbounded in-memory growth of these structures, a node triggers a flush once the message list exceeds 200 messages. It writes the oldest 150 messages (out of the 200) to disk in an append-only format and removes the corresponding flushed IDs from the in-memory deduplication set.

Trade-off: Flushing keeps memory usage stable, but dropping old IDs means that extremely delayed duplicates of flushed messages may be re-accepted and re-displayed. honeycomb accepts this rare edge case in exchange for bounded memory and sustained availability, in addition to reducing active load on user devices.

EVALUATION

Note: Most runs had EC2 machines timing out frequently due to overload, performance across experiment trials varied widely.

Experiment 1: Network Recovery from Node Failures

Experiment Setup

- **Network Formation:**
  - A network of 70 nodes was created from a pool of EC2 machines. One node initialized the Chord ring, with the remaining nodes joining by contacting randomly selected joiners, mimicking a decentralized join behavior.

- **Stabilization:**
    - After all nodes joined, the network was allowed 10 minutes (600 seconds) of idle time for stabilization. This period allowed successor pointers to attempt to converge.
- **Failure Injection:**
    - Progressive node failures were introduced by killing random nodes at different failure percentages: 8%, 15%, 29%, and 42%. For each failure percentage, the system's ability to recover was tested by observing the time taken for the Chord ring to converge after a set of nodes were killed. Failure injection is indicated with red line on table
- **Recovery Measurement:**
    - The convergence time was defined as the time it took for the network to reach a state where the largest partition contained a supermajority (80%) of all the living nodes in the network, and the system was stable. For our purposes, we defined that as the moment when the network satisfied the following criteria:
        - **Responding Nodes:** The number of nodes that are responsive and participating in the network after the failure injection.
        - **Stable Topology:** Convergence is reached when the network's topology remains stable for 5 consecutive polls, with no new partitions forming and no further changes to the largest partition.

Results

Trial	F%	Successfully Launched Nodes	Initial Launch Failures	Largest Partition Size After Stabilization	Living Nodes after failure injection	Convergence Time (s)
1	8	65	5	30	17	47
2	8	53	17	37	16	45
3	8	53	17	33	16	38
1	15	57	13	28	9	286
2	15	57	13	32	9	100
3	15	57	10	37	6	92
1	29	53	17	36	16	47
2	29	53	17	37	16	45
3	29	53	17	33	16	38

Table: Data from Experiment 1: Network Recovery Performance across Different Failure Percentages

This table represents the Network Recovery Performance from Node Failures across different failure percentages (f%). It shows the results of three trials for each of the failure percentages: 8%, 15%, and 29%. Each trial was tested on 70 nodes, and upon initial launch, a varying rate failed and didn't connect to SSH (Initial Launch Failures). After stabilization, we inject failures on the largest partition size (Largest Partition Size After Stabilization). Then we inject F% of failures and allow time for convergence. Largest Nodes After failure injection represents the number of living nodes in the largest partition after convergence has occurred.

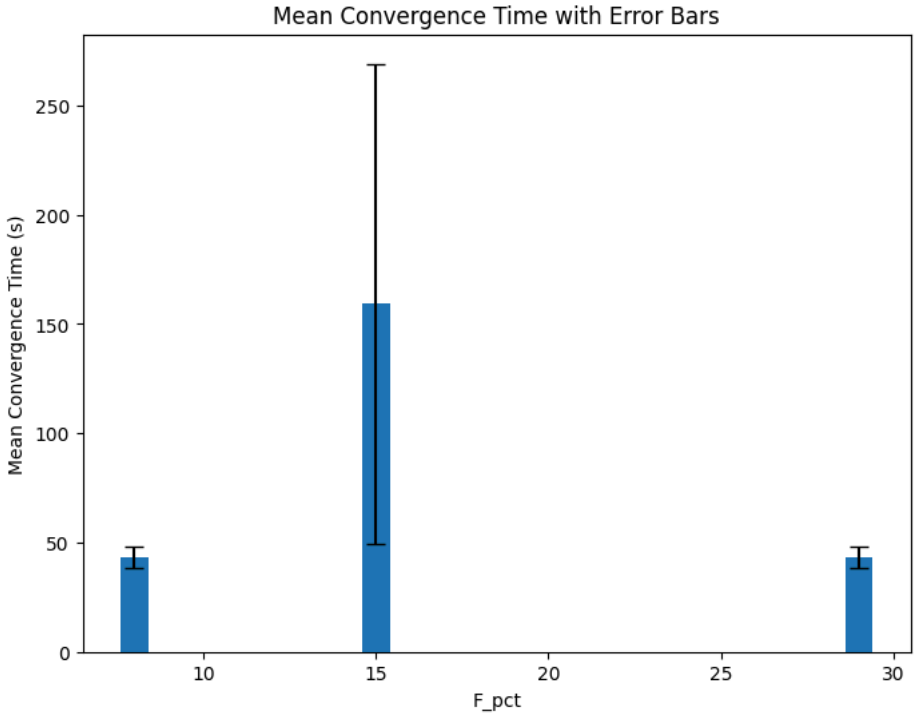

Figure: Mean convergence time after node failure across trials

The graph above shows the mean convergence time (in seconds) for different failure levels (8%, 15%, 29% node failures) across multiple trials. The mean convergence time is the average time taken for the network to recover from node failures.

Analysis

We initially expected the mean convergence time to increase steadily as the percentage of node failures (f) rose. However, the results revealed a more complex pattern, particularly at the 15% failure level, where a sharp spike in convergence time occurred. This unexpected spike suggests that the network faced significant stress during this trial, which could be attributed to the failure of critical nodes that disrupted the recovery process.

One possible explanation for this spike is widely varying network conditions across experimental trials. As we were performing this and other experiments we observed periods of significant

failures and slow responses among the test nodes. If the trial for f = 15 occurred during one of these periods, it makes sense that the ring would take longer to heal than in trials for f = 8 and f = 29.

Overall, the data indicates that the network's performance under node failures was less resilient than initially anticipated. Although the decentralized design should theoretically handle failures with minimal disruption, the results show that a failure of only 15% of the nodes in the ring was enough to cause recovery times as long as almost five minutes. However, the fact that our ring was able to recover to some degree demonstrates that our system can self-heal to some extent.

Experiment 2: Message delivery reliability under high traffic and concurrent node failures

Experiment Setup

- **Network Formation**:
  - A randomly selected group of 40 working nodes was created from the EC2 machine pool.
  - A subset of f (5, 10, 15, 20) nodes were chosen to be killed during the experiment.
  - One node initialized the ring, with 6 early joiners establishing multiple stable contact points in the network. (*Note: different number of nodes and join strategy than experiment 1 due to higher EC2 machine failure level*)
  - The remaining nodes joined by contacting randomly selected early joiners, as opposed to all nodes contacting the initial seed node. This better mimics decentralized join behavior and avoids a bottleneck at one node.
- **Stabilization**:
  - After the join process, the network was allowed 5 minutes of idle time to stabilize.
- **Message Workload**:
  - Once stabilization was complete, a message workload was initiated. Each non-victim node sent 10 broadcast messages concurrently, with a 0.75-second delay between messages to avoid burst transmission.
  - Flushing was disabled to keep message history in memory.
- **Failure Injection**:
  - After 60 seconds of message sending, node failures were injected progressively, with one node being killed every 2 seconds
  - After the failures, an additional 60 seconds was allotted for ring repair and message propagation to continue on the degraded network.
- **Reliability Measurement**:
  - Message delivery reliability was evaluated by polling every reachable node for its in-memory message list and comparing it to the expected set of messages for the trial.

- A delivery fraction (received/injected) was computed for each node to measure how many of the injected messages were successfully delivered despite node failures.

Results

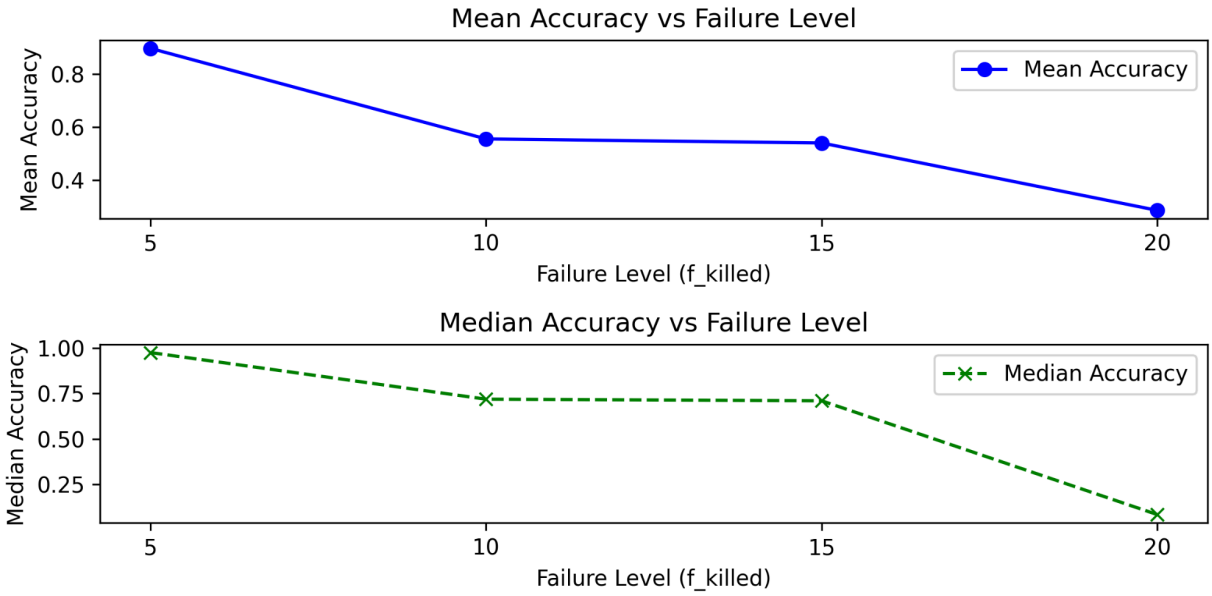

Figure: Impact of node failures on message delivery accuracy across trials for different failure levels.

Analysis

Our data demonstrates that message delivery accuracy, measured by both mean and median accuracy, decreases as the number of node failures increases. For each failure level, the accuracy metrics represent the averaged performance of multiple trials. As nodes are progressively killed, the system's ability to deliver messages reliably declines. This indicates that the network becomes less robust as more nodes fail. The mean accuracy decreases more significantly than the median as node failures increase, suggesting that a number of nodes did not receive many messages.

Overall

Based on both our experiments, to enhance the system’s delivery reliability and reduce convergence time under higher failure rates, several strategies can be implemented:

1. *Increasing redundancy*: By adding more fingers or increasing forwarding probabilities, the system can find alternative paths when failures occur, thus speeding up the recovery and delivery process.

2. *Increased retry mechanisms:* Allowing nodes to attempt message delivery and recovery more times before considering a failure could help in maintaining reliability.

However, these enhancements would come at the expense of increased network overhead, as maintaining extra routing information and performing more retries adds complexity and requires more resources. At a certain point, the increased overhead could create so much traffic that reliability would actually decrease.

Another key insight is that our experiments were severely impacted by external failures, like machine overloads or SSH connection issues. These problems contributed to longer recovery times & lower delivery reliability and should be considered when scaling up the system.

Note: We attempted to run a third experiment measuring the per-node load as the number of nodes in the network increased, but we were not able to perform a successful run due to network failures. We attempted this on December 18th and did not have a chance to run the experiment under better network conditions.

CONCLUSION

In this paper, we presented honeycomb, a lightweight, decentralized peer-to-peer communication platform designed to address the critical gap in communication capabilities during disaster situations. Utilizing a Chord-based ring architecture, honeycomb ensures relatively fault-tolerant and scalable message delivery in the face of node failures and high churn. Through a combination of probabilistic message forwarding and periodic network maintenance, honeycomb enables efficient message propagation in low-bandwidth, high-failure environments without relying on a central server.

Our experiments evaluated honeycomb's resilience under various failure scenarios, revealing that while the network performs well under moderate failure levels, the convergence time and message delivery accuracy degrade as failure rates increase. External factors, such as machine overloads and SSH issues, also contributed to longer recovery times, highlighting the need for additional measures to improve robustness.

Based on the results, we propose potential optimizations to improve performance under higher failure rates, such as increasing forwarding probabilities, modularizing our locking mechanisms, and tuning retry parameters. However, these enhancements come with trade-offs, including increased network overhead, which must be carefully balanced based on the specific requirements of disaster response networks.

In conclusion, while honeycomb provides a solution for peer-to-peer communication in disaster scenarios, future work should focus on optimizing network performance and fault tolerance to handle large-scale failures efficiently. These improvements will ensure that honeycomb can effectively provide reliable, real-time communication when it is needed most.

REFERENCES

[1] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup protocol for Internet applications," *MIT Laboratory for Computer Science*, Tech. Rep. TR-819, 2001. Available: https://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf

[2] Free Software Foundation, "Gnutella: A decentralized, distributed file-sharing protocol," GNU, 2000. [Online]. Available: https://www.gnu.org/philosophy/gnutella.en.html.

[3] M. Gardner, "Bring science home: The probability birthday paradox," *Scientific American*, Dec. 1990. [Online]. Available: https://www.scientificamerican.com/article/bring-science-home-probability-birthday-paradox/.