YZ.social: The Braided Blockchain

A Decentralized Resource Economy

davidasmith@gmail.com

February 9, 2026 Draft 0.20

The YZ.social platform is designed to enable a new class of social applications—from high-performance gaming, messaging, to global collaboration tools—by hosting them on a truly democratic, user-owned resource economy. Our vision is to eliminate centralized extraction and return all value directly to the community that creates it. We believe anyone, anywhere, should be able to participate in and benefit from the digital infrastructure they rely on. To achieve this, the system is engineered to allow any device—from high-end servers to mobile phones—to provide essential services like computing, communication, and storage and be rewarded in YZ coins. This compensation is not an extraction from the user base; it is the currency that fuels the very usage of the platform, creating a self-sustaining and democratic economic cycle.

To make this mass participation and equitable distribution possible, we are building the Braided Blockchain, the financial and computational engine of the YZ.social network. This architecture achieves mass-scale decentralization through a sharded architecture—a mesh of numerous smaller, parallel chains that break down massive computational loads, allowing them to be processed with high efficiency.

Ultimately, the system's most powerful feature is Universal Supplementary Income. Every node is regularly rewarded with YZ coins for merely demonstrating Proof of Availability—ensuring their device is online and ready to provide services. This simple yet powerful financial model is

the user’s dividend of ownership, ensuring that direct value and economic control remain in the hands of the people who power the network.

YZ.social Architecture

The YZ.social blockchain is built on proven technologies to ensure robust, low-risk operation, resting on two core innovations: a specialized network layer and a high-performance, distributed compute platform.

Distributed Hash Tables and the Concordance Engine

At its foundation, the YZ.social network relies on a **Distributed Hash Table (DHT)**, which is a decentralized directory system that functions much like the internet's Domain Name System (DNS), but for node devices and data. We use it to enable any node to find any other node or piece of data across the vast, global network without relying on a central server. In simple terms, it's a massive, shared "phone book" that no single entity owns, allowing nodes to securely communicate and store the small pointers that help the network operate.

The **Geographic Distributed Hash Table (G-DHT)** is our key optimization of the standard DHT. While a basic DHT uses a purely random addressing scheme that results in slow message times across global distances, the G-DHT introduces a smart optimization: it embeds a location identifier into a node's address. By doing this, your most frequent, geographically local connections are also logically close to you within the network's structure. This simple but powerful change dramatically improves performance for local interactions, making the average message time for local connections significantly faster.

The **Concordance Engine** is the high-performance, distributed compute platform at the heart of running the shared, multi-user applications, including the Braided Blockchain ledger. It is a censorship-resistant system that ensures reliable operation by running **bit-identical replicated processes** across multiple nodes. This capability guarantees that all participating validators arrive at auditable, identical results, a prerequisite for a trustworthy decentralized ledger.

The Braided Blockchain

The core application in the YZ network is the Braided Blockchain, a decentralized ledger designed to achieve mass scalability and speed through a sharded architecture. This system breaks the network into numerous smaller, parallel chains, with each one—called a shard—operating autonomously to achieve fast, local finality for its transactions.

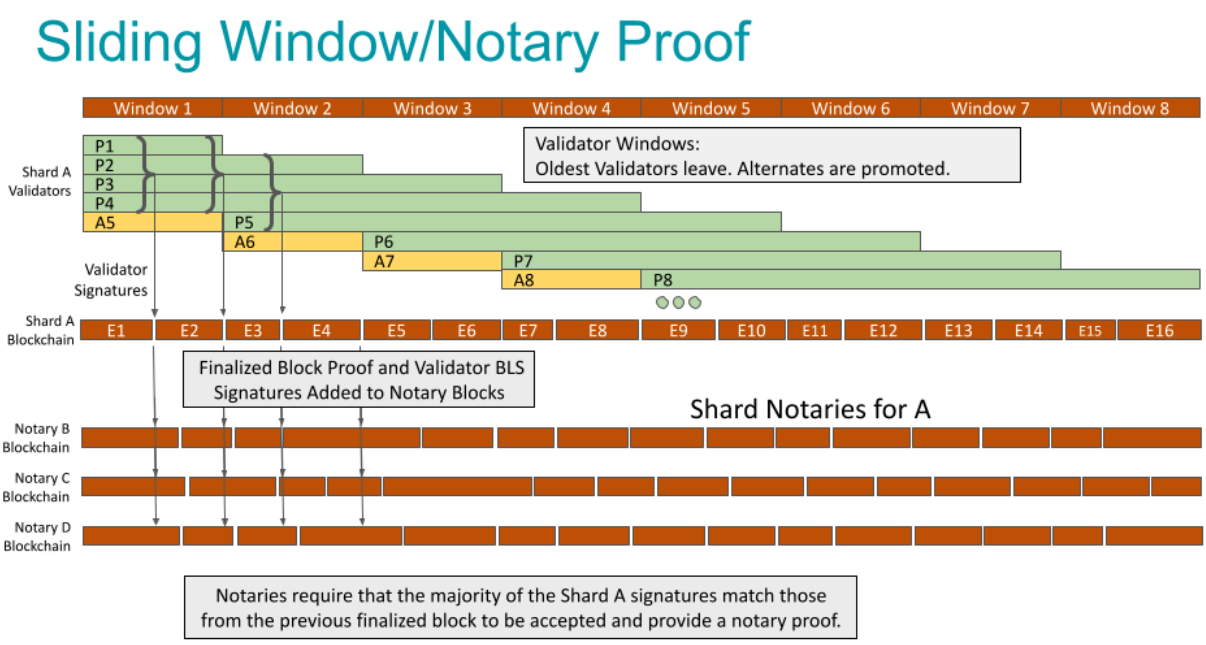
The key innovation that replaces the need for a single, slow, global blockchain is the Braid of Trust. This is a mesh of persistent peer-shard connections where a small, random group of peer shards (referred to as Notaries) are assigned to audit a shard's history. Instead of relying on a slow, global consensus process, the Notaries issue a signed Notary Validation Receipt as a public seal of legitimacy for each finalized block. This receipt is mandatory for all cross-chain activity, providing an auditable and fast cryptographic audit that secures communication and value transfer across the interconnected network.

This sharded, audit-based approach is essential for managing system costs and ensuring long-term financial stability by efficiently compensating all participating nodes for providing computing, storage, and communication services through Proof of Availability and Proof of Service rewards.

Local Shard Consensus

Local Consensus means that each shard operates autonomously with its own small, fast, rotating group of participants - now acting as validators of the blockchain (**Sliding Window Validation**), allowing thousands of local transactions to be finalized simultaneously across the network. As an example, a Concordance process is made up of a number of nodes: one relay node (plus a number of alternates that also act as transaction witnesses until they are promoted to relay); a larger number of validator nodes (for example, 32 nodes that perform the replicated transaction computation); and four alternates that also perform the computations. The alternates are not directly involved in the hash-proof of the block, but are required to compute the same hash proofs in order to join the session. At regular intervals, the oldest session validators retire and the alternates become full validator nodes. If a validator drops out before the end of this interval, one of the alternates can be automatically promoted (if required). Another four alternates are selected from the DHT community at random to continue the process. This is the Sliding Window of validators/participants to ensure consensus under churn and potential bad actors. Acting as the Relay or a validator is a Proof of Service activity, which provides additional rewards beyond that provided by Proof of Availability, incentivizing continuous participation in the Concordance Engine.

The image below provides a simple example where we have four validators and one alternate in a Concordance blockchain. A window is simply the time at which the oldest validator retires and the alternate becomes a full validator. A new alternate is then selected at random from the full DHT. The alternate is synched to the current shard state so it is able to track the transaction stream just as a full validator does.



Fast Local Finality

Byzantine Fault Tolerance (BFT) Consensus, the model employed by many modern single-chain Proof-of-Stake systems, requires all validators—or a large, synchronized subset—to participate in multi-round voting to achieve "Global Finality." This significant overhead severely limits transaction throughput.

In contrast, the Braided Blockchain utilizes an Auditable Fast-Path approach, which achieves "Local Finality" instantly on each individual shard. For cross-shard communication, the system shifts the security check from a slow consensus process to a fast, efficient, two-part cryptographic audit: a double-verification security primitive combining Notarized Validity Receipts and a Light Client Proof Package. This change enables the receiver to audit the transaction quickly and efficiently, making the system massively more scalable than BFT.

Braid of Trust

Cross shard security is provided by the shards themselves through **The Braid of Trust (Notaries)**. Each shard is assigned a persistent, small, random set of peer shards that act as Notaries, auditing its operational history and issuing a signed **Notary Validation Receipt** as a public seal of legitimacy for each finalized block. This **Notary Receipt** and associated entry into the Notary shard's blockchain is mandatory to enable all cross-chain transactions, as it prevents a malicious actor from cloning a shard to attempt a fraudulent transfer.

At the end of every shard's blockchain epoch, the blockchain header and the validator attestation signatures (BLS signatures) are sent to the notary blockchains. The notary blockchains compare their new signature set to the previous set. The notaries require that there is a defined threshold of overlap (e.g., 66% of the previous epoch's validator BLS signatures must be present in the new set) between the previous attestation and the new one sent to the notary to provide a proof of validity for the requesting shard and its finalized block. This notarized proof is shared with the sending block and placed onto the notary block's chain as well.

As an example, when a transaction moves from Shard A to Shard B, Shard B relies on two proofs from Shard A instead of slow, global consensus: first, Shard Validity (**Notary Receipts**), which proves that Shard A - specifically, its finalized block - is legitimate; and second, Transaction Proof (**Light Client Proof**), a small, cryptographic proof confirming that the specific transaction was correctly recorded and finalized on Shard A.

Cross-Shard Transactions: Double Verification

The Braided Blockchain achieves scalability by replacing the global security model with a system of localized, mutually-attesting security domains. Transactions are secured by two independent, verifiable proofs, satisfying the recipient shard that both the sending shard and the availability of its funds are legitimate.

Alice, residing on Shard A, decides to send some tokens to Bob, whose account is on Shard B. The entire process begins in Alice's home, Shard A, which first processes and quickly confirms the debit, subtracting the tokens from her account using its own fast, local consensus. To ensure the transfer is secure and legitimate, Shard A then gathers two critical pieces of certification: first, it obtains a signed Validity Receipts from its trusted Notary Shards (Proof 1), which vouches for the validity of the finalized block generated by Shard A itself - that is, it provides a proof that Shard A constructed the block, but makes no assertion about the contents. Second, it creates a Light Client Proof Package (Proof 2), which cryptographically confirms that Alice's debit transaction was officially included in Shard A's block. This combined proof package is then dispatched across the network to Bob's home, Shard B. Upon receiving the package, Shard B's contract acts as the final gatekeeper, performing a double-verification check: it verifies the Notary Receipts to trust the source, Shard A, and verifies the Light Client Proof to confirm the specific debit event. Only after both proofs pass this fast, auditable cryptographic check does Shard B confidently credit the tokens to Bob's account. This efficient method eliminates the need for a slow, global consensus across the entire network.

1. **Execution and Local Finality (Shard A):**
    - Shard **A** finalizes the block containing Alice's debit.
    - The **validators**/concordance participants produce a Finality Certificate (C) for the block header (H), which is a single, compact, aggregated BLS signature.

2. **Proof Generation (Shard A):**
   Shard A prepares two distinct sets of proofs to deliver to Shard B:
   - Proof 1: Shard Validity Proof (Notary Receipts):
     Shard A obtains the signed Notarized Validity Receipts from its Notary Shards (N_1, N_2, N_3). These receipts certify that Shard A's transition to the current epoch (which contains the debit) was legitimate and that its validator set passed the "Overlap Proof" check. This confirms the validity and lineage of the sending shard itself.
   - Proof 2: Transaction Proof (Light Client Proof Package P):
     (See **Light Client Transaction Proof** in the Appendix)
     This package cryptographically proves that the debit transaction occurred within the certified block.
     P = { H, C, M }
     - H (Header) &
     - C (Certificate): Certifies the finality of the block.
     - M (Merkle-branch): Links Alice's specific transaction to the certified block header by proving its inclusion in the transaction root.
3. **Relay & On-Chain Double Verification (Shard B):**
   - The combined proof bundle (Notary Receipts + Proof Package P) is sent to Shard B.
   - Shard B's bridge contract performs two validation steps:
     - Verify Proof 1 (Shard Validity):
       Shard B verifies the digital signatures on the Notary Receipts. This establishes trust in the source, Shard A, by confirming its validity via known, trusted peer shards/notaries.
     - Verify Proof 2 (Transaction Inclusion):
       Shard B verifies the Light Client Proof Package P (checks the aggregated BLS signature C and the Merkle Branch M). This confirms the specific debit event happened as claimed.
4. **Credit:**
   If both proofs are successful, Shard B accepts the funds and credits Bob's account. This double-verification process ensures both the legitimacy of the sending shard and the correctness of the specific transaction, meeting all security constraints while remaining highly efficient.

Proof of Availability and Proof of Service

User nodes are rewarded for participating in the network, provided they demonstrate availability. Users must prove they've participated to at least a minimal level; they can't expect rewards by joining and rarely returning. For instance, rewards can be awarded hourly for availability. If a

node also participates in value creation, such as completing a full cycle in a concordance engine blockchain, they receive additional rewards as long as they don't exit before finalization. Providing services for a full day earns an extra reward, and consistent, long-term availability and service are rewarded even more.

Participants are rewarded with YZ coins for providing a **Proof of Availability** or a **Proof of Service**.

Proof of Availability is checked hourly on a randomized schedule. Every participating node subscribes to its banking shard's Availability pub/sub. The banking shard constructs a test with a randomized key value and publishes it. Subscribers must automatically "sign" and republish the key back to the banking shard. The only requirement for the user is that the application is running and available. Once the banking shard receives the signed key, the participant's account is credited with the availability reward.

Proof of Service requires the participant to actively provide services, such as acting as a Concordance Engine Relay or participant, or offering compute or storage services. Proofs of Service are essentially identical to the proofs needed to complete transactions on the Braided Blockchain. These proofs are co-signatures from the other participants and relays who hosted the Concordance Engine, confirming the participant provided the required services. Participants are chosen at random for these services, and the rewards are significantly greater than those for Proof of Availability, incentivizing continuous availability to the YZ network.

Proof of Storage also requires a proof that the nodes indeed continue to store the data without the owner having to retrieve it. One way to determine that the data is actually available is to request that the nodes storing the data compute a hash of a subset of the file and forward it to the user whenever the owner’s subscription is updated. For example, compute the hash of the file range of 2001-3210. This hash would be computed by all of the nodes that are supposedly storing that file, and can only be completed if they actually have it. Upon receipt of this proof, the owner can then approve the continuance of the storage subscription.

Participants can spend their earned reward income to use the YZ platform or for any other purpose.

To minimize churn, participants may be randomly selected to receive a large reward, but only if they are available to receive it.

Users are also rewarded for various services provided to the system, including growing and validating the user population, and providing communication, compute, and storage. Services for blockchain ledgers are particularly valuable.

Auditable Fast-Path vs. BFT

One noteworthy aspect of the Concordance Engine is that it provides an alternative to classical Byzantine Fault Tolerant (BFT) consensus, the model typically used for blockchain systems.

This section contrasts a Concordance Engine-style synchronization architecture, augmented with end-to-end encryption, deterministic state hashing, and independent observer logs, against classical Byzantine Fault Tolerant (BFT) consensus. The focus is on practical performance, rapid identification of malicious behavior, and fast replay-based recovery.

In a BFT system, safety and liveness are achieved by requiring quorums of mutually distrusting validators to agree on the order of events. The protocol prevents a single leader from finalizing conflicting histories, equivocating, or selectively reordering messages without agreement. These properties come at a well-known cost: multiple rounds of coordination and cryptographic verification increase latency and jitter, and network overhead grows with the size of the validator set. Modern BFT variants improve constant factors and employ optimistic fast paths under synchrony, but their essential trade-off remains: they prioritize prevention, providing strong guarantees before action is taken, rather than detecting and correcting faults after the fact.

By contrast, the Concordance Engine architecture is designed for ultra-low-latency collaboration. A single blind relay orders and relays events to deterministic clients that replay them in lockstep. With the additional constraints introduced here, the model gains strong auditability without sacrificing performance. First, events are end-to-end encrypted: the relay cannot read payloads and therefore cannot apply content-aware filtering. Second, participants compute and share periodic hashes of their full instantaneous state, derived from the same deterministic replay; honest clients will converge on identical hashes at each checkpoint. Third, independent observer nodes (backup relays for example) receive the same encrypted events and construct a signed, hash-chained ordered log. These observers do not forward traffic to clients, but they can publish log tips, such as the hash and, given the initial state, enable anyone to reconstruct the identical state trajectory and verify the participants' checkpoint hashes.

Together, these constraints shift the Concordance Engine model from "trust the relay" to "trust but verify, quickly." The relay remains a single fast-path component; there are no quorum rounds on the critical path, so interactive latency remains extremely low and throughput scales well with fan-out. The observers and state-hash checkpoints furnish immediate detectability and provability of faults: any attempt by the relay to censor, reorder, or equivocate becomes visible as a divergence in the hash chain seen by observers, a mismatch between participants' checkpoint hashes, or a gap/inconsistency in the published ordered log. Misbehavior is not merely suspected; it is attributable via signatures on sequence indices and ciphertext digests, and it is reconstructible by replay.

This architecture is expressly optimized for fast identification and fast recovery. Upon detection of inconsistency—whether through observer tip disagreement, a broken sequence chain, or a checkpoint hash mismatch—clients can pause, roll back to the last agreed index, and deterministically replay from an observer's canonical log. Because execution is deterministic and checkpoints are frequent, replay is swift and reliable, restoring convergence with minimal user disruption. Operational policy can further reduce downtime by designating an ordered preference among observers for automatic failover of the relay role, using the last common signed tip as a handover point. While this approach does not provide the formal liveness guarantees of BFT under adversarial conditions, in practice it yields rapid continuity with bounded interruption.

The comparative fault model clarifies the distinction. BFT prevents a minority of malicious operators from unilaterally imposing order, stalling progress, or finalizing conflicting histories; it ensures safety by construction and liveness under stated timing assumptions. The Concordance Engine design with audit constraints does not prevent short-term misordering or censorship by a malicious relay; rather, it renders such behavior quickly evident and provable, enabling swift remediation through rollback, replay, and failover. In effect, it trades preventive consensus for accountability and agility: the system moves fast in the common case and corrects fast in the exceptional one.

Performance characteristics follow naturally from these choices. BFT incurs multi-round communication and cryptographic aggregation, resulting in higher median latency and heavier tails during leader changes or network perturbations. The audited Concordance Engine model retains single-hop ordering and deterministic client execution, so end-to-end latency remains close to network round trip time plus minimal sequencing delay. Overheads are modest: per-message authentication and encryption, periodic state hashing (which can be made incremental or Merkleized for large states), and the maintenance of a hash-chained log by observers. Additional bandwidth arises from mirroring events to observers and from disseminating observer tips and checkpoint hashes, but none of this sits on the interactive critical path.

From an assurance standpoint, the Concordance Engine model gains properties that are often mistakenly assumed to require consensus. Public verifiability is achieved because anyone with the initial state and the observer log can reproduce the final state and compare it against the participants' checkpoint hashes. Privacy is preserved because payloads remain opaque to infrastructure, including observers. Accountability is enforced through signatures and hash chains, yielding clear evidence in the event of misconduct. What remains outside its scope is real-time prevention: a single sequencer can still briefly delay or bias ordering before being detected and replaced, a risk that BFT's quorum mechanics explicitly mitigate.

BFT offers preventive consistency and progress among distrusting operators with higher latency and coordination cost. The Concordance Engine approach, augmented with end-to-end

encryption, deterministic state hashing, and independent observer logs, offers ultra-low latency and high throughput while delivering rapid, provable detection of faults and fast, deterministic replay-based recovery. Where real-time prevention under Byzantine conditions is mandatory, BFT remains the appropriate choice. Where user experience, performance, and operational accountability are paramount—and rapid identification and correction of faults suffice—the audited Concordance Engine design provides a practical alternative.

Light Client Transaction Proof

Block Transaction Proof: Construction, Merkle Tree, and Verification

The transaction proof P = { H, C, M } is constructed to allow any entity—including other shards or applications—to quickly and securely verify that a specific transaction has been finalized on a Braided Blockchain shard without needing to download the entire shard's history.

1. Construction of the Block Transaction Proof P = { H, C, M }

The proof is generated through the coordinated effort of the executing shard's **Validator Nodes** and the cross-shard auditors (**Notaries**).

1. **Block Execution and Merkle Root Generation:**
    - The shard's **Validator Nodes** (participants in the Concordance) compute the block, which includes a batch of transactions.
    - As part of this process, all transactions in the block are organized into a **Merkle Tree**. The single, final hash at the top of this tree, the **Merkle Root**, is computed.
2. **Header (H) Finalization:**
    - The **Block Header (H)** is assembled, containing all necessary block metadata, including the crucial **Merkle Root**.
3. **Certificate (C) Issuance (Notarization):**
    - The finalized Block Header (H) is then audited by the persistent peer shards known as the **Notaries** (The Braid of Trust).
    - Once the Notaries verify the shard's validity, they cryptographically sign the Block Header (H), which produces the **Notary Validation Receipt (C)**, certifying the block's finality and inclusion into the braid.
4. **Merkle Proof (M) Generation:**
    - A user requesting a proof for a specific transaction generates the **Merkle Proof (M)**. This is not the whole block, but the minimal set of sibling hashes from the Merkle Tree required to reconstruct the Merkle Root for that single transaction.

The complete proof P is then comprised of: the **Header (H)**, the **Notary Validation Receipt (C)**, and the requested **Merkle Proof (M)**.

2. The Nature of the Light Merkle Tree (Merkle Proof)

The term "light Merkle tree" in this context refers to the **Merkle Proof (M)**, which is the **Merkle Branch**.

- **Merkle Tree:** This is a data structure in which every leaf node is a hash of a transaction, and every non-leaf node is a cryptographic hash of its two child nodes. This process continues until a single hash, the **Merkle Root**, remains at the top.
- **Merkle Branch (M):** The Branch is the small, ordered list of hashes needed to link a single transaction hash back to the Merkle Root. For a block with thousands of transactions, this list is logarithmically small (e.g., around 10-15 hashes) compared to the size of the whole block. It is "light" because it provides proof of inclusion with very minimal data.

3. Proof Utilization by the Recipient

A recipient (e.g., a node, a cross-shard bridge, or a wallet) uses the proof P = { H, C, M } to verify the transaction in a three-step trustless process:

1. **Step 1: Verify Finality (Using C):**
    - The recipient first checks the **Certificate (C)** (Notary Validation Receipt). By verifying the cryptographic signatures of the Notaries, the recipient proves that the block header (H) is legitimate, has been audited, and is finalized. This confirms the block is a valid part of the Braided Blockchain.
2. **Step 2: Verify Inclusion (Using M and H):**
    - The recipient takes the transaction they are interested in, hashes it, and then combines this hash with the hashes provided in the **Merkle Proof (M)**, following the structure of the Merkle Tree.
    - The result of this process must exactly match the **Merkle Root** contained in the verified **Header (H)**. If they match, the recipient has mathematically proven that the transaction was included in the block referenced by the header.
3. **Step 3: Transaction Occurred:**
    - Since the recipient has a cryptographically-signed certificate (C) attesting to the finality of the block (H), and an inclusion proof (M) linking the transaction to that finalized block, the recipient can be certain that the transaction has successfully occurred and been recorded by the network.

Appendix: Network Layer Protocol Details

Under the hood, YZ.social relies on a very high performance, location aware, decentralized communication platform called a Geographic Distributed Hash Table (G-DHT), a shared directory that is spread across the community's devices. In plain terms this system behaves like the Internet's Domain Name System (DNS), but for node devices and data. No single entity owns the entire "phone book." Instead, each node holds a small cache of directory information (like DNS records), knows who to query next in the network hierarchy, and works with others to resolve an ID into the correct network address or location. This is how nodes find one another and store the small pointers and facts that help the network operate. Conversations themselves, and any time-sensitive activity, can move to faster, direct connections between the people or applications involved. For topics and groups, we use a publish/subscribe model also managed by the DHT nodes. Rather than addressing individual inboxes, you follow subjects you care about—such as the location cells of the map around your neighborhood in the Alert app, or a particular celebrity or live economic data—and when someone publishes to that subject, your device receives the message. This pub/sub system tolerates duplicates and brief interruptions, and it retains recent items long enough that a latecomer can still catch up on important information.

Kademlia Distributed Hash Table

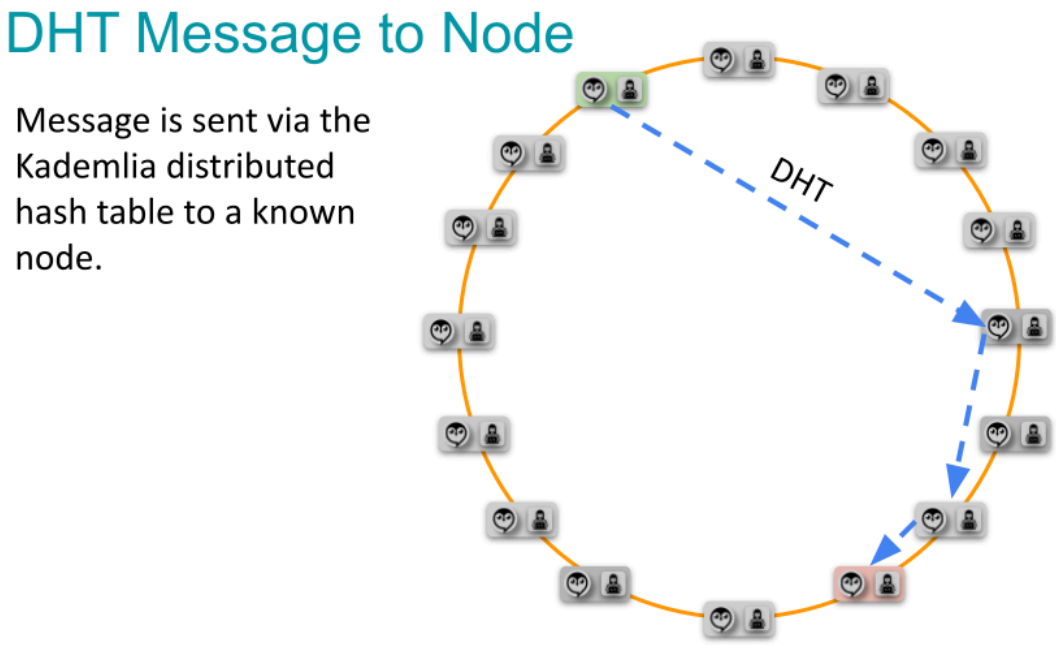
The foundation of the Geographic DHT is a modified Kademlia Distributed Hash Table (K-DHT). Instead of a hash, each node in the K-DHT has a public/private key pair where the public key also acts as the node ID. The public key is crucial as it serves a dual purpose: it's used for encrypted communication, and also acts as the node's address or ID within the distributed network, enabling secure communication without additional negotiation or data transfer. The Geographic DHT utilizes an additional location information prefix to the node ID that dramatically improves performance at the expense of some anonymity.

Messages within the YZ Network

There are three kinds of messages that can be sent in the YZ network: messages sent indirectly via DHT to a node; messages sent directly via IP to a node (required by some topologies such as the Concordance Engine); and messages sent via a neighborhood publish/subscribe connection (for example, locations shared in the YZ Alert application). The Concordance Engine supports one other: a direct time-based publish/subscribe.

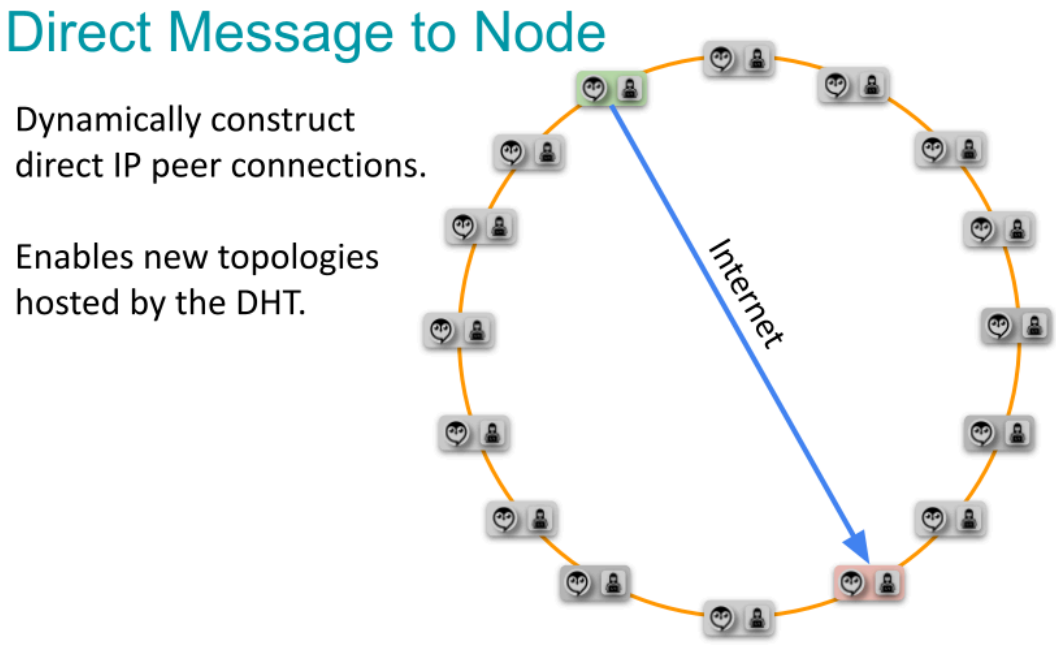
Indirect Messages (DHT Routing)

Knowing a target node's ID allows a node to send an encrypted message to that specific target using the DHT overlay network. The message is encrypted using the target node's public key/address, and is then routed via the DHT. The target is not obliged to acknowledge receipt - for example, the message might just be a comment in a text chat. If the message is a request to connect, it would include the direct link to connect live. Virtually any message can be distributed this way. This mechanism is potentially slow - each message may require a number of hops between nodes - but it is essential for much of the infrastructure that is built on top of it. (Dotted lines are DHT-based messages. Solid lines are messages sent directly between peers.)



Direct Messages (High-Speed Topologies)

Additional topologies can be added to the DHT so that we have high-speed, direct connections between nodes. As an example, this would be required for live peer-to-peer video streaming.



Publish/Subscribe (Pub/Sub)

The kinds of messages described above meet a range of potential application needs: indirect messages that maximize security, and direct messages that offer best performance. Pub/sub messages complete the set, by providing a way for one node to sign up to receive messages sent by any other nodes without having to directly know their address or connect to them.

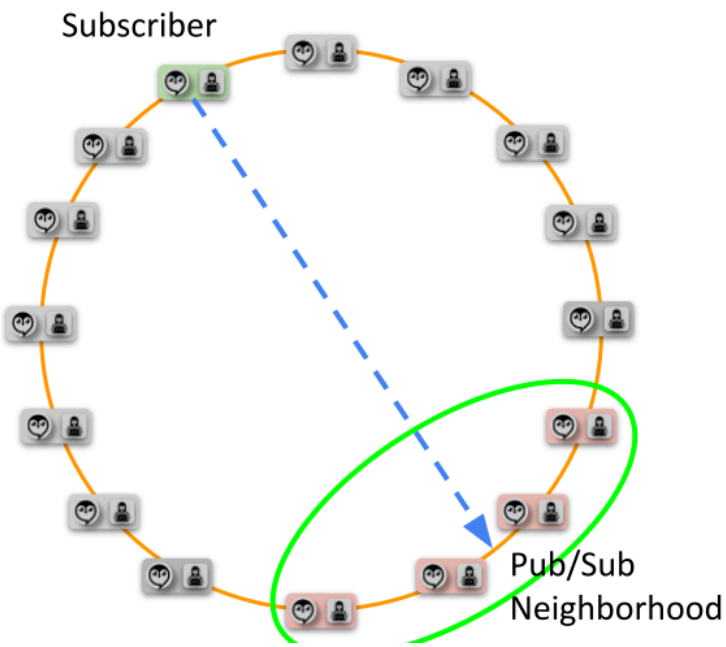
Pub/sub messages are specific kinds of neighborhood messages that are addressed to a domain key. A domain key is mapped to the DHT neighborhood ID. This key may be the hash of a known string or an S2 geographic location as used by the YZ Alert application. A node subscribes to a particular event associated with this domain key. All the nodes in the key neighborhood add this node's subscription to their pub/sub table.

As an example, we can subscribe to the YZ.social system update messages to track what is happening on the platform.

subscribe(myID, "YZ.social", "updates") - myID is my address. "YZ.social" is hashed to generate the domain address on the DHT, "updates" is the event type that the subscriber is interested in tracking.

Subscribe

A node sends a message to generate a subscription in a neighborhood. All members of this neighborhood have a table that includes the Neighborhood ID, the event name, and the direct (or indirect) connection to the subscriber.

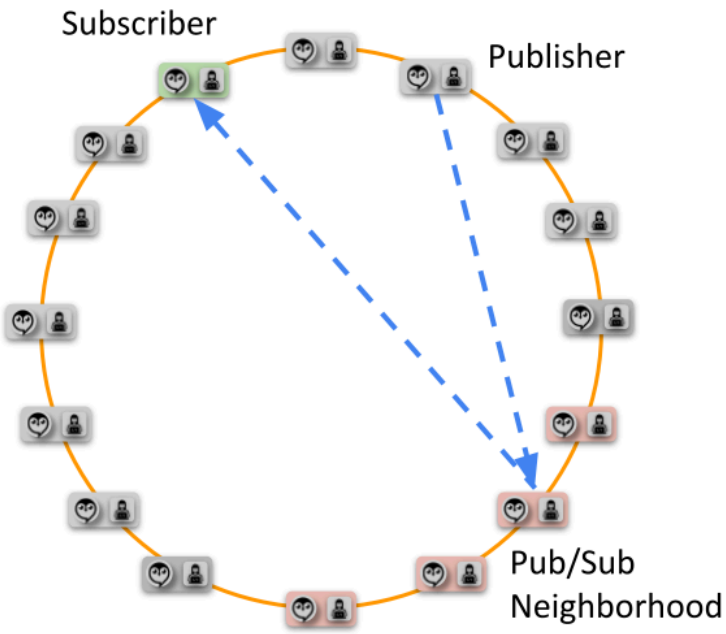


A publish event is also sent to the same domain. The domain and event must both match the subscriptions for the message to be processed and forwarded. Thus:
publish(YZID, "YZ.social", "updates", "A new version of the app is available", 60000) - YZID is the publisher domain as well as the public decryption key, the domain and event are the same, followed by the message to be relayed and the wait time to keep it available - 10 minutes in this case.

Publish

A node sends a publish event message to the domain neighborhood.

The first node encountered with a record for this event subscription forwards that event to all subscribers.



Published events can be "sticky" in that they may be stored by the neighborhood nodes for some time period. This is done so that late-subscribing nodes can immediately receive recent published events. This might be useful for a community watch application where an event has occurred just before a new user sends their subscription. This enables them to see the live current state of their area without having to wait for updated information. Another application is a store and forward mechanism for text messages or emails; in this case the wait time might be many days before the recipient is available.

Any given node may support multiple overlapping pub/sub domains, each with multiple events and each of these with multiple subscribers.

An example of the publish/subscribe API is:

publish(publisherID, domain, event, data, wait)

publisherID: node ID of the publisher (optional)
domain: DHT neighborhood address
event: a string - or stringified message
message: perhaps JSON object (an array of arguments for example)
wait: the stickiness of the message (in milliseconds). For example, the YZ Alert app may keep the event alive for ten minutes.

subscribe(subscriberID, domain, event)

subscriberID: node ID of the subscriber
domain: DHT neighborhood address
event: a string - or stringified message

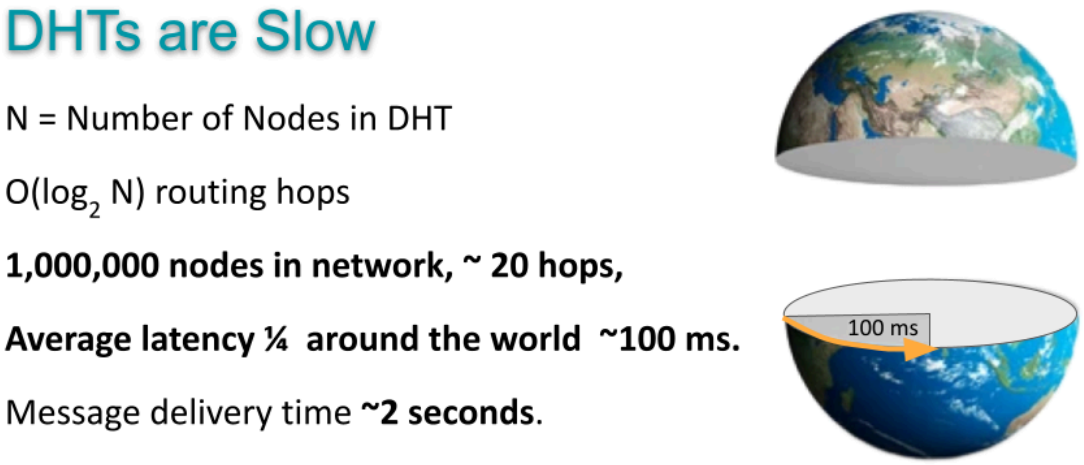
Subscriptions also have a time to live, but a user can renew their subscription with a **subscription.renew()**

Each message sent to a subscriber is of the form **(publisherID, domain, event, data, utcTime)**

utcTime: the UTC time at which a pub/sub node receives the published event

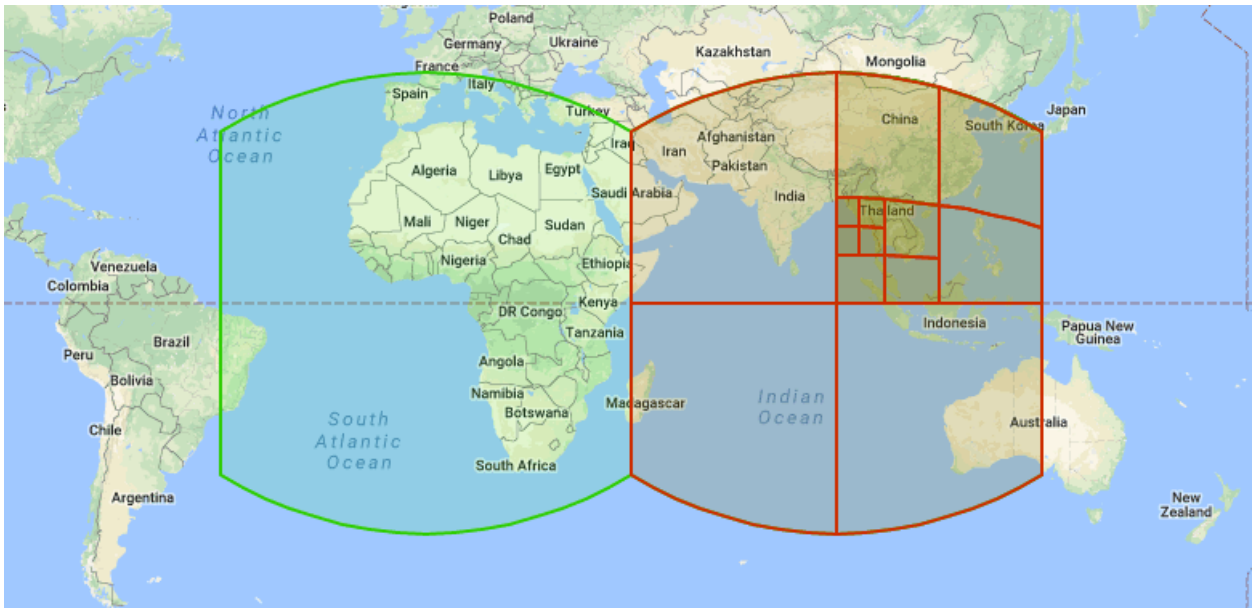
The Geographic Kademlia Network

The Kademlia algorithm is considered efficient because the base DHT requires on average $O(\log_2 N)$ routing hops to communicate with any node. However, since the IDs/public keys are randomly distributed, and if we presume that the nodes are spread homogeneously across the globe (they aren't quite, but it is a reasonable approximation), then the average distance between two nodes is one half the distance to the far side of the world.



As an approximation, we can assume that this distance has 100ms of roundtrip latency. Thus, if we have 1,000,000 nodes in our network, and the Kademlia algorithm provides a $\log_2(N) \sim 20$

hops, the average message time is 2000 ms - i.e., two seconds. For many applications, this would be too slow.



S2 Geometry

A feature of the S2 library is that unlike traditional geographic information systems, which represent data as flat two-dimensional projections (similar to an atlas), the S2 library represents all data on a three-dimensional sphere (similar to a globe). This makes it possible to build a worldwide geographic database with no seams or singularities, using a single coordinate system, and with low distortion everywhere compared to the true shape of the Earth. The resulting global address is used as a prefix to the public key to define the node ID.

We can dramatically improve this performance by recognizing that most connections a user will have will be geographically local. By augmenting the node ID with a location prefix, for example one defined by the previously mentioned S2 geometry library https://s2geometry.io, we can establish a metric that means that our local connections, which are most used, will also be located nearby in the Kademlia node space.

Geo DHTs are Faster

N = Number of Nodes in G-DHT

$O(\log_2 N)$ routing hops

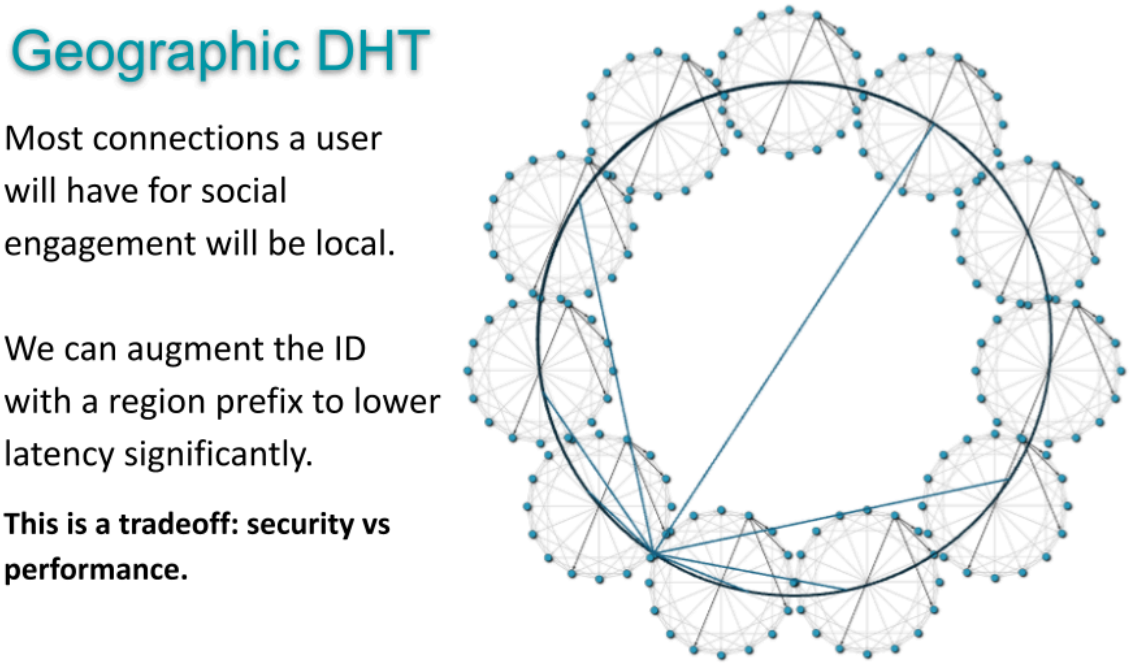
10,000 nodes in region, ~ 13 hops,

Average latency within area ~7 ms.

Message delivery time **~90 ms**.

This improves global performance as well!

In this subset, our population of nodes is much smaller, hence far fewer hops, and because the nodes are geographically close, the connection time is significantly lower with a target of 10-30 ms per hop. Thus, if we presume 10,000 nodes in the local population - thus an average number of 13 hops - and an average of 7 ms latency, we get about 90 ms latency between local connections. One way to think about this is when communicating with a far-away node, we first need to find a connection within that node's local geographic neighborhood; once there, the rest of the hops are extremely efficient.



One way to think of the Geographic Kademlia Network is a network of networks.

It is important to note that the G-DHT is a tradeoff between security and performance. A public key is essentially a large random number and its connections in the base Kademlia algorithm have no relationships to actual locations. Applying a geographic constraint to a node ID gives a potential attacker a way to surmise the approximate region of the node holder. This would be offset by the sheer number of nodes in a given location, and of course a node holder can decide to spoof their location for added security at the expense of performance.

User System Recovery

A regular snapshot can be made of the state of a user's YZ.social application and stored across the DHT network. The user need only have access to their keywords, their node ID/public key and private key to restore their application state. The data address is a hash of the keyword and the public key. The data itself is encrypted with the node's public key. This allows the user to, if necessary, instantly and completely delete their node and all associated information stored by it, but recover completely at a later time. The user must still remember their keys, probably stored on a flash drive or similar persistent device, to fully reconstruct the state of their application.

Application Network Topologies

Transport is independent of function. The transport layer is aware of IP addresses and Public Keys; the application layer is only aware of Public Keys as necessary. The two layers do not directly connect to each other, but communicate via a well defined API. They are both running as independent virtual machines.

The Kademlia algorithm provides a base level of connection and communication between any two nodes. Most messages will be sent via this indirect graph, but for many network applications, this architecture will not provide the performance required. Instead, an application can use the base K-DHT network to construct a new topology on top of it where the nodes have direct connections to the other nodes.

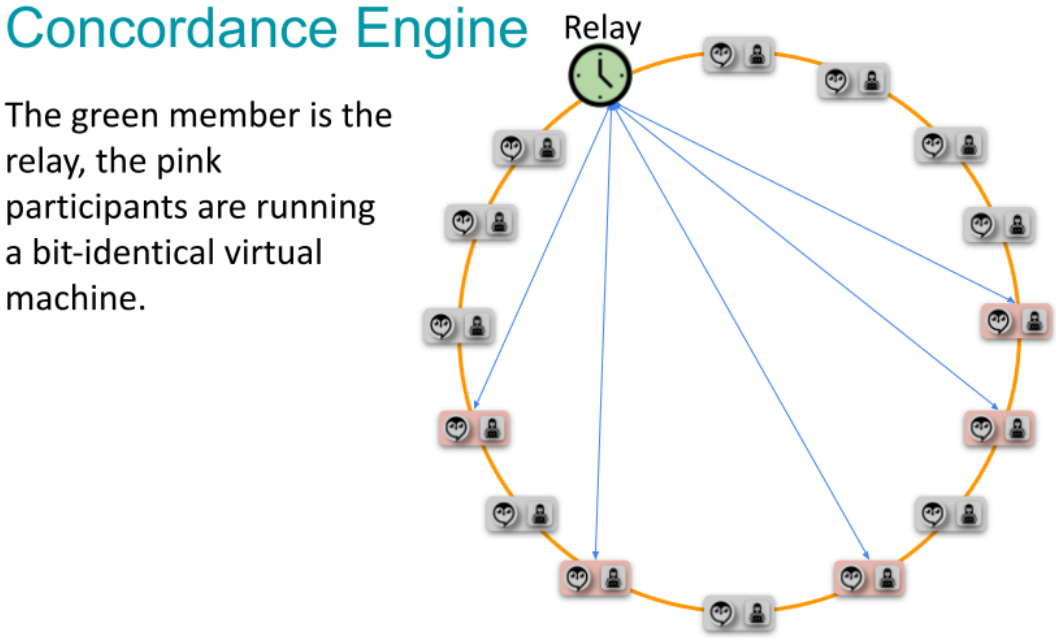
An example of this would be decentralized video chat, where the participants can be anywhere in the world. A message is sent providing them with the direct connection information required for high bandwidth/low latency communications, and each participant's video chat application can then request a direct connection to the other participants via an overlay network message. This would be accomplished using the DHT network publish/subscribe. A new user to the application simply subscribes to a domain or topic ID (perhaps a hash (Topic.ID)) that maps to a neighborhood which holds the connection information for the application.

The Concordance Engine, described below, is a very high performance, replicated computation platform built as a directly connected star network, assembled and maintained via the DHT.

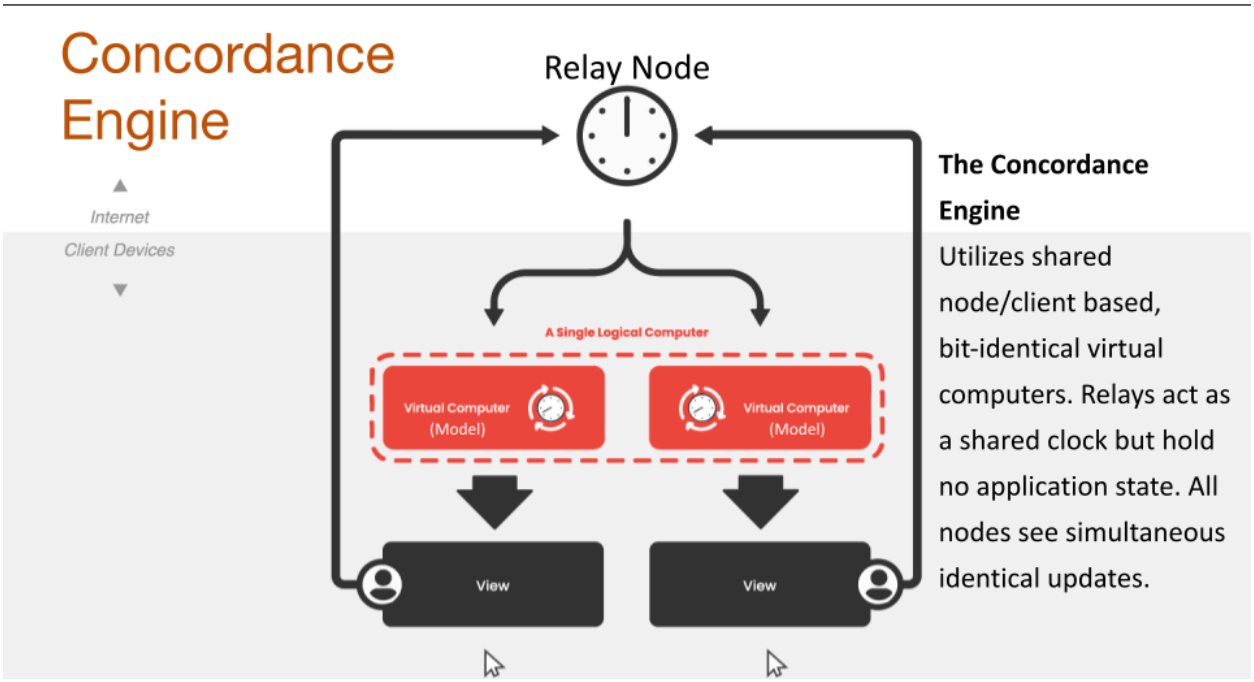
If the user requires it for security, any communication of information can be performed indirectly – so that there is no need for a direct edge connection to another node - but this is at the expense of latency.

Concordance Engine

Typical decentralized systems suffer from the churn of the participating nodes. There is no way to predict when a given node may lose its connection to the network: a user may shut their computer off, or they might be entering a dark signal area in a subway. Furthermore, any given computation running on a single node can be corrupted - either intentionally or simply in error. The Concordance Engine is a robust, extremely highly performant, Proof-of-History platform that provides the ability to run bit-identical, replicated computation processes on the YZ network.



This is achieved by providing bit-identical virtual machines on multiple DHT compute nodes. These virtual machines run a given program in lock step on all the participating compute nodes connected to a relay node. The relay node provides a shared clock as well as message forwarding services via the YZ network publish/subscribe. Every node in the YZ system is able to provide compute services and relay services. The collection of nodes supporting the computation of this process is called a Concordance.



Key features include:

Decentralized. The Concordance Engine moves the computation normally performed on centralized servers to the clients themselves, providing dynamic, bit-identical synchronization of user events and simulations. The Concordance Engine utilizes **Relays** which are a kind of lightweight server that maintains no application state. Participant/Node events generated within an application are immediately forwarded to the Relay which adds a time-stamp and redistributes it. For all nodes this event thus occurs at exactly the same "virtual" time, and the results of the event are computed identically.

Replicated computation. All nodes in a Concordance session compute exactly the same state, including complex simulations. There is no requirement to update, synchronize or roll back to maintain consistency.

Join process at any time. A snapshot of the running application is copied to the new node from an existing node. Missed events are executed and the new node is quickly and perfectly synchronized. This includes even complex simulations.

Guaranteed Synchronization. At the end of every epoch, each Concordance node process generates a snapshot of the state - or a block in the blockchain - and computes the hash of the state of the computations. These hashes are compared among the nodes and must be

unanimously bit-identical for the computation to proceed. Otherwise, the new block is abandoned and recomputed with the addition of “watcher” nodes (proven to be uncompromised).

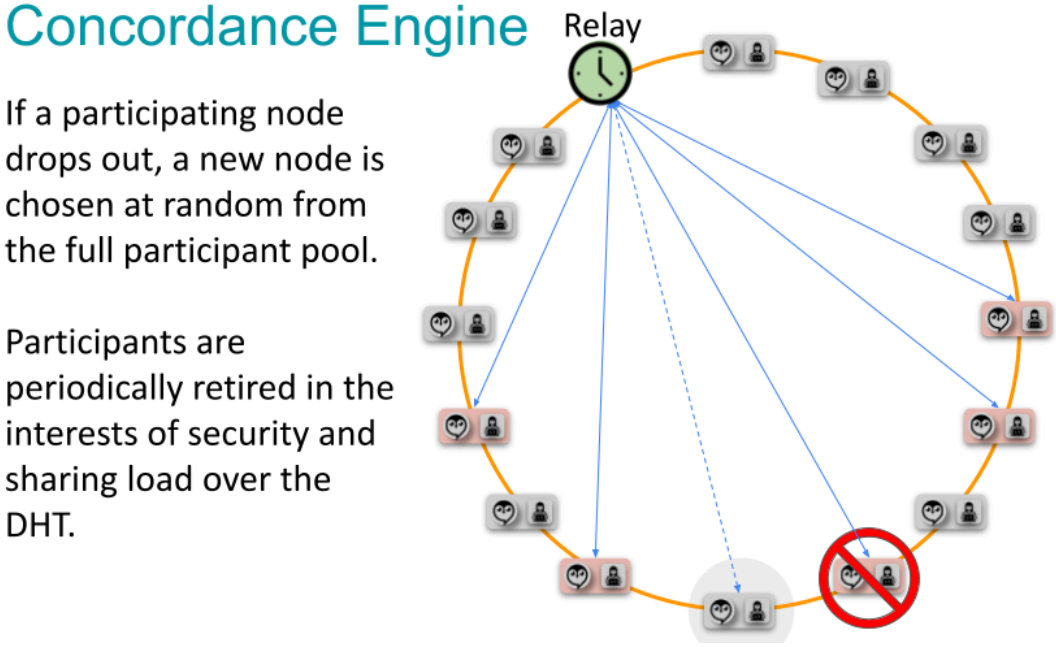
Publish/Subscribe. Utilizes the same pub/sub model of the DHT with particular emphasis on the replicated timestamps.

The Concordance Engine kernel is written in JavaScript and Rust via WASM, and both the Relay and participant systems are included in every YZ DHT node. This kernel runs bit-identically and deterministically on any device that supports JavaScript and WASM, providing a perfectly synchronized compute environment on every participant's machine.

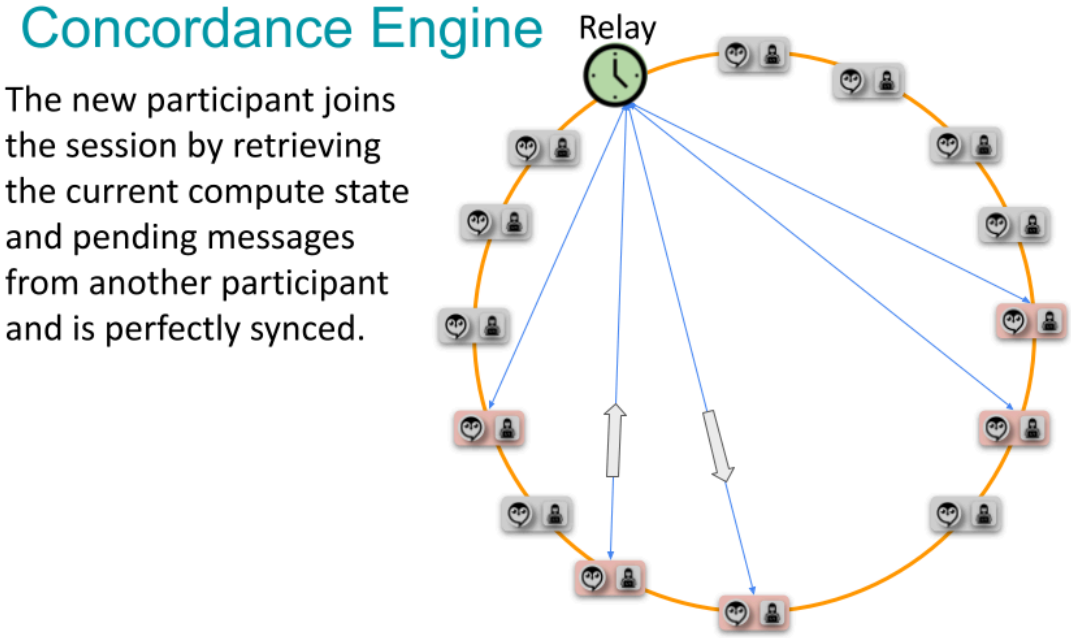
Fault Tolerance and Redundancy

Redundancy

Churn is addressed by simply selecting new participating nodes to join the concordance compute session - usually beforehand, like alternates in a jury pool. If a session participant node drops out, a new participant node is chosen at random (subject to reputation). One of the existing participant nodes takes an instantaneous snapshot of the state of the computation and sends it via the relay to the new participant. That participant is then able to continue the bit-identical replicated computation from that point along with all the others. Nothing is lost. This is a particularly important capability. It means that over time, all of the original compute nodes of a given process may be replaced by new nodes - but the process lives on, as long as it is required and is provided the proper resources. **A virtual “ship of Theseus”.**



Taking a snapshot of the current state of the virtual machine is also a replicated event which is run on every participating node at exactly the same virtual time. Each node generates a hash of the state; by confirming that all nodes have provided the same hash we ensure continuing bit-identical synchronization. This snapshot is performed at regular intervals in addition to when a new participating node joins the concordance. In addition to the snapshot, a new session encryption key is generated. The previous key is then shared with the Relay node and its backups, as well as published to anyone interested in following the computations.



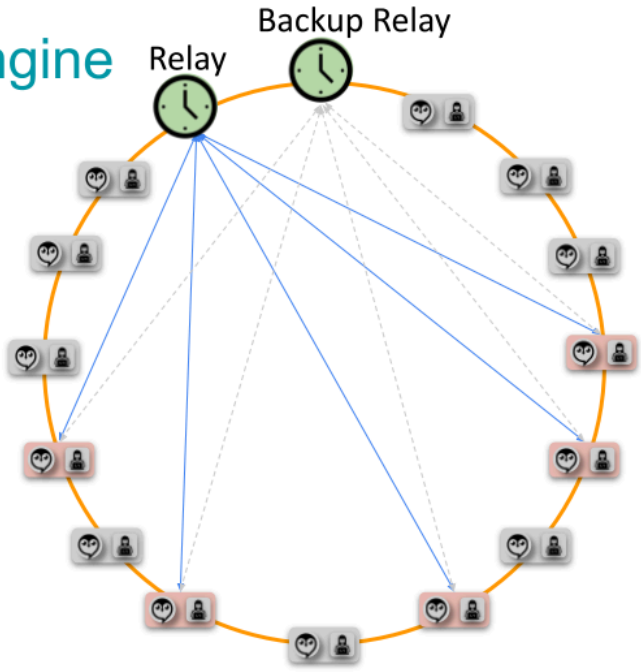
The identical operation of the virtual machine for every participating node means that a programmer can approach the development as if programming for a single computer, dramatically simplifying the programming task. Further, there is no need to be concerned about node churn interfering with the computation process.

Backup Relays

Every concordance includes multiple Relays: a main Relay and backup Relays. If the main relay is lost, one of the backup relays moves to main and continues operation seamlessly. The backup relays also ensure that the main relay is operating properly by receiving the same messages in the same order from the participants and constructing a hash from the array of these messages that the main and backup relays must match.

Concordance Engine

There are also backup relays, ready to take over instantly if the current relay drops out.

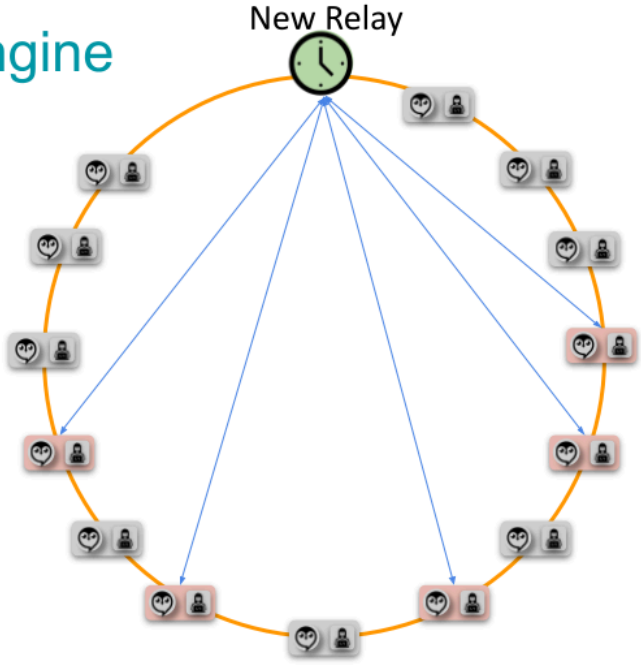


When the main relay fails, the backup relay can instantly take over.

Concordance Engine

Since the backup relay has access to the same set of event messages, nothing gets lost.

Relays are also changed regularly for the same reason as participants are changed.



Relay Security

It is essential that the Concordance Engine Relay have no read access to unencrypted event messages that are processed in the engine. Otherwise, the Relay is in a position to re-order or censor messages. Further, we require a proof that the main Relay is sending exactly the same messages it receives, adding just its timestamp. Finally, we need to ensure that we are not dealing with a malicious pub/sub node.

This is addressed in the following way:

- The validating nodes compute a new replicated encryption key whenever an alternate participant is promoted. The Concordance Engine supports the ability to compute a replicated random value, based on the entropy of the computations within the ongoing session. All participant nodes compute exactly the same value, and this is used to compute identical encryption/decryption keys. Thus, any participant node in the Concordance can post an event via the Relay such that the Relay has no access to the data contained in that event. Relays do not maintain state, aside from timestamps. They have no ability to read the participant messages or the external published messages; they can only add a timestamp and broadcast the encrypted event message to all of the participants.
- Every participant node in the session subscribes to the Concordance Engine's pub/sub location. Messages posted there are forwarded to the participants which in turn, if they have not already processed the message via the Relay, forward the message to the Relay which adds a timestamp and nonce to the message and forwards it back to all the participants including the original sender. Since there are multiple pub/sub nodes forwarding the messages, it is extremely difficult for a single malicious pub/sub node to censor or modify the messages without this being readily apparent when the messages are processed by the participant nodes.
- To ensure the main Relay doesn't modify or censor messages, the system uses a cross-checking mechanism:
    1. Participant send their encrypted messages to both the main Relay and the backup (alternate) Relays.
    2. The main Relay forwards the message back to the participants where the sending participant is able to verify that the message is unmodified from the one it sent to the Relay. The sending participant can also raise an alert if the message it sent to the Relay is not then returned to it after a suitable (short) delay.

3. All participants and all Relays (main and backups) construct an identical 'message block' from these encrypted messages. (This is a separate process from creating the transaction block).
4. Everyone (participants and relays) computes the cryptographic hash of their message block and compares the results.

This verification step confirms that the main Relay has forwarded the messages back to the participant nodes exactly as it received them, guaranteeing the data's integrity and preventing censorship. The Concordance encryption keys are shared with the Relays and published to the DHT whenever a new snapshot is generated of the Concordance state - which may be at regular intervals, or when a new participant node joins the Concordance.

Communicating with a Concordance

A concordance does not have a location on the DHT; it only has an ID.

The Concordance Engine uses the YZ network publish/subscribe mechanism. When a concordance is constructed, or when the main relay changes, the participating nodes subscribe to the concordance ID mapped to a neighborhood ID. This ID could be something like the hash of the string “David’s Process” concatenated to David’s node public key.

When a user wishes to send a message to the process they publish it to the pub/sub neighborhood for “David’s Process+key” with the “signal” event, and the first node the message reaches that has that subscription entry will, once the message is validated from another pub/sub node, forward the message to the Concordance participants which then forwards the message to the relays.

If a user is interested in following update messages from the concordance, they can themselves subscribe to the same domain with an “update” event tag. This publish message can be generated by the participants or forwarded to the Relay to publish.

References

Thomas Paine: **Common Sense**
https://billofrightsinstitute.org/primary-sources/common-sense

John Perry Barlow: **A Declaration of the Independence of Cyberspace**
https://www.eff.org/cyberspace-independence

Chris Dixon: **Read Write Own**
https://readwriteown.com/

Petar Maymounkov, David Mazières: **Kademlia: A Peer-to-Peer Information System Based on the XOR Metric**
https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf

Dan Boneh, Ben Lynn, Hovav Shacham: **Short Signatures from the Weil Pairing** (BLS Signatures)
https://www.iacr.org/archive/asiacrypt2001/22480516.pdf

Peter Triantafillou, Ioannis Aekaterinidis: **Content-based Publish-Subscribe Over Structured P2P Networks.**
https://homepage.divms.uiowa.edu/~ghosh/triantafillou.pdf

S2 Geometry Library
http://s2geometry.io/

Leaflet: Mobile-Friendly Interactive Maps
https://leafletjs.com/

Miguel Castro, Barbara Liskov: **Practical Byzantine Fault Tolerance**
http://pmg.csail.mit.edu/papers/osdi99.pdf

Alan Kay, David A Smith, David P Reed, Andreas Raab: **Croquet: A Collaboration Architecture**
https://tinlizzie.org/VPRIPapers/tr2003001_croq_collab.pdf

Wikipedia: **Virtual world framework**
https://en.wikipedia.org/wiki/Virtual_world_framework

Nakamoto, Satoshi: Bitcoin: **A Peer-to-Peer Electronic Cash System**
https://bitcoin.org/bitcoin.pdf

Buterin, Vitalik: **A Next-Generation Smart Contract and Decentralized Application Platform** (Ethereum Whitepaper)
https://ethereum.org/whitepaper/

Mariano Sorgente: **Don't trust, verify: An introduction to light clients**
https://a16zcrypto.com/posts/article/an-introduction-to-light-clients/

NIST: **Post-Quantum Cryptography**
https://csrc.nist.gov/projects/post-quantum-cryptography

Signal social network encryption
https://signal.org/docs/specifications/pqxdh/

Philip Rosedale: **Fairshare: a self-regulating digital currency managed by groups**
https://www.fairshare.social/_files/ugd/f25890_f49e6ba53c734c2faf5d2fb79a54b5e4.pdf

Bruce M. Boghosian: **Is Inequality Inevitable?**
https://www.scientificamerican.com/article/is-inequality-inevitable/

Adrian Devitt-Lee, Hongyan Wang, Jie Li, and Bruce Boghosian: **A Nonstandard Description of Wealth Concentration in Large-Scale Economies**
https://epubs.siam.org/doi/10.1137/17M1119627

John Perry Barlow: **A Declaration of the Independence of Cyberspace**
https://www.eff.org/cyberspace-independence

Thomas Paine: **Common Sense**
https://oll.libertyfund.org/pages/1776-paine-common-sense-pamphlet

Proof of Humanity Platforms
https://www.humanity.org/protocol

https://world.org/world-id

https://www.identity.com/

Braided Blockchain Compared with Existing Sharded Blockchain Systems

There are two key distinctions between existing sharded blockchain systems and the Braided Blockchain architecture. First, the Braided Blockchain relies on the Concordance Engine to provide an auditable fast-path designed for ultra-low-latency, high-throughput, and rapid recovery, rather than the multi-round prevention-focused consensus of BFT systems. Second,

there is no need for a central relay or main chain. The Braided Blockchain relies on peer shards to provide the required proofs of legitimacy.

Polkadot: Polkadot uses a central Relay Chain to finalize transactions and provide shared security for its independent Parachains. The YZ Braided Blockchain is a single, sharded infrastructure built with the Concordance Engine, where all participants are miners/users, and its fast-path synchronization is based on a single, blind relay and a "trust but verify" model, avoiding the multi-round BFT required for cross-chain finality.
https://polkadot.com/papers/Polkadot-whitepaper.pdf

Near Protocol: Near uses Nightshade to partition the state into shards, and the main chain aggregates the shard sections (chunks) which are secured by a BFT-like consensus (Doomslug). The YZ system's Concordance Engine uses a deterministic replay model, where a single blind relay orders events for deterministic clients, achieving ultra-low latency by avoiding the multi-round coordination inherent in BFT consensus needed for chunk finality.
https://pages.near.org/papers/the-official-near-white-paper/

Elrond / MultiversX: Elrond's Adaptive State Sharding partitions the network state and processes transactions in parallel, but it relies on a consensus mechanism (Secure Proof of Stake) to secure the network. The YZ system's sharded blockchain uses the Concordance Engine's deterministic state hashing and independent observer logs to provide auditability and secure synchronization, trading the formal prevention guarantees of BFT for faster detection and correction (rollback/replay) of faults.
https://cdn.multiversx.com/webflow/multiversx-whitepaper.pdf

Zilliqa: Zilliqa partitions its network into shards for parallel transaction processing, with a separate root chain to confirm and finalize the results from the shards. The YZ braided blockchain is fundamentally different as its core technology, the Concordance Engine, provides a high-speed, replicated computation platform that achieves consensus by requiring unanimously bit-identical state hashes from all participants at the end of every epoch, which is a different approach than the cross-shard finality mechanisms of BFT-based systems.
https://docs.zilliqa.com/whitepaper.pdf

Ethereum 2.0 (The Merge and beyond): Ethereum's sharding plan focuses on Data Sharding to increase data availability for Layer 2 rollups, with a central consensus layer (the Beacon Chain) securing the network. The YZ Braided Blockchain is an execution and state sharding architecture built on the Concordance Engine, whose high-performance, deterministic model is explicitly designed as an alternative to the BFT model that secures Ethereum's main chain. The YZ system's developers also note that the Solidity language/Ethereum Virtual Machine was not designed to operate in their sharded system.
https://ethereum.org/roadmap/danksharding/

Monad: Monad takes a monolithic approach but radically optimizes the Ethereum Virtual Machine (EVM) for speed. It solves the problem by maximizing the efficiency of *single-chain* execution through parallel execution and pipelining. Its key value proposition is offering Ethereum's security and decentralization with Solana-level speed and full compatibility for existing Ethereum developers. **YZ.social** takes a **sharding approach** to scalability. It solves the problem by breaking the single global committee into many smaller, faster, interconnected chains. This is a radical departure from the monolithic chain design, focusing on *inter-chain* consensus security (Braid of Trust) rather than single-chain speed.