Fleek Network: Decentralized Edge Platform

Whitepaper - July, 2023

https://fleek.network/

Fleek Network Whitepaper

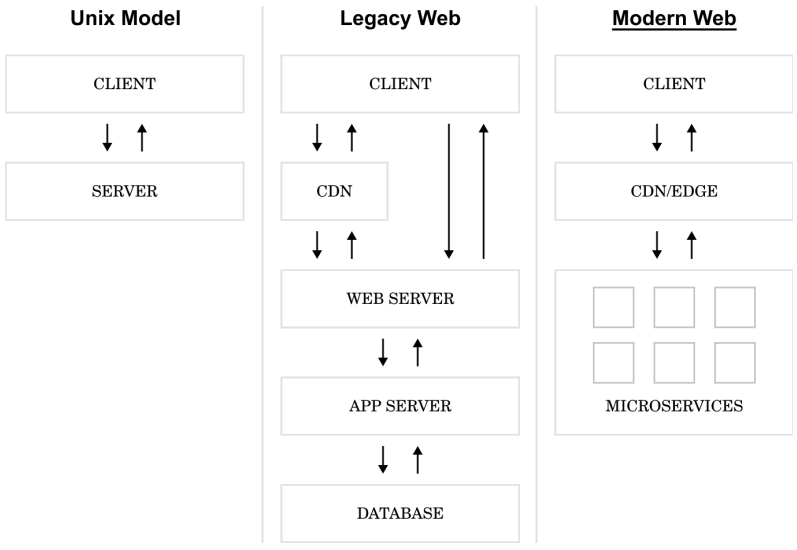
Abstract

Fleek Network is a decentralized edge platform optimized to facilitate the deployment and running of performant web services (CDN, serverless functions, etc.). Fleek Network's globally distributed and autonomously controlled network of edge nodes allows developers to frictionlessly create and utilize a multitude of edge services. These services inherit cryptographically and economically secured infrastructure, node and geographic coverage guarantees, stable and predictable costs, and consistent quality and performance across all services running on the network. Fleek Network's goal is to provide a platform that all Web3 protocols, middleware, services, and applications can benefit from to further decentralize their stack without sacrificing cost, performance, complexity, or developer/end-user experience.

Background: Evolution of the Web

Modern Web Stack Evolution

The web stack has evolved substantially. Moving to the 'cloud' is quickly being replaced with moving to the 'edge'. While this trend does have security and cost benefits, the main reason for the migration to the edge is performance/latency. Latency and load times have an enormous impact on internet users and their usage of services/apps, and therefore also on the developers building them (and also the ones building the infra/middleware powering them). As Google's research showed, the chance of a bounce increased by 32% when a page load time went from one to three seconds, and by 90% when the page load time went from one to five seconds. Combining that with the current usage/growth of content/streaming oriented services, online gaming's exponential yearly growth, and the rise of AI/AR/VR, the demand for low latency optimized web infrastructure will only increase.

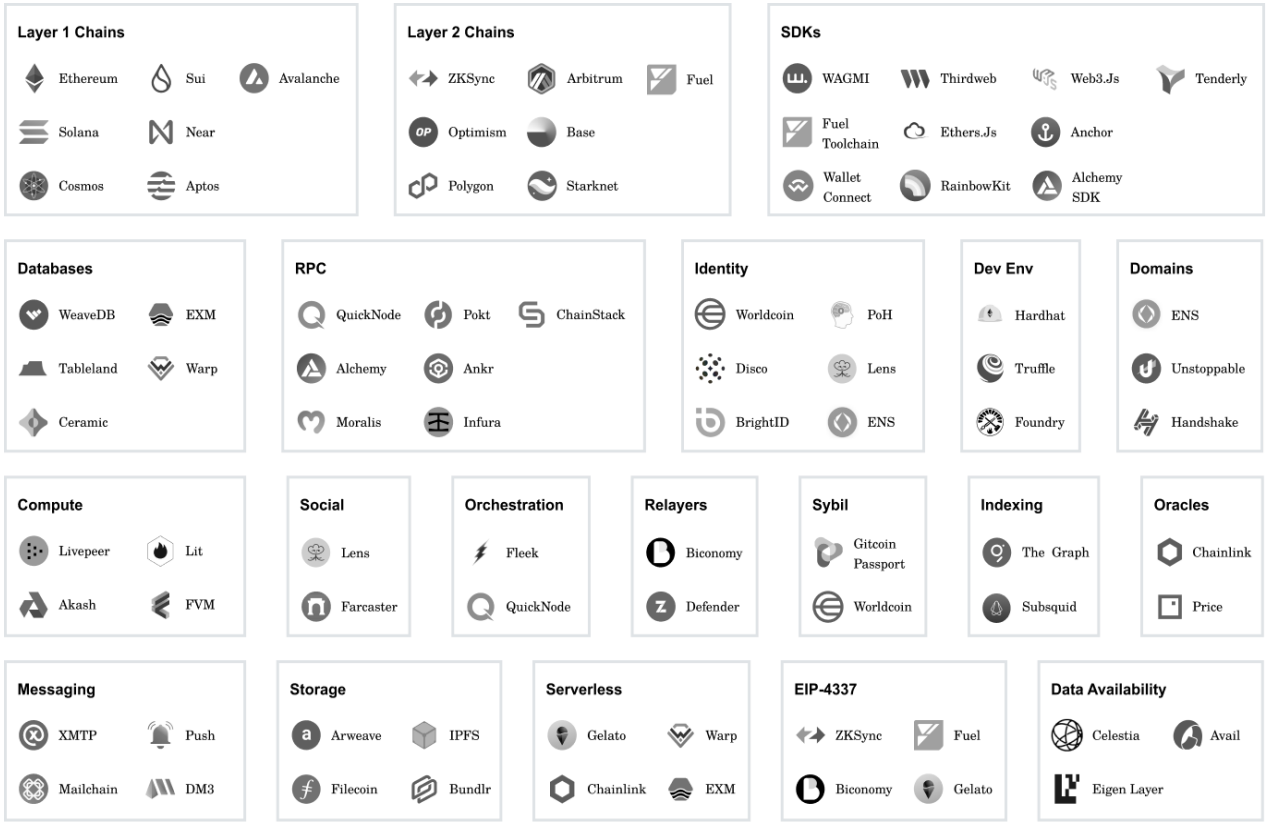


Edge Networks are an evolution of the concept and success of content delivery networks (CDNs) which serve content from multiple geographic locations depending on the location of the end user. The goal is to reduce latency by performing the work (e.g., serving content) as close to the end user as possible. This pattern is now being applied to other types of traditional web infrastructure and services well suited for the edge, such as compute (ex. serverless functions, server-side rendering), databases (ex. CRDT), DNS, container orchestration, etc.

The move to the edge has also been accelerated by modern frontend frameworks like React, Next.js, etc. These frameworks, often referred to as the Jamstack, are an architectural approach that decouples the web experience layer from data and business logic. Decoupling experience and logic improves flexibility, scalability, performance, and maintainability, and provides a composable architecture for the web where custom logic and 3rd party services are consumed through APIs.

Web3's Modular and Composable Evolution

Similar to the modern web's evolution, Web3 is also quickly moving towards a modular and composable future. The industry is witnessing the emergence of numerous specialized protocols and middleware that offer specific services spanning the entire infrastructure stack. Previous 'monolithic' architectures are now being decoupled and broken apart into smaller, stand-alone modules that interoperate.



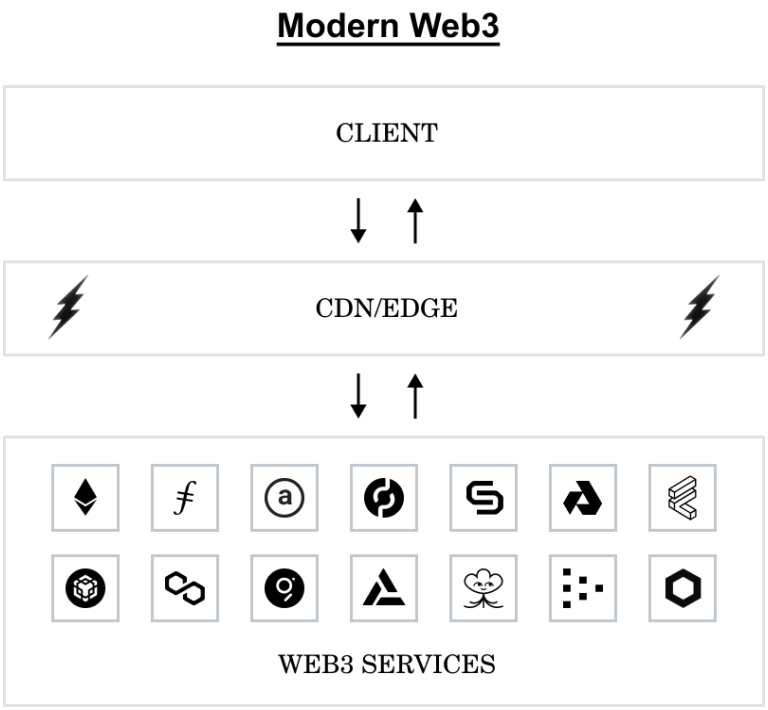
However, this trend in Web3 is still in its infancy, and there are still a lot of inefficiencies and redundant work being done across the space. The absence of a Web3 cdn/edge network forces almost every Web3 protocol, middleware, service, and app to choose between the following two sub-optimal paths:

A) Utilize corporate-controlled cloud infrastructure within their stack (ex. AWS or Cloudflare) to achieve ‘web2-like’ performance (a requirement for most developers and end users today), which often creates a central point of failure in their stack and significantly weakens their decentralization, censorship resistance, and permissionless-ness; OR

B) Attempt to build all of these performance optimizations into their network/economic model/nodes/stack (specialized networking & p2p, geographic and reputation-based routing, algorithmically determining what nodes should do what work/serve what content, load balancing requests, etc.), which is highly complex, time-consuming, and inefficient for every project to try to build themselves.

Bringing the Modern Web to Web3

Introducing a highly-performant decentralized edge network into the Web3 stack as a shared infrastructure, orchestration, and performance layer could be useful to almost every Web3 project (protocols, middleware, services, apps). It could help bring centralized web-like performance to web3 without sacrificing on web3 values.

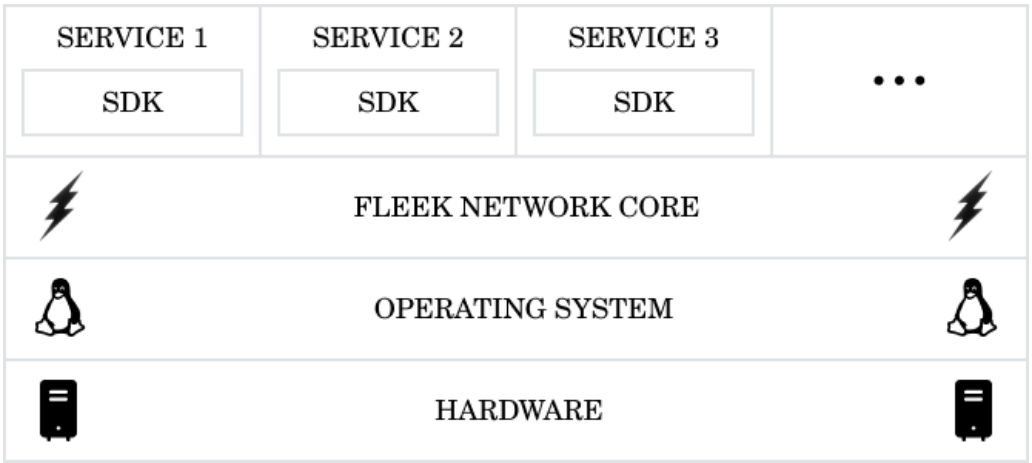


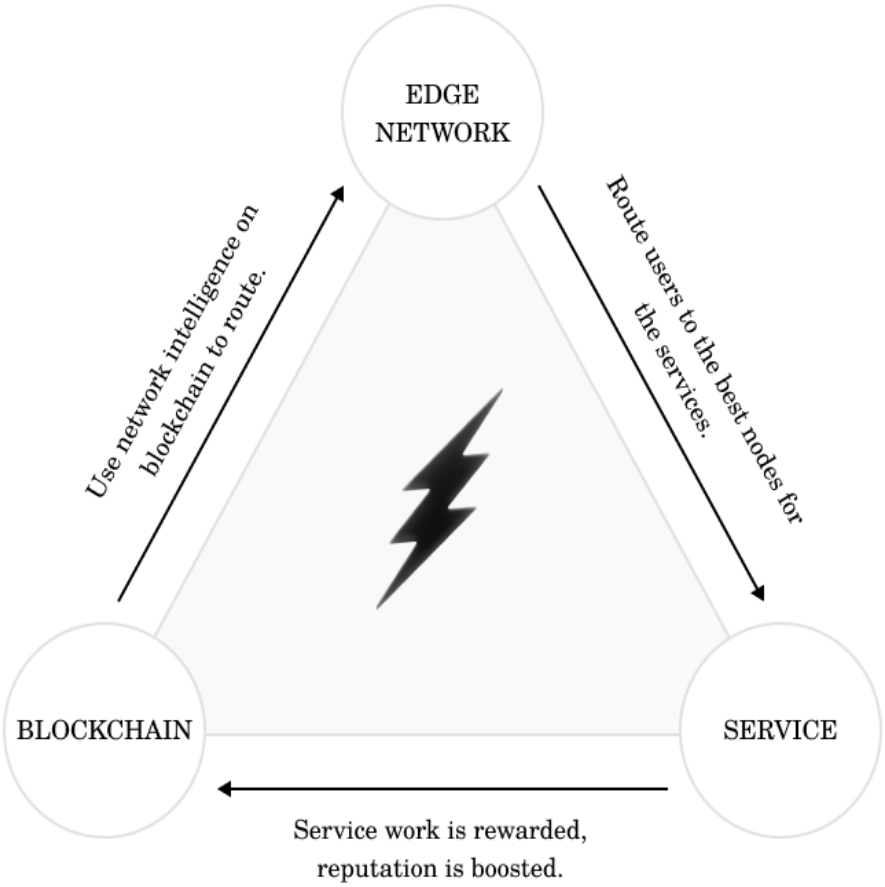
The shared edge infrastructure could also help lower barriers to entry and speed up time to market for developers building web3 infra/middleware by offloading a portion of their stack to the decentralized edge rather than needing to build that functionality into their own network. This would allow them to focus more time/resources on the aspects of their products/services that are unique to them, and leave the performance/latency optimizations to the edge layer above, similar to the modern web.

Fleek Network: Decentralized Edge Platform

The goal of Fleek Network is to provide an efficient, trustless, decentralized edge network that enables a multitude of edge services. Given the vast amount of edge services that can be built, the intent of the Fleek Network core is to be a foundational layer that could allow anyone to develop and deliver new edge services quickly and without needing to worry about the non-trivial aspects like networking, smart routing/work allocation, load balancing or any other layer that is not a core feature/functionality of their service. Fleek Network achieves this by making the network's core geographically aware and optimized for speed.

The network's core is minimal and focused around efficiently verifying succinct Delivery Acknowledgement SNARKs (expanded upon in the Protocol section) that nodes batch and submit to the protocol. All service-specific logic is handled outside of the core protocol. Fleek Network believes this architecture can be a powerful addition to the Web3 stack and useful for building and/or performantly optimizing decentralized web infra/services. And as Fleek Network grows and its global coverage increases, all services built on the network benefit equally.



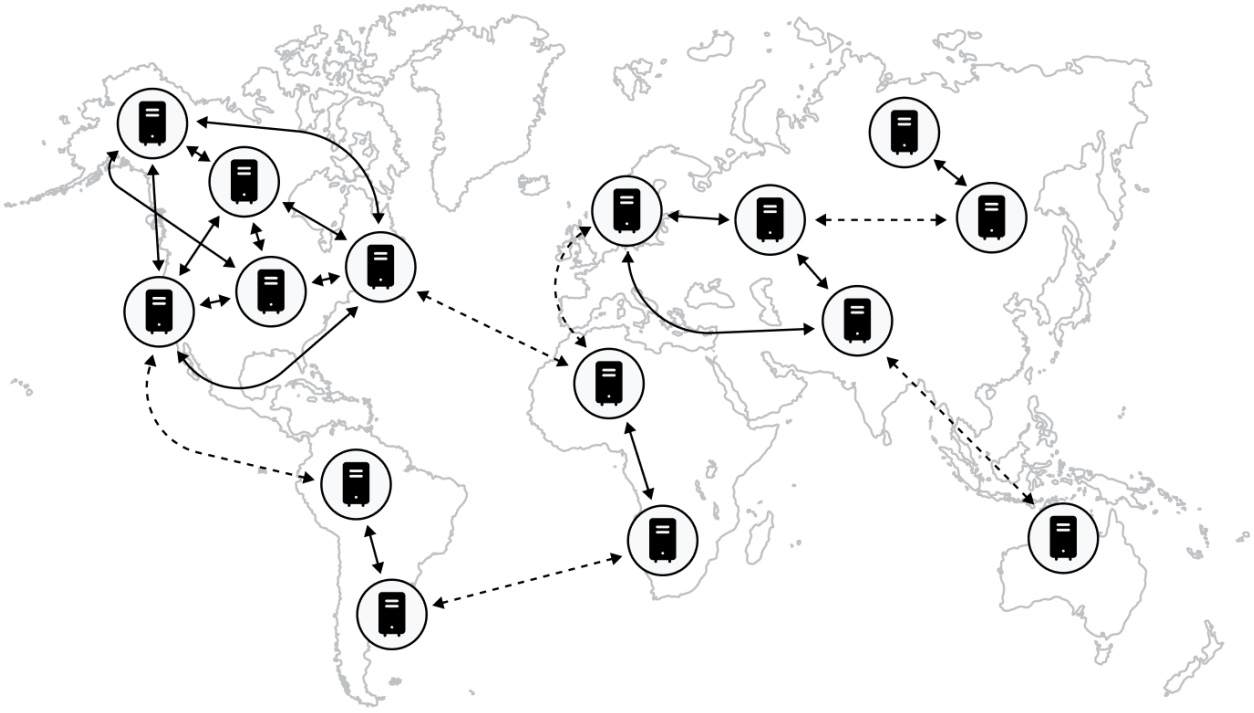


* Fleek Network nodes share a common core/consensus. Services are loaded and run on-demand based on requests made by clients. The network intelligently orchestrates which nodes run which services and perform what work each epoch based on request data from clients in their geographic vicinity.

Key Concepts & Performance Optimizations

The foundation of an edge network is high performance, low latency, and geographic work distribution. These are characteristics that most decentralized systems struggle with at the base protocol layer. The sections below are a description of the key concepts and noteworthy performance optimizations baked into Fleek Network that allow us to achieve the performance requirements of an Edge Network:

Geographic Awareness



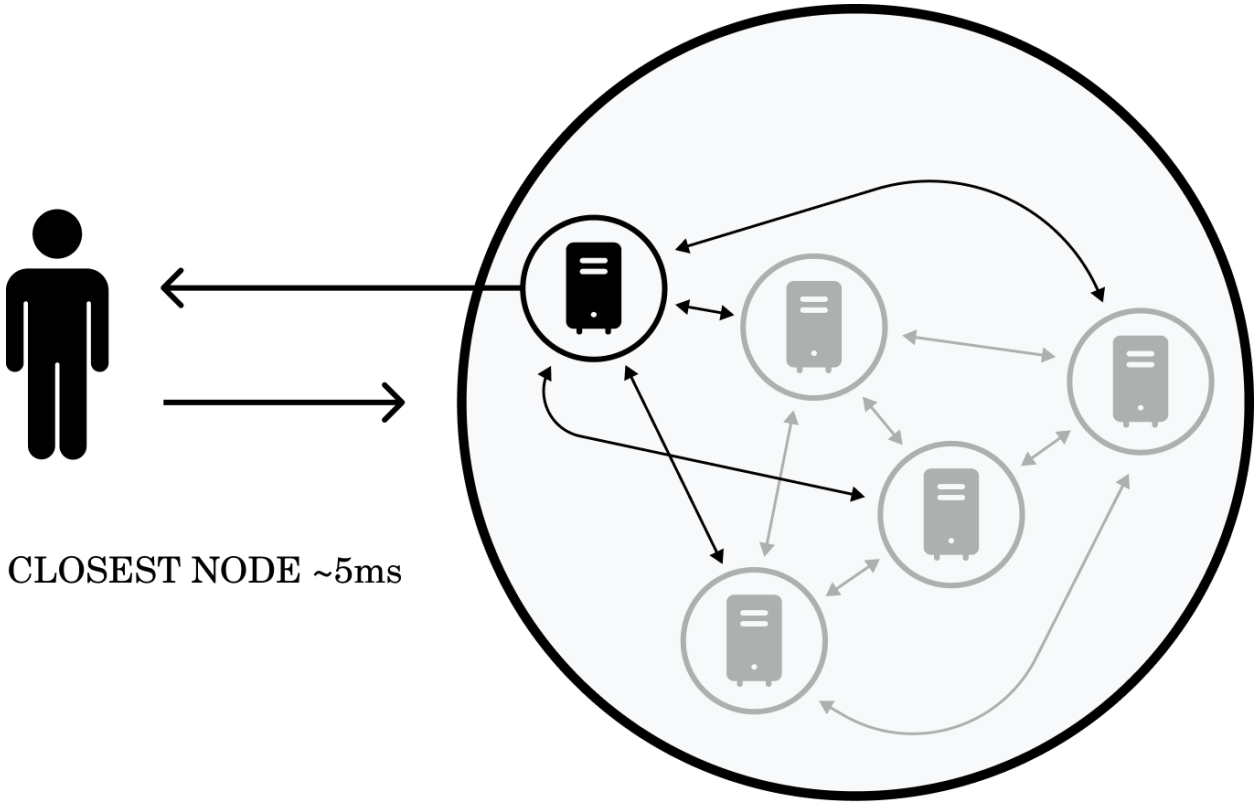
Nodes create connections with their lowest latency peers, and only connect with further options when needed/ forced to.

While the network does not possess a specific concept of geography, it gains an implicit understanding of geographical proximity through the data it collects on latency and hop counts between nodes, which is a part of the reputation system. Three key aspects contributing to geographic awareness are:

I. Nodes that exhibit low latency and hop counts with each other are considered to be closer in proximity.
II. Periodically, each node shares a set approximation of the most popular (i.e. most requested) content it holds with its k-closest peers.
III. A Cuckoo filter is employed as a probabilistic data structure for set approximation to achieve this.

This approach seamlessly integrates with other system components and effectively enables Fleek Network nodes to distribute workloads among nearby/best peers.

Smart Routing & Work Allocation



The network intelligently routes the client's request to the lowest latency/highest reputation node available in the geographic vicinity of the request, to ensure the user always gets a fast/geographically optimized response.

Many decentralized systems rely on stake-weighted work allocation and gossiping mechanisms to facilitate broadcast and communication functions. The algorithms of those networks typically operate without assuming prior knowledge about the reliability or geographic location of other nodes in the network. Even if data about reliability is collected during runtime, it is often used as a best-effort mechanism and not as a setup parameter.

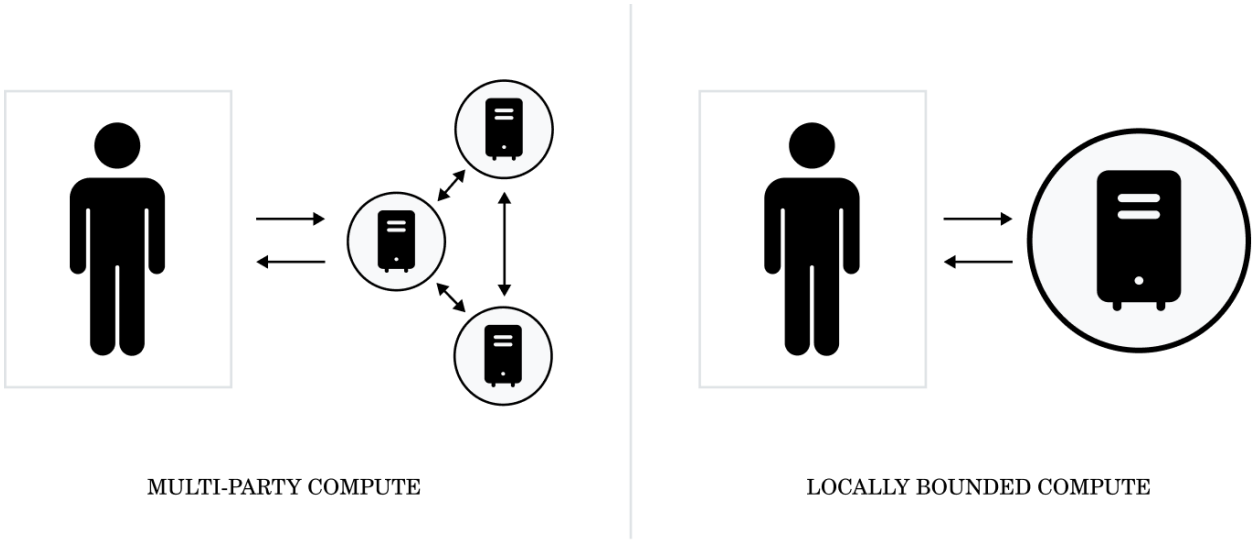
In comparison to most other decentralized systems, Fleek Network uses a unique setup:

1. The network's nodes remain fixed at every epoch, with no new nodes joining until the start of the next epoch (given the node completed their onboarding).
2. There are reputation statistics stored and replicated in the application state that all nodes can access.
3. All nodes stake the same amount, and work allocation is strictly based on geography and reputation (i.e stake amount has no impact on how much work a node receives).

Leveraging these additional assumptions enables significant performance optimizations of the gossip and broadcast layer of the network, and reduces latency of requests by routing work to nodes closest to the end user requesting it.

In essence, the protocol utilizes this information and employs a deterministic algorithm to create an efficient and fault-tolerant connection graph for the entire network. At the beginning of each new epoch, nodes in the network can converge on the same overall network connection graph.

Stateless Execution



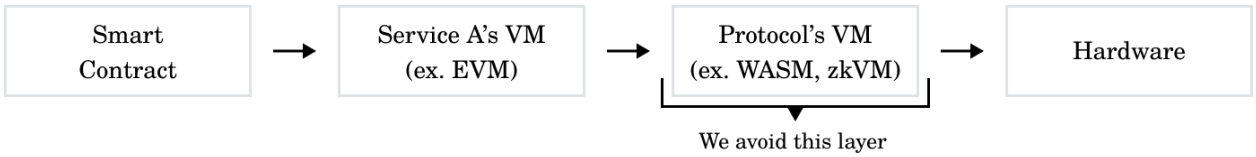
The state of the execution environment of Fleek Network is stateless by default. Locally bound to the edge nodes running the service.

By default, the state of the execution environment of Fleek Network is locally bound to the nodes running a service. This makes Fleek Network an optimal choice for creating services where stateless computation and execution could benefit the end user.

Keeping the execution environment globally stateless (locally bounded) helps the network maintain maximum performance and low latency, but it also allows us to shuffle services across nodes each epoch to ensure decentralization, and it significantly reduces the ability for nodes to collude or act maliciously. This also contributes to the network's ability to autonomously optimize which nodes perform what services each epoch.

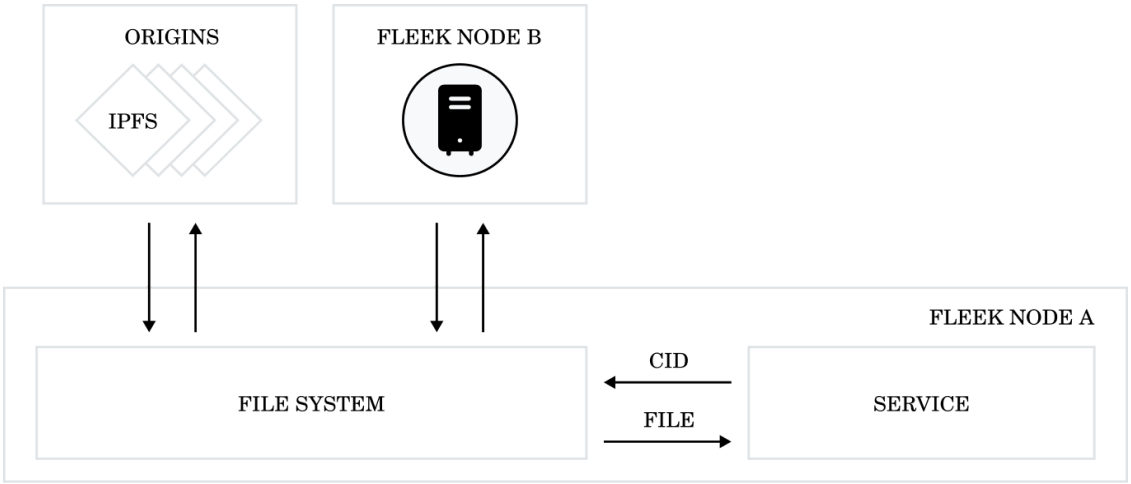
VM-Less Core

While developers can build/utilize VM's at the service layer (explained in the Services section), Fleek Network's core protocol does not use a VM. The VM-less core enables more efficient usage of node resources for services, and results in less constraints for developers who are building services. For example, sometimes non-EVM protocols implement the EVM as a smart contract/service running on their protocol in order to achieve 'EVM compatibility'. However this results in inefficiencies by essentially requiring two VM's (the EVM running in the protocols native VM). By contrast, on Fleek Network this wouldn't be the case, and various VM's could be built as services that could consume edge node resources directly without any unneeded overhead. While this architecture does introduce new security considerations, Fleek Network is following similar strategies to those used in modern containerization software, such as cgroups, selinux/apparmor, and other associated kernel level solutions. The final details of the sandboxing system will be published separately and are out of the scope of this paper.



The VM-less core enables more efficient usage of node resources for services, and results in less constraints for developers who are building services.

Built-In (Externally Powered) File System



Fleek Network natively integrates external decentralized storage protocols (IPFS, Filecoin, Arweave, etc.). This modular approach increases flexibility while reducing vendor lock-in, and enables Fleek Network to optimize for higher performance/lower latency by off-loading storage and keeping edge nodes lean.

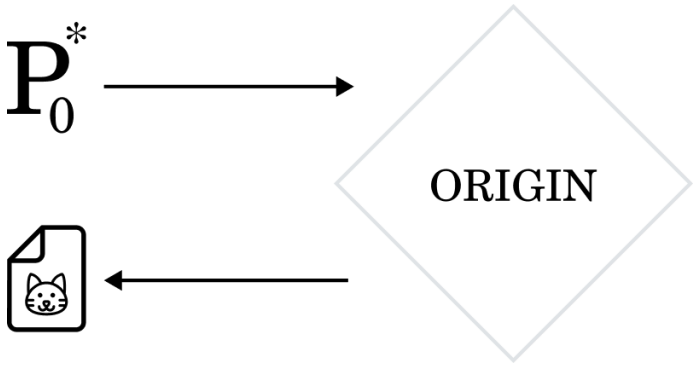
Fleek Network has a built-in file system powered by multiple external decentralized storage protocols (ex. Filecoin, Arweave, IPFS, etc.). By embracing modularity and leveraging existing external file storage protocols rather than trying to build storage into the network, the nodes can be kept extremely lean, enabling the network to focus on optimizing performance and low latency much more so than a network whose nodes need to carry storage, and reducing lock-in risk for developers using the network in terms of storage/data layer choices.

Services operating on Fleek Network can leverage the SDK to access the shared content retrieval component efficiently. For example, a service builder can readily implement compute-over-data services, similar to how such a service would be created in a centralized setting.

Fleek Network stores certain limited state in the network (see Succinct Chain State section), but any other data storage requirements services are handled through either these built-in storage options or any other external option the service wants to use (ex. any other web3 protocol).

Content Addressable Core

Fleek Network uses Blake3 hashing for efficient content identification and streaming verifiability. The entire network operates based on content addressing. In addition to its file system, Fleek Network incorporates a Distributed Hash Table (DHT), which enables the network to store a flexible mapping from any "immutable data pointer" or IPLD (Interplanetary Linked Data) to its corresponding Blake3 hash. An immutable data pointer refers to a reference to external data that remains consistent over time.

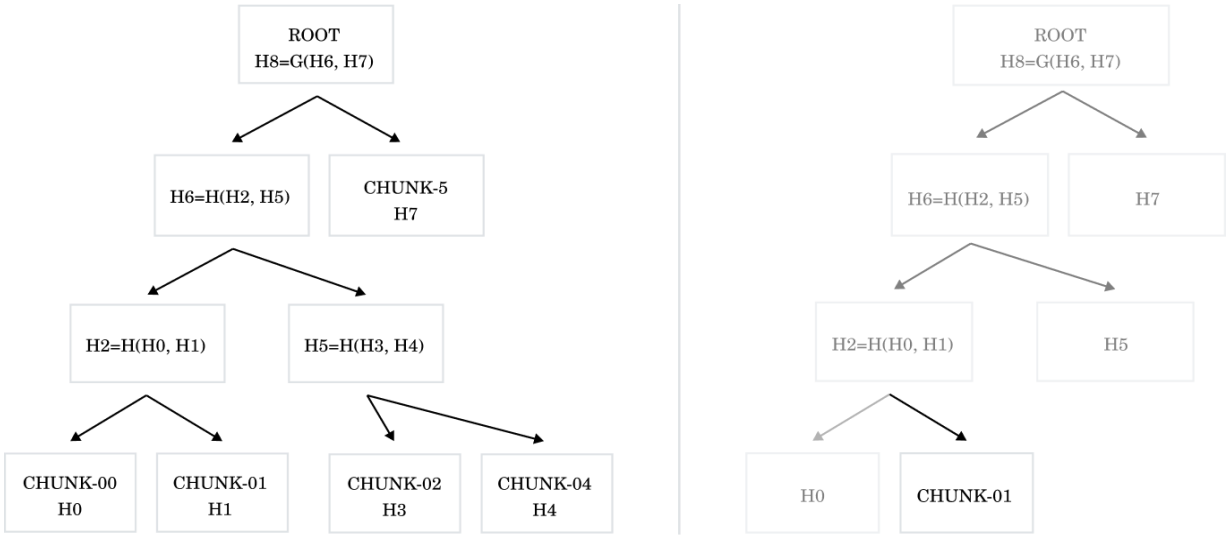


Immutable data pointers serve as a broader concept of content addressability. For instance, if a data storage protocol utilizes sequential IDs to identify content and prohibits future modifications, those sequential IDs can function as immutable pointers. In formal terms, an immutable data pointer is a string that permanently corresponds to the same content.

It is worth noting that any mutable pointer (like a regular HTTP link) can be transformed into an immutable pointer by appending an integrity hash to the URI. These kinds of immutable pointers will be ephemeral, and if the pointer goes stale (ie, the origin has been responding with something else for a length of time) it could get dropped from the DHT (cache invalidation).

Fleek Network acts as a resourceful caching layer, mitigating the limited performance of data retrieval from most web3 protocols/middleware services. It considers the geographical distribution of nodes and the popularity of files/data in specific regions. The network ensures efficient data retrieval by replicating the cache among nearby nodes.

Incremental Content Retrieval & Verification



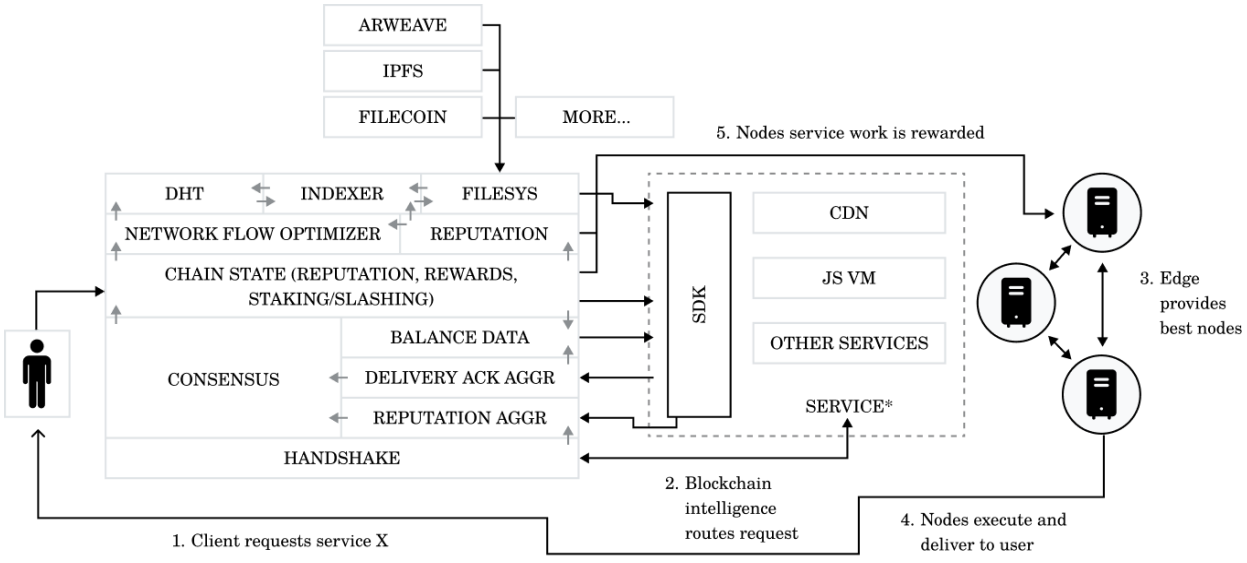
Using Blake3 we can chunk and stream data, while verifying each chunk against the root CID.

The tree structure of Blake3 allows for the efficient employment of verifiable content streams. While the authors of Blake3 have implemented a library called Bao, the performance requirements and customization needs of Fleek Network prompted us to develop a replacement for Bao, which is extensively used in this development.

This replacement utilizes precomputed data trees with larger block sizes and offers utility functions to efficiently prune these trees and create minimal subtrees when streaming content blocks. This approach enables significant performance optimizations, making the cost of incremental verification for each node virtually insignificant.

Fleek Network: Protocol

Fleek Network is a proof-of-stake Ethereum side-chain with its own network of edge nodes that leverages Ethereum for the FLK token (ERC-20), staking, payments, governance, and other economic aspects of the protocol. Node operators stake FLK tokens (and possibly also ETH via EigenLayer) in order to perform work on the network, and developers, service creators, and clients use stablecoins to pay for utilizing the resources and commodities on the network. Edge nodes provide resources (Bandwidth, CPU, etc.) to the network that are packaged into commodities and priced at the network level in USD. Services built on Fleek Network consume whatever underlying resources/commodities required from nodes. A client requests specific services to be run and nodes run these services and perform work to earn fees (full economic details are out of the scope of this paper). Fleek Network leverages a combination of SNARKs (Succinct Non-interactive Argument of Knowledge), Narwhal and Bullshark consensus, as well as other cryptographic and economic incentives and guarantees to achieve a trustless, decentralized, and long-term sustainable environment. The core of the protocol is kept as succinct and un-opinionated as possible so that any given node resources are used to run services. This section elaborates how this is achieved.



This is an overview of the various components integrated in the protocol. To clarify the relationships between these components, we have used different arrows.

1. The black small arrows represent the dependencies between each component inside of the core, highlighting how they rely on one another to function effectively.
2. The black dashed arrows depict the communication channels with external resources or clients on the network, indicating the system's interactions beyond its immediate boundaries.
3. Solid arrows indicate the interprocess communication (IPC) between a service and the SDK, and their instructions/data flow.

Succinct Chain State

At the core of the protocol, the following things are held in a state:

I. Token Balances (FLK token and stablecoins),
II. Staking information
III. Node Reputation
IV. Data on how much work a node has performed in a given epoch.

With a decentralized system, there is a need to replicate this state across all of the nodes in the network. This is done by forming a blockchain and coming to consensus on transactions that transition this state.

Narwhal & Bullshark Consensus

One of the most significant issues faced was preventing this process (ordering and coming to consensus on transactions) from bottlenecking the performant edge network. After considering a few options over the complexities of distributing transactions among nodes efficiently for consensus on the total order, the conclusion was that the research done on Narwhal by Meta and MystenLabs is a uniquely well-suited solution. Narwhal is an efficient DAG-based mempool, and Bullshark allows the network to come to consensus on the total ordering with zero networking overhead.

Every node in the network does not need to perform this ordering, so a committee-based approach was taken. Any properly staked node is eligible to be on the Narwhal committee. At the end of every epoch (roughly 24 hours), using a decentralized random beacon already present in the network, a new committee is formed from a subset of participating nodes. The committee overseeing the transaction ordering process transfers its responsibilities to the new committee. This periodic rotation reduces the risks associated with compromised committees, ensuring the integrity of the consensus mechanism. The rest of the network is focused on doing work, batching Delivery Acknowledgements, and submitting them to the committee to be ordered.

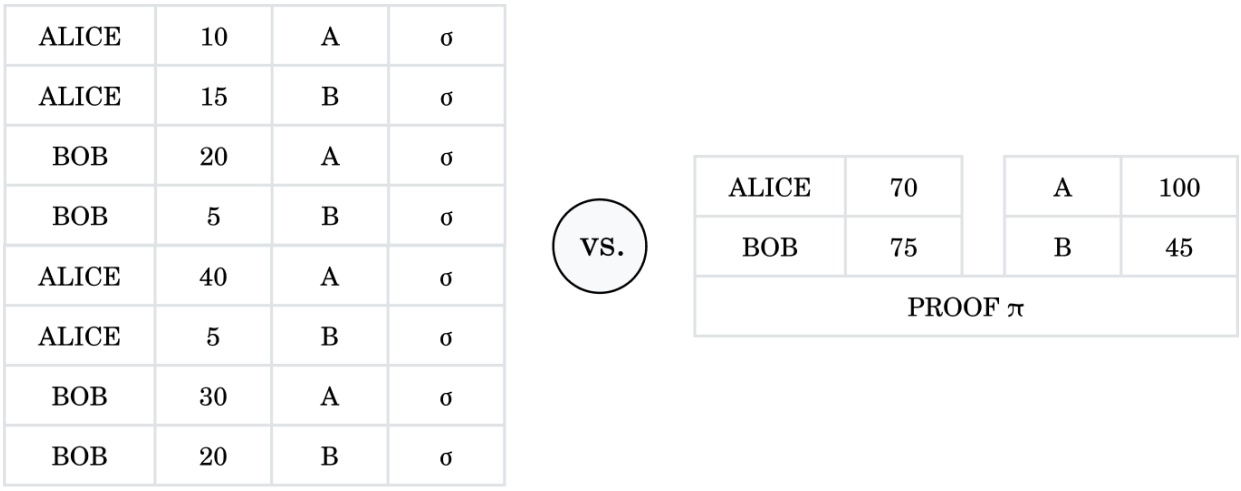
Delivery Acknowledgement SNARKs

The primary transaction being ordered by Narwhal/Bullshark is a batch of Delivery Acknowledgments. As a node is serving requests, it is collecting delivery acknowledgments that, when submitted, will reward the node at the end of an epoch.

Delivery Acknowledgement is the term used to describe a signed message by a client attesting that a node has successfully delivered a task to the client. These acknowledgments are instantly finalized locally and cannot be reverted by the client.

Once a node receives a delivery acknowledgment containing information about the resources that it has served a client, it can add it to a local pool of delivery acknowledgments and periodically send these messages to consensus (and, therefore, every other node) to claim their fees. Once an acknowledgment has been received, the client's pre-paid stablecoin balance is adjusted accordingly. The amounts deducted from all clients during an epoch will move to the payout pool to be distributed fairly to nodes based on the work they performed that epoch.

This periodic submission of delivery acknowledgments allows us to leverage recursive SNARK proofs to aggregate many of these delivery acknowledgments, reducing the networking cost of broadcasting the batch and significantly reducing the computing power required to verify all of the delivery acknowledgments individually otherwise.



Instead of sending all transactions individually (LEFT), we use SNARKs to send the culmination of Alice and Bob's balances, recursively proving their balances as a whole.

Performance Based Reputation

Fleek Network has implemented a reputation system that allows nodes to provide scores for each other. These scores are collected over time, and at the end of each epoch, an aggregation algorithm is run to calculate overall scores for each node in the network (See Appendix A for Algorithm).

To prevent the impact that dishonest nodes may have, Fleek Network borrows insights from the EigenTrust algorithm, a seminal contribution to reputation management in peer-to-peer systems.

A key ingredient to the EigenTrust algorithm is the notion of transitive trust. A node will assign a higher reputation score to peers with whom it had positive interactions. A node is also more likely to trust the opinions of peers that it has assigned a high reputation score to.

In the EigenTrust algorithm, the global reputation of a node is given by the local trust values assigned by other peers, weighted by the global reputations of those peers.

Drawing on this notion of transitive trust, weights are assigned to the reported scores for each node by the reputation of the nodes that reported them. The network also performs outlier removal to further mitigate the impact of false reporting.

Using the reputation of nodes to weight the reported scores prevents Sybil attacks where node operators create many additional nodes for the sole purpose of reporting favorable scores. Furthermore, a high reputation can only be built up over time. The reputation score of a node is the exponentially weighted moving average of the scores that were computed in all the previous epochs.

Reputation data can also be obtained through interactions between nodes while providing services. Each node is assigned a rating for each service interaction. This information is valuable and replicated across the network, serving as a reliable source of knowledge for optimization tasks. Some examples include optimizing network flow and assigning services to different nodes.

NODE A	
FACTOR	*SCORE*
LATENCY	5ms
BANDWIDTH	1mb/s
SATISFACTION	80%
REPUTATION	**95/100**

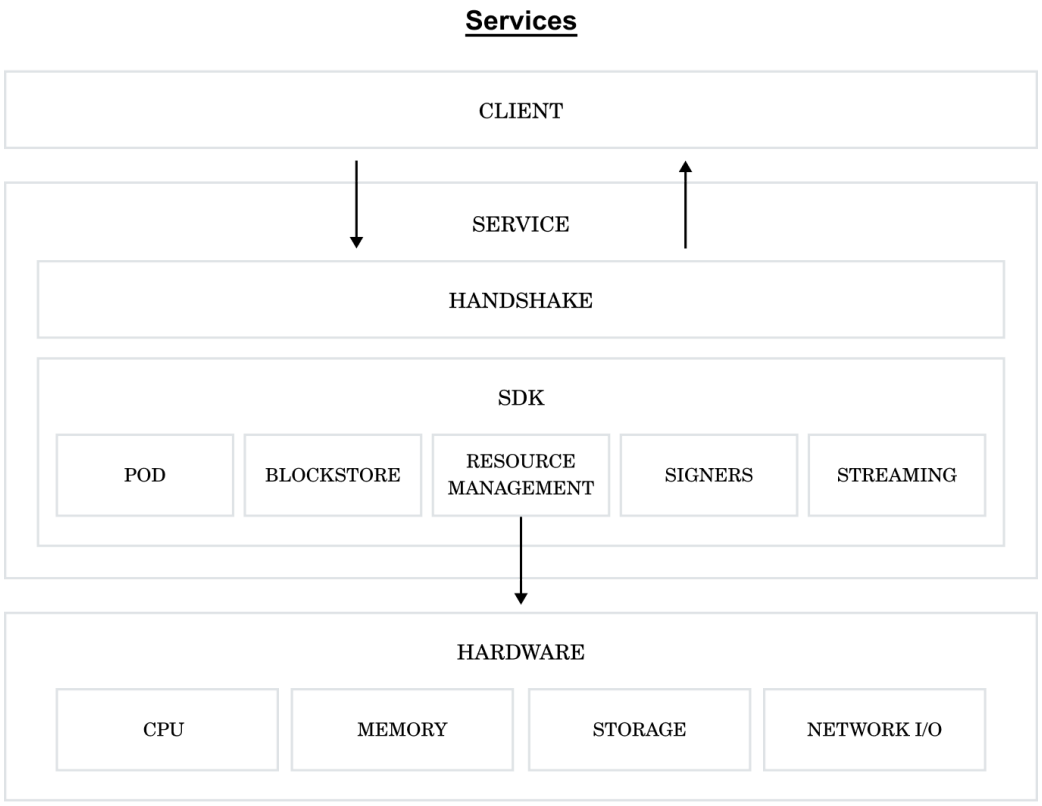
NODE B	
FACTOR	*SCORE*
LATENCY	50ms
BANDWIDTH	50kb/s
SATISFACTION	10%
REPUTATION	**13/100**

NODE C	
FACTOR	*SCORE*
LATENCY	5ms
BANDWIDTH	50kb/s
SATISFACTION	80%
REPUTATION	**69/100**

In this hypothetical/simplified scenario, Node A would likely get the request. Node B would be in danger of getting kicked off the network. Node C would still get requests, just not as many as Node A. Where the requests come from would also influence work allocation.

Fleek Network: Services

A service on Fleek Network is a modular piece of edge-enabled software that runs on (and utilizes the features of) the edge nodes, delivering unique functionality to its end users. These services could be considered as the user interface of the network.



The above is an overview of the basic service architecture and its interactions with both the node and client layers.

Third party developers can permissionlessly deploy their own edge-enabled decentralized web services to the network, which will run in a sandboxed environment (as mentioned in VM-Less Core section). Initially the core services will be statically linked to the node binary, but in the future the plan is to load services dynamically and allow developers to write their service in any system level language and deploy during run time of a node.

Scope-wise, services will only have access to resources made available to it via the SDK. These resources will include things like filestore, cryptographic primitives, and metrics reporting APIs. Services can’t access other parts of the application or processes, nor other services (since they will be isolated), and they also don't have direct access to the hardware or kernel of the edge nodes. Given the permissionless nature of the system, code for the services would be managed in the respective service developers’ repositories (similar to smart contracts).

SDK

To create a Fleek Network service, it is necessary to utilize the SDK. The SDK abstracts the core protocol features exposed for service builders to create services with. It facilitates efficient communication with the middleware through an advanced Inter-Process Communication (IPC) system. One solution could be built on shared-memory objects and a ring buffer, which is very performant, but other options are viable as well.

In essence, any system-level programming language like Rust, C/C++, or Go can be employed to establish this communication channel with the middleware.

The SDK offers several capabilities to services, including the ability to request resources from the middleware. Additionally, it provides service creators with various cryptographic commitment schemes (ex. Delivery Acknowledgement SNARK templates for different commodities). These commitments, generated by nodes, are sent to the client to prove work completion (ex. bandwidth served, compute performed, etc.). The design of these cryptographic primitives is to 1) ensure clients receive the correct work being requested (content, computed response, etc.); and 2) prevent malicious services from coercing a node into making false commitments.

Interacting with a Service

The handshake component in Fleek Network core serves as the entrypoint for external communication with a node, enabling clients to establish a session and interact with various services. The handshake component listens over a secure transport layer, such as TLS+TCP or QUIC, providing encryption of all communication with the node. When the node receives a new connection from a client, public keys are exchanged, and a lane is negotiated for the session. This process only takes a few milliseconds (same as a normal TLS handshake).

Once the handshake is completed and a secure connection is established, the client is free to request access to a service, which becomes aware of the new session and takes over the subsequent interaction with the client. This involves reading messages from the client, performing necessary tasks, and sending back responses. When a session with a service is terminated, the client is free to request a connection to another service or even the same service while keeping the existing connection alive.

Node Assignment & Shuffling

Service creators don't need to worry about infrastructure coverage (geographic distribution, scalability, etc.) nor incentivizing edge nodes to run their service, as the network abstracts and handles that aspect on behalf of all services. Every epoch the network uses various graph algorithms to assign edge nodes in each geographic region to perform the work for every service. These algorithms help determine the most efficient placement of services within the network to maximize resource utilization and overall network performance while providing sufficient and reliable levels of trust, security, infrastructure coverage, and performance guarantees. Importantly, when service experiences increased demand and scale-up, other services no longer heavily utilized are automatically downscaled. This dynamic resource allocation process prevents unused services from occupying unnecessary resources and allows for efficient garbage collection.

Resources and Commodities

Services are run locally on a node as a sandboxed process. Since the core of Fleek Network has a small footprint, services have access to most of the hardware resources on a node. The hardware resources consumed from a service execution are packaged and measured as commodities. Resources can be but are not limited to things like Bandwidth, CPU, and GPU, and their respective commodities (GB/s, cpu cycles, etc.). Delivery Acknowledgements include the commodities consumed by a node while executing a service. This information is used to reward the node, based on the current pricing of those commodities by the network.

Service Examples: Edge Compute

For illustration purposes, edge compute is used as a category to showcase the different types of services that could be built on Fleek Network. Edge compute comes in many flavors, and Fleek Network can support mostly all of them. For example:

1. Cheap compute for simple JavaScript functions. Build/use a Deno or Lambda type service that any edge node could run the function and return the response.

2. Server-side rendering. Build/use an SSR service built on a container engine or serverless service like the JavaScript functions.

3. Consensus-based computation Build/use a service that sends the request to *N edge* nodes and comes to its own consensus on the result (which in many cases would achieve similar computational integrity but be way less expensive than needing to prove the computation).

4. Deterministic computation. Build/use a service that deterministically returns the response, and slashes a node for lying.

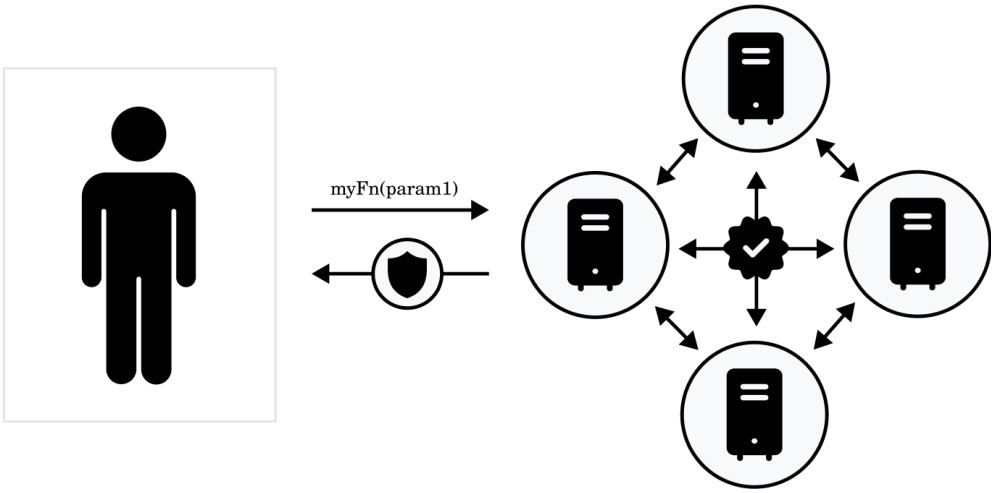
5. Support multiple options that can be specified on request. Build/use a service where the SDK would report statistics to the blockchain based on the work performed, and that data would then feed into the reputation engine to improve service routing.

6. Zero-knowledge computation Build/use any zkVM as a service, and then offload certain zkVM computation to the edge.

7. EVM compatible computation Build/use the EVM as a service, and then offload certain EVM computation to the edge.

8. Grid Compute. Build/use a service that breaks up and parallelizes compute across multiple nodes in the same geographical proximity.

VERIFIED COMPUTE SERVICE EXAMPLE



In contrast to a CDN type service where storage (memory + disk space) and bandwidth are key resources, here it would be the CPU and memory that gets utilized more, but storage and bandwidth would still get charged for depending on the logic the JavaScript function executed. For this example service the network would also need some sort of interface descriptor language so that the RPC function parameters could be composed by the client. There would also need to be admin RPC functions for this feature that lets a client deploy or destroy a function.

Building on Fleek Network: Who, What & Why

Fleek Network should help Web3 (protocols, middleware, services, apps, etc.) to be as performant as centralized solutions without needing to complicate (building it themselves) or compromise (using centralized infra in their stack). Even for all software developers and services more broadly (not just web3), Fleek Network should become a viable edge platform to performantly run different pieces of the modern stack on, or to optimize existing infra/services/data living elsewhere (ex. other web3 protocols).

The cost and performance of Fleek Network should be in-line with existing centralized cloud/edge platforms, but with the added Web3 benefits (decentralized, permissionless, censorship resistant, verifiable, cryptographically/economically secured, etc.).

Ideas for Building/Using Services

There are a wide range of edge services that could potentially be built on Fleek Network. Below are some examples of both traditional web/edge services that could be built and how those services could be utilized by web3 (and all web) developers, as well as some web3 specific ideas/use cases that developers could potentially build as services on Fleek Network.

Web/Edge Services:

- Hosting
- CDN
- Edge compute/Serverless
- SSR/ISR (Incremental Static Regeneration)
- Image Optimization
- Load balancing
- Grid computing
- Naming Services
- Databases (Document, KV, CRDT)
- Container orchestration
- Queues/Events
- Step Functions
- Analytics

Web 3 Specific Use Cases for Services:

Decentralized CDN

A decentralized CDN is a big missing piece in the web3 infra stack. Every web3 protocol, middleware, service, and app needs and/or could benefit from content acceleration. While today most projects use Cloudflare in front of their stacks to optimize performance/latency, once Fleek Network launches, web3 projects could use the decentralized CDN service as a drop-in replacement to gain the web3 benefits without sacrificing on performance or cost. The CDN service caches content based on usage, at the nodes that make the most sense for geo-based distribution of files (like a traditional CDN). It keeps track of requests serviced and charges the client and rewards the nodes based on the bandwidth served. The reputation system will track good CDN service and the routing will be determined based on that.

Decentralized Site/App Hosting

While the backends in web3 apps are more decentralized, the frontends still remain quite centralized. Fleek Network provides a great opportunity to take a big step forward on this issue/topic. For example, a service that could be built is something that leverages the blockstore and IPFS content addressing to keep track of applications that are deployed with additional metadata. It would essentially be the storage layer for frontends similar to how platforms like S3 or Netlify work. Hosting can further be extended by using the CDN and a compute service to add server-side rendering capabilities. The storage or processing for the hosting or SSR respectively could charge the client for the work performed and distribute rewards to the node operators.

Decentralized IPFS Pinning

Given the way Fleek Network works, all files/content on the network are content addressed (given a CID), and a mapping of the CID to origin(s) is stored in perpetuity. Combining that with the built-in (externally powered) file system that allows direct access to decentralized storage protocols (Filecoin, Arweave, etc.), as well as the CDN service , an IPFS pinning service could be built that provides the exact same service/experience as centralized IPFS pinning today, but using only decentralized infrastructure. Not only that, but this version of IPFS pinning would be just as performant, cheaper (storing on web3 storage protocols is typically cheaper than cloud platforms), and with much better data security/availability guarantees. Even if the file ever fell off IPFS, Fleek Network would be able to retrieve it (so long as it’s still valid on at least 1 origin) since the network stores the IPFS CID to origin(s) mapping in perpetuity.

Decentralized Edge Compute (on top of Web3 Protocols)

Compute as a service was discussed in more detail in the Services section prior. Most projects building apps on top of Ethereum or other protocols have additional logic such as account and team management/metadata or computing on private data, or integration to 3rd party APIs. Currently this logic is run on centralized platforms like AWS (lambdas) or Google Cloud. Running them on a Fleek Network edge compute service would mean the storage and execution of these processes are decentralized, permissionless, and trust-minimized, without

sacrificing performance, and most likely saving money on cost (compared to. using centralized platform edge compute).

Blockchain Snapshot as a Service

Syncing a full node for other chains is a cpu intensive process and can take hours, days or weeks depending on the chain. Fleek Network's internal blockchain uses content addressability to store snapshots of the chain head and uses the CDN to accelerate the entire state to the node to allow fast sync. A service could be built that does the exact same thing, but for other chains. The service could automate storing the snapshot to always stay up to date to HEAD and deliver the entire state to a node that requests it, drastically reducing sync time.

zkVM's, EVM, and other VM's as Services

A service that deploys a VM like one of the many zkVMs or the EVM is also possible. A service could provide compute in the zkVM and provide the zkSnark from the node that proves the correctness of the response. Since the zkVM would be running on geo-distributed, intelligently routed edge nodes, the network could help ensure the zkVM computation is happening in close geographic proximity to the client to further optimize performance/latency.

Alternative Rollup Sequencer

In the era of L2s, most use a centralized sequencer to post transactions to the L1s settlement contracts. These L2 networks cope with this layer of centralization by providing an alternate route around the sequencer by posting the transactions manually to the settlement contracts. The problem is, there is a reduction in block speeds users are accustomed to on an L2 and are then stuck with the L1s finalization times. In Fleek Network a developer could provide a service that offers a decentralized alternative to an L2s sequencers that batches and posts them to the L1s settlement contract. This could then achieve L2 settlement times while still offering a decentralized path. Another benefit of this service is it could enable the end user to not need the L2 specific gas token to submit these transactions.

Ephemeral Rollups

If you wanted a short lived rollup for something like an NFT mint event or a game/event, you could use Fleek Network to build/utilize a service that allows you to spawn ephemeral rollups that users could interact with for a certain amount of time (ex. mint window or game/event duration), and after that time elapses, the service could rollup the results to your smart contract. This could help users avoid gas wars/high fees while providing instant finality during the duration of the event. And since the rollups would be running on the decentralized edge, they would be fault resistant and highly performant.

Edge Proof Generation

With the rise of SNARKs/STARKs and the growing demand for performant and cost efficient proof generation, there could be compelling advantages to generating these proofs on the edge (closer to end users) in a decentralized way. As an example, a hypothetical Groth16 service can read the setup parameters as files (using the file system) and generate a proof based on user

specified public parameters. Support for other proving technologies can be built/utilized as separate services.

Verified Randomness

Fleek network uses an internal verified randomness beacon throughout its processes. Exposing this beacon through a service could be valuable to a lot of web3 applications. A service that does this could also support posting the verified randomness to other blockchains to achieve on-chain randomness where not previously possible.

Web3 Queries/Events

Services can easily be built to provide utilities for other blockchains besides Fleek Network. For example a service could be built that indexes events emitted from another chain and allows requests to query this data. This service could similarly just provide a common interface for these other chains to provide queries or submit transactions. Taking this a step further, a service could offer a level of atomicity across chains by submitting a few multi-chain requests depending on the previous requests being successful (i.e., bridge as a service).

Acknowledgements

This paper is a culmination of work and research done by multiple individuals on the Fleek team, as well as helpful comments/reviews from external ecosystem members. However special recognition is deserved for Alireza Ghadimi and Dalton Coder who led the design and development of the new decentralized edge protocol and wrote most of the technical details of the paper along with help from Janison Sivarajah, Matthias Wright, Ossian Mapes, Miguel Guarniz, Muhammad Arslan. Harrison Hines focused on the overall narrative and structure of the paper. Royce Moroch created all the illustrations for the paper. Nicolas Poggi handled editing and finalizing the paper. Helder Oliveira, Vojtech Studenka, Ignacio Rivera and several other members provided insightful comments, clarifications, and support. We thank everyone both named and unnamed who contributed.

References

[1] IPFS - Interplanetary File System (https://ipfs.io/)
[2] Filecoin Network (https://filecoin.io/)
[3] IPLD (https://ipld.io/)
[4] Narwhal (https://arxiv.org/pdf/2105.11827.pdf)
[5] Bullshark (https://arxiv.org/pdf/2209.05633.pdf)
[6] Fleek Network - Lightning Github Repository (https://github.com/fleek-network/lightning)
[7] Arweave (https://www.arweave.org/)
[8] Fleek Network's CDN Whitepaper (https://fleek.network/fleek-network.pdf)
[9] Mysten Labs (https://github.com/MystenLabs/)
[10] Blake3 (https://github.com/BLAKE3-team/BLAKE3)
[11] EigenLayer (https://docs.eigenlayer.xyz/overview/whitepaper)
[12] Ethereum paper (https://ethereum.org/en/whitepaper/)
[13] Snarks (https://eprint.iacr.org/2011/443)
[14] Next.js (https://nextjs.org/)
[15] ZKVMs (https://github.com/0xpolygonhermez / https://github.com/matter-labs/zksync-era)
[16] EVMs (https://ethereum.org/en/developers/docs/evm/)
[17] WASM (https://github.com/WebAssembly)
[18] Cuckoo filter
(https://docs.fleek.network/blog/bloom-and-cuckoo-filters-for-cache-summarization)
[19] Jamstack (https://jamstack.org/)
[20] Google bounce rate statistics
(https://thinkwithgoogle.com/marketing-strategies/app-and-mobile/page-load-time-statistics)
[21] EigenTrust Algorithm (https://nlp.stanford.edu/pubs/eigentrust.pdf)

Appendix A: Performance Reputation Algorithm

Algorithm 1: Reputation Score Calculation

/* For node i, the reported measurements are $X^{(i)} = [\{x_{1,1}^{(i)}, x_{1,2}^{(i)}, ..., x_{1,n}^{(i)}\}, \{x_{2,1}^{(i)}, x_{2,2}^{(i)}, ..., x_{2,n}^{(i)}\}, ..., \{x_{m,1}^{(i)}, x_{m,2}^{(i)}, ..., x_{m,n}^{(i)}\}]$, where $x_{j,k}^{(i)}$ denotes measurement j (for example latency), that was reported by node k. */

Input: X := Reported measurements for each node
Input: S := Current reputation scores for each node
Input: β := Parameter for exponentially weighted moving average
Output: Y := New reputation scores for each node

/* We perform outlier removal on the different measurements (latency, bandwidth, etc) using Z-Score normalization. */

$\hat{X} = \text{zScoreNormalization}(X)$

/* We calculate weights for each node based on the current reputation score. The weight for node k is denoted by w_k. */

$W = \text{calculateWeights}(Y)$

/* For each node, we compute the weighted average for each measurement, weighted by the reputation score of the reporting node. */

for *each node* i **do**
  for *each measurement* j **do**
    $\mu_j^{(i)} = \sum_k x_{j,k}^{(i)} \cdot w_k$

/* For each $\mu_j^{(i)}$, we compute the mininum and maximum value across all nodes. */

for *each measurement* j **do**
  $\min_j = \min_i \mu_j^{(i)}$
  $\max_j = \max_i \mu_j^{(i)}$

/* Each $\mu_j^{(i)}$ is normalized using min-max normalization. This will ensure that $\mu_j^{(i)} \in [0, 1]$. */

for *each node* i **do**
  for *each measurement* j **do**
    $\hat{\mu}_j^{(i)} = \text{minMaxNormalization}(\mu_j^{(i)}, \min_j, \max_j)$

/* For each node, we compute the average over the normalized measurements. The new reputation score is then given by the exponentially weighted moving average over the scores of the previous epochs, where $s^{(i)}$ denotes the current reputation score of node i. */

for *each node* i **do**
  $y_{next}^{(i)} = \frac{1}{n} \cdot \sum_j \hat{\mu}_j^{(i)}$
  $y^{(i)} = (1 - \beta) \cdot s^{(i)} + \beta \cdot y_{next}^{(i)}$