Picolo: Fast p2p open database network[☆]

Adi Kancherla, Arunesh Mishra
Picolo Labs*
San Francisco, California

Summary

Picolo is a fast, scalable, fully decentralized, globally distributed transactional database network for blockchain based applications. It is a p2p network with an open participation model where any node can freely join and leave at will. Nodes discover each other in a decentralized manner via DHTs where each node maintains just enough routing information that will enable it to discover the topology of the whole network. It uses a probabilistic replication framework on top of DHTs to achieve an O(1) lookup latency in the average case. It also adds a layer on top of DHTs that allows locating nodes by geographic regions.

All nodes in the network are symmetric - they all run the same database software that provides a number of desirable features. Clients can interact with them via a generic SQL interface that has familiar methods such as selections, aggregations and projections. Nodes typically are part of a replica group (of a small size like 5) and the network is made up of many such groups. Nodes in a replica group are responsible for the same data where mutations to it are synchronously replicated to all replicas. Mutations are applied in the order seen by nodes where order is determined by timestamps that are a combination of physical and logical clocks. This gives Picolo the property of external consistency - the strongest consistency property a database can achieve. Distributed transactions across nodes are also made possible by using these hybrid timestamps. Automatic data sharding is supported when it grows too big or for load balancing purposes. Queries posed to the network can be run across all nodes with vastly differing schemas. Overtime, semantically similar schemas are logically grouped together for faster processing and richer results. There is a notion of user-controlled and app-controlled schemas that aids in achieving data sovereignty. Fine grained access control rules to data can be defined via a declarative language which are then enforced via encryption.

There are four types of participants in the network: storage providers, storage consumers, data providers and data consumers. Storage providers join the network to offer their spare computing resources like disk space, cpu and memory in exchange for cryptographic tokens. Consumers can pay providers with these tokens (among other ways) and use the resources provided to satisfy their needs. Since the network is open for anyone to participate, consumers need some safety guarantees before they can reliably store their data on untrusted nodes. So providers are required to put up security deposits and the network is designed to be tolerant to their byzantine behavior, to punish malicious nodes by slashing their deposits and to reward honest nodes for diagnosing and reporting byzantine behavior. A mechanism called Proof-of-Query is used for the diagnosis and deposit slashing is enforced by a smart contract system.

Keywords: decentralization, blockchain, ethereum, database, sql, access control, secret sharing, p2p, tokens, incentives, mechanism design, byzantine faults, data sharing

[☆]Version 1.3
*https://picolo.network

Contents

1. Introduction

We are in the midst of building a new Internet. Blockchain networks like Ethereum have popularized the idea of unstoppable, owner less applications that are run on an open network of untrusted but incentivized nodes without the oversight of a central authority. An increasing number of these decentralized applications (Đapps) are being created everyday with evolving data storage needs. The first generation of these dapps stored their data entirely on a blockchain itself while the current ones are storing it on decentralized file storage systems like IPFS with just hashes of the data stored on the blockchain. Although this method works for dapps with simple storage needs like the need for storing data as a blob, it is very limiting for dapps with more complex requirements such as the need to store data at a more fine grained level and efficiently querying it. A popular option is then to use a cloud hosted database (ex: Google Cloud SQL) but that turns a dapp into a non-dapp by introducing centrality into the system.

In this paper we introduce Picolo, an open database network that combines the benefits of a traditional database and p2p software. It is the world's first NewSQL *decentralized* database system that offers features such as a SQL interface, distributed transactions, external consistency [1], lock-free reads and snapshot reads. It is an open network that allows anyone with spare computing power and disk space to join the network and get rewarded for hosting and serving structured data.

Features that set Picolo apart from other decentralized databases:

- Transactions can be applied across rows, columns and tables across nodes
- Client controlled replication and data placement
- Support for storage of typed data
- Support for semi-relational structure for tables
- Configurable backups and restore mechanisms
- Allows for verifiable transaction logs
- Data poisoning detection
- Distributed query processing
- Fine grained access control to data

Rest of the paper is structured as follows: in section 2, we discuss some related work. Section 3 presents the overall design of the system, section 4 presents the network subsystem, section 5 presents the database subsystem and section 6 presents our approach to dealing with problems that arise in a network made up of untrusted nodes and discusses an incentive mechanism to make them behave as per system's needs. In section 7, we propose a message exchange protocol in the same vein as http but for relational data sharing. Section 8 concludes the paper.

2. Related Work

There have been attempts in the academia at building p2p data management systems (PDMS). PeerDB [2] pioneered a full-fledged data management system that supports fine-grained content-based searching on a distributed network of nodes with heterogeneous schemas. It proposed a "code goes to data" paradigm where mobile agents are employed to perform query processing at a peer node in order to reduce network bandwidth. PIER [3] provides a relational data model and query operators on top of any distributed storage system. It embraces the notion of *data independence* and only concerns itself with query processing. It does not provide any sort of replication or ACID guarantees of a database and does not maintain any indexes of its own. Piazza [4] describes a system where peers have pairwise schema mappings between heterogeneous schemas and how

they are transitively extended to answer queries from peers separated by more than one edge. Queries are reformulated at each peer to match the local schema before execution. While this approach works well for a few reformulations, a large number of them may result in information loss or in returning of irrelevant results.

In the blockchain space, there are a few projects tackling the decentralized storage problem. Filecoin [5] adds an incentive layer on top of widely used IPFS [6] - a p2p file sharing network. It pays miners for contributing disk space and network bandwidth to the filecoin network. Storj [7] and Sia [8] are two similar projects that offer decentralized file storage networks for end users wanting to store files on more resilient, secure and censorship resistant networks. While these networks are great for storing large unstructured files like images, videos, documents etc, they are not suitable for storing structured data and do not support complex queries beyond simple keyword search.

BigchainDB [9] offers blockchain like characteristics on top of a MongoDB engine lending itself to being queried like any other NoSql database. Its data model consisting of assets, transactions and outputs makes it easy to assign, track and transfer ownership of data. However, it doesn't have an open participation model where nodes can freely join and leave the network. Membership is controlled and resembles a typical database cluster maintained by devops teams at companies except in this case the teams can be inter-company. Bluzelle [10] has an open participation model although it only supports key value lookup semantics, has no notion of data sharing and access control, does not support distributed query processing, data availability proofs or verifiability of mutations and has no access logs. Mediachain [11] created a decentralized data network for tracking ownership of creative arts like music. It has features like automatic schema translation that enables searching across sources with different schemas. It also supports a query language that allows for complex querying within a namespace. Our vision is aligned with Mediachain's vision, however, the project is not in active development.

3. Overall Design

Picolo is a homogeneous network of nodes running a p2p database software. The network has an open participation model - any node that runs Picolo's software can freely join the network. It may elect to become a storage provider (**§3.2**) by depositing PINTs (Picolo Network Tokens) and become eligible for earning rewards or it can take on the role of a data provider or a consumer. Storage providers power the network and are organized into clusters. Each cluster serves a storage consumer, typically a dapp and consists of shards. Each shard is made up of small pieces of data sets defined by a key range. Each such key range is replicated for durability and availability where consensus among replicas is achieved by running a paxos based algorithm. Clusters are horizontally scalable by adding more nodes and splitting data into new shards. Storage consumers can directly connect to clusters by-passing the need for DHT based lookup, improving latencies.

Nodes are connected to each other via a DHT overlay network and a prefix-based routing algorithm that locates content using a content-based addressing mechanism. The overlay network includes methods for discovery of content/nodes, fast and opportunistic routing and handle node dynamics such as failures and node additions. The routing framework includes optimizations which allow nodes that belong to a cluster to cache localized route information. Cached links allow for optimizations that provide an O(1) lookup latency for most queries. Cached content and links use what we call soft-state publishing to allow for faster convergence in the face of node churn. A ladder based trust and governance model which is built using a combination of earned trust and stake, ensures proper decentralized operation of the network in the face of certain types of malicious intent. This design is discussed in detail in **§4**.

3.1. Incentive compatibility

Picolo is an incentive compatible network. All participants (**§3.2**) in the network are assumed to be rational and acting in their best interest and may not necessarily be interested in the overall benefit of the network. Hence we employ a mechanism design that incentivizes participants to work towards the proper functioning of the network while keeping individual interests in mind. For instance, we enhanced the normal pings that nodes in a database cluster send to each other for health checking with requests for small amounts of

randomly sampled data a node is responsible for. If a node fails this check, its security deposit is slashed. Hence, storage consumers can have reasonable confident that their data is being reliably stored. At the same time, storage providers are paid for storing and serving data correctly, hence it is also in their best interest to not corrupt or delete consumer's data.

3.2. Participants

There are four types of participants in the network:

1. Storage providers
2. Storage consumers
3. Data providers
4. Data consumers

Storage providers: Storage providers provide compute resources that power the network such as CPU, memory and disk. In our current architecture, these resources are on the same machine. While its possible to separate the "server" part from "storage" part as in databases with NAS (network attached storage), in a p2p world with commodity network links, such a design introduces unacceptable latencies. Storage providers are compensated for powering the network by storage consumers.

Storage consumers: Đapp developers that need to store structured data fall into this category. They can simply use the database functionalities provided by the network without the need to perform any database administration tasks. In this sense, the network gives them the same ease of use as a cloud hosted database provider but at a much cheaper rate.

Data providers: These are people who want to share their data with the world. This could be highly domain specific data that doesn't lend itself well for sharing using existing tools. As an example, one might write a program that reads the state of a blockchain like Ethereum and store it in a query-able format on Picolo. They can then sell access to this data. More specifically, one such program might entail observing blockchain state to draw insights on how dapps of a particular category are being used. Storage consumers above can also be data providers; for example an end user using a dapp that stores data on Picolo and puts data control in the hands of the user enables them to become a data provider for other dapps.

Data consumers: These are people/applications/nodes interested in data shared by data providers. Future of application building does not include hoarding user data in silos, instead innovative applications will be built using data commonly available or shared at a single location. This is the idea behind the paradigm of "code comes to data" where users solely control their data and allow applications to provide services by selectively giving access to their data.

4. Network Subsystem

The network layer is a fully decentralized, p2p overlay-based routing layer among the nodes participating in the Picolo network. The most important goal of the network layer is to help locate content as efficiently and quickly as possible while surviving certain kinds of failure or malicious intent. It focuses on delivering messages which have application semantics such as database queries, read/write requests or other management functions to the respective nodes which can satisfy them. In the rest of this section, we use the word object or content to loosely refer to content such as tables, shards, metadata and such, i.e., any application specific data that is content-addressable and needs to be part of the lookup layer. An item is content addressable if its *address* or *key* or the primary *identifier* is purely a function of its content and does not include parameters related to the location and method of storage. Our p2p overlay routing infrastructure offers efficient, scalable, location-independent routing of messages directly to nearby copies of an object or service using only localized resources.

Location independent routing: Location independent routing refers to a class of techniques for locating

objects based on content rather than their location, while attempting to find the shortest path possible to reach such objects. This property is important in our design for Picolo and any decentralized system since it helps reduce the impact of failures, node departures and Byzantine node behavior (including collusion) since a node does not have explicit *control* over the content it can host or be responsible for. This control gets delegated to the content routing layer, where it becomes possible to provide some level of hosting/location control to the application (dapp) developer, user or other policy-based decision processes (such as privacy policies enforced by a government or organization). P2p networking literature refers to this functionality as Decentralized Object Location and Routing (DOLR) [12]. Picolo's design offers the following properties which are required of any high performance p2p overlay-based lookup layer:

- *Determinstic node mapping*: Picolo is able to locate objects anywhere in the network. That is, there should be no object in the network that cannot be accessed using the lookup layer. Also, the mapping of objects to their location in the network should be the same regardless of where the lookup originates.
- *Low routing inefficiency*: Routes should have low *stretch*. Stretch is the ratio between (network level) distance traveled by a query to an object and the minimal network distance from the query to the object. Optimal solution would be to always send the query to the nearest copy possible.
- *Balanced load*: The load should be evenly distributed across the nodes in the network, thus, reducing hotspots.
- *Dynamic membership*: The system allows arrival and departure of nodes while maintaining functionality. This is important for handling failures and Byzantine behavior of nodes.

In **§??**, we summarize a wide-range of research on p2p overlays and storage systems over the last two decades that has influenced our design and thought process. In **§4.1** we present the overall design of the network layer including the Picolo namespace (**§4.1.1**). **§4.2** presents mechanisms to handle node arrivals and departures into the Picolo overlay network. We discuss route optimizations using caching and replication in **§4.3**.

The Picolo overlay layer can be implemented on top of any datagram network protocol such as UDP or IP for transport. While we use IP or IPv6 for the layer-3 connectivity on the Internet, it is possible to use a suite of protocols for the transport layer depending on the localized network conditions. In **§4.4**, we discuss the details of the transport mechanism used in Picolo for node-to-node communication. Finally, we discuss how responsibility is delegated to nodes and how trust and governance happens in **§4.5**. For a detailed presentation of the prior research in the area of p2p networks, please read Appendix **§B**.

4.1. Design of the Picolo overlay network

The Picolo overlay network consists of a p2p DHT based lookup for mapping keys to objects, content or services. This mapping resides inside a namespace, thus functionality gets replicated across namespaces allowing a wide-range of applications to co-exist. The APIs provided by the network layer are influenced by research from the p2p networking community and discussed in **§4.1.2**. There is a caching layer that allows for frequently used items to be propagated closer to the demand endpoints in the network including ability to replicate as needed. A Picolo *network node* is a node that participates in the Picolo network overlay functionality such as primary and secondary routing, functions that allow topology maintenance and related mechanisms. It is possible for a Picolo node to particpate in the network functionality, the database functionality or both depending on its resources (such as disk space, internet connectivity).

4.1.1. Picolo namespace

Both nodes and content have IDs that map to a single 256-bit space using a hash function. This technique is fundamental to making the routing layer content-addressable. The addressing is hexadecimal based, which is the radix used for the prefix calculation and lookup in the routing table. This design is similar to how addressing and routing works in IPv6. The node ID is a hash of its public-key certificate using a SHA-256 hash function. The content ID is a SHA-256 hash of the application type, database name and version and the table name (and possibly a shard number if needed). As long as the same consistent method is used

to create content IDs, the actual format of the input string doesn't affect functionality at the network layer (and can thus be determined by the application being used).

This space can be segregated using a namespace identifier, which allows multiple such "naming layers" to co-exist. For example, one way to assign naming layers could be based on a per-application type. The content and node IDs have to be unique only within the namespace. The default namespace, which is global, is the keyword "default". Each namespace has a separate datastructure and functionality thus, creating an isolated "virtual overlay" on a per-namespace basis. For the rest of this section, the discussion gets isolated to within a single naming layer, such as the "default" namespace.

4.1.2. Core API

Picolo's network layer supports an API similar to a standard p2p overlay network, for a detailed discussion refer [12]. The primary goal of the API is to publish and locate objects, content or service identifiers within a given namespace. All operations that occur in a namespace can be replicated across namespaces as needed. Currenlty, we support the following API methods in a decentralized manner:

- Publish(namespace, content key, object, credentials): Makes the content available on the network, including first a copy on the local node. The object could be a database table or a shard. Operations on the object fall within the context of the database layer and are discussed in a later section.
- Unpublish(namespace, content key, credentials): Removes the content from the network making it inaccessible.
- Lookup(namespace, content key, credentials): Lookup item with the given content key using the p2p lookup protocol discussed in this section. The API returns a node identifier which can be used in the next method to execute a remote funationality on it.
- RemoteCall(namespace, node ID or content key, credentials, method details): This executes a given functionality on a remote node. If the key is given, it first does a node lookup using the Lookup() method.

These methods can be called on a single node. They can also be called using a client SDK which shall be provided. The client SDK would use the DHT layer to first find "any" node, connect to it and call the above methods. These methods might use other algorithmic constructs which define core aspects of the routing layer and shall be discussed further in the following subsections.

4.1.3. Routing and Lookup

Picolo uses a prefix-based routing mechanism to find the node for a given content ID. The content ID can be computed by using a combination of the database name, table name and shard ID (if sharding exists, or version information). Optional values such as read/write shards or specific functionality flavors can also be used to "color" the content key (i.e. create variants or replicas etc).

Routing to a destination ID: Given a destination ID, each Picolo node that participates in the routing functionality constructs a routing map, which is similar to routing tables used in layer-3 systems today. Using these routing maps, routing of a message happens *locally* and *incrementally* at each node in the overlay path to the destination. The core routing method is to route using a prefix-match. This is similar to the Classless InterDomain Routing (CIDR) routing architecture for the Internet [13]. The routing works as follows. We resolve digits from the most significant to the least (left to right). A node N has a routing map with multiple levels. The number of levels are equal to the number of digits in the ID space. For a 256 bit ID with a hexadecimal base, we have 64 digits. At level i, we have nodes that match i digits with the destination ID for $i \in [\ 0..63\ ]$. At any given level, the number of entries are equal to the base of the ID space, that is, we have 16 entries, one for each digit [0, 1, 2..F]. At the i th level, the j th entry is the node with a prefix of prefix(ID, i) + j, where prefix(A,j) represents the prefix of A of length j. As an example, Table 4.1.3 shows the different levels for the node with ID "691D". Suppose we wish to route the ID "692B" given the routing table and we are doing a lookup at level 2. This means we have a prefix match of "69" and are searching for a node in the column for the digit "2". This happens to be node C_2 and thus the query is then forwarded

there. Similarly, suppose we are asked to route to node "615B". This would be at level 1 since we have a match for the prefix "6". We would thus route to node B_1 matching the column"1" and row "6".

Prefix	0	1	2	..	F
None	A_0	A_1	A_2	..	A_F
6	B_0	B_1	B_2	..	B_F
69	C_0	C_1	C_2	..	C_F
691	D_0	D_1	D_2	..	D_F

Table 1: Example of a routing map for node with ID "691D". Suppose we were to route to ID "601D", we would forward this to node B_0. Similarly, to route to ID "6911", we would forward to node D_1.

Home Node Set: The content ID for a given content is computed using a cryptographic function such as SHA-256. The network design allows the use of any cryptographic hash function with an even distribution of hash values. The content ID consists of this hash value, a *replica count* **r** and possibly any other parameters bundled as a JSON blob. As discussed in the previous section, both node IDs and content IDs are confined to the same 256-bit space. Every content ID maps to one or more *home nodes* that form a *home node set*. A home node is the node that is responsible for either directly hosting the content if it has that capability or has a link to another node that hosts content. Essentially from the routing layer's perspetive, the routing function ends once we reach a home node. If the replica count $r > 1$, then the content maps to a *set* of home nodes where each home node is independent from the rest. The function HomeNodes(ContentID) returns a set of home nodes. An implementation of this function using SHA-256 is as follows:

$$X_{id} = \{X_{256}, r\} \tag{1}$$
$$H_X = HomeNodes(X_{id}) \tag{2}$$
$$= \{y_i : y_i = \text{SHA-256}(X_{256}, j) \quad \forall \quad j = 1..r\} \tag{3}$$

Mapping a content to multiple home nodes is purely for fault tolerance purposes and can be configured by the database schema (application designer) as needed. For simplicity of the rest of this section, we shall assume that there is a single home node, i.e., $r = 1$ or $|H_X| = 1$ as all the algorithms and concepts easily extend to the case with multiple home nodes. In any case, the size of H_X is expected to be small for any given X.

In Picolo, nodes can participate in different kinds of functionality such as hosting or routing or both. This segregation allows certain nodes to either participate in the network routing functionality (and handle that load) or provide the database functionality or both depending on various factors (such as resource availability). In the case that a Picolo home node stores a link to another node which hosts the content, such a link is called a *content link*. This link is not part of the routing layer since the routing functionality ends with finding a home node. When a Picolo node stores a link to another node as a part of the routing map such a link is called a *node link*. Thus, the links stored in a routing map as discussed above are all node links.

Property 1 (Unique Home node set)**.** The home node set H_X for a content ID X must be unique, that is, H_X must generated the exact same set regardless of where it is computed in the network.

Note that the example implementation of HomeNodes(ContentID) does satisfy this property. This property is important to ensure that our decentralized overlay-based routing algorithm as discussed in this section will work in a deterministic manner regardless of the starting point in the network.

The goal of the routing protocol is to find a route with low stretch (ideally a stretch of 1.0). Stretch is the ratio between (network level) distance traveled from (say) node A to node B to the minimal network distance between node A and B. Since we are routing at the overlay layer, a route on the p2p overlay might result in a suboptimal route at the network level. Figure 1 illustrates the concept of network stretch and why a low stretch is important. In this example, we are routing from node A to F. The p2p overlay route

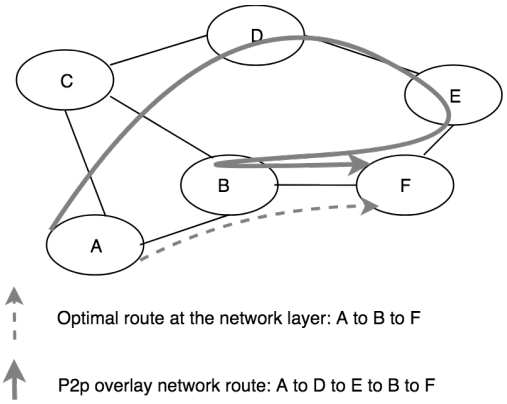

Figure 1: Example of why a low network stretch is important fo any p2p routing overlay protocol. The solid lines show the layer-3 network topology. The solid grey arrow shows the route taken by a p2p overlay routing protocol to route from A to F (which is A, D, E, B, F).

goes from node A to D, then to E, B and finally to F. This results in an inefficiency as seen at the layer-3 since messages travel the same links multiple times. This inefficiency is captured by the "stretch" metric as a ratio of the p2p overlay route to the optimal layer-3 route (which is A to B to F in this example). Since this is at the routing layer which most network messages would take, even small improvements would result in significant practical performance gains seen at the application level.

Secondary routing layer for low-stretch: Network nodes in Picolo have two types of routing layers. The first is called a *principal* route layer which defines the core neighbor links that connect the node to its neighbors. The Principal or primary route layer uses the route map (as discussed above) that uses a prefix-based routing algorithm. The goal of the principal routing layer is to maintain a highly reliable foundational network topology. Issues with these links might cause the node to be marked as having serious failures (and thus trigger failure mode operation semantics). The second set of links are direct links to specific nodes and are maintained as a hash table of <Node ID, IP Address> mappings. These are used for optimizations, such as to store direct content links to popular items or to cache links to popular home nodes, etc. We call these the *cache links* or *secondary links*. We shall discuss the distributed caching algorithm separately in **§4.3** which will provide probabilistic guarantees of a lookup cost of $O(1)$ for most queries thus reducing the routing stretch.

Explicit soft-state publishing: Nodes in Picolo send periodic *heartbeat* messages to other nodes through the p2p overlay network as given by their routing tables. This heatbeat message includes any content links or content IDs that the nodes are hosting. These messages are routed along the *principal* routing links which every node maintains for two reasons: Firstly, these messages are not latency critical and thus don't need to use the secondary layer. They serve to validate and refresh the network metrics of the primary links. They help discover "holes" or link/node failures. Secondly, these heartbeat messages also help deposit content links at various intermediate nodes that help route them. The content links and latency metrics expire according to a timeout and are refreshed with such messages. This operation of advertising content with an automatic expiry is called *soft-state publishing*. Picolo also supports targeted re-publishing of content such as when a node comes back online after a disruption or when a node leaves/joins the network.

Query routing: Lookup or querying for a specific content ID (or a set) happens in the following manner. A client that wishes to make such a query connects to one of the network nodes in Picolo. The network node then computes HomeNodes(contentID) for the requested content. The lookup or query request is then forwarded to one (or more) of the home nodes from this set. In the process of this lookup, if at any stage, multiple node paths are encountered, then the query can be forwarded along all paths (to minimize latency but incur a slightly higher cost) or can be forwarded along the path with the best latency. These tradeoffs can be exposed at the HomeNodes(contentID) API call to allow the application developer to make such choices.

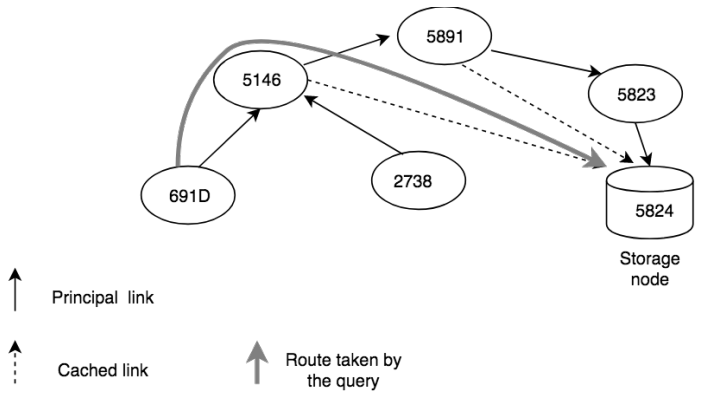

Figure 2: Example of routing a query for the ID "5824". The query originates at node "691D" in the above illustration. The solid black lines show the principal routing map. The dashed lines show the cached links. The think grey curved arrow shows the eventual path taken by the query.

Figure 2 shows an example of routing a query for a content ID "5824" originating at node with ID "691D". The dotted arrows show the cached content links from secondary or cache routing network. The primary or principal links are shown in solid arrow. In this example, node "691D" routes the query for "5824" to the node "5146" first as a level 0 match in its principal routing map. At this point, the first digit "5" is match and the node "5146" has a principal route to "5891" (matches "58") and then to "5823" (matches "582") which is the home node for the content ID "5824". "5823" then routes the query to the storage node which hosts the content. This would be the route that a query could take if it were to follow the principal routes in the routing map.

The secondary routes are also shown in the Figure as dotted lines. In the example shown, node "5146" has a cached link to the storage node for the content ID "5824" and thus directly routes the message to that node. Cached links can thus optimize the routing and reduce the network stretch by a significant factor. In practice, our route caching algorithm will ensure that most lookups and queries can be satisfied using the cache routing network, so that we can get an $O(1)$ lookup performance in the expected/average case.

Theorem 1. Picolo's network layer can perform location independent routing given Property 1.

The proof for this theorem follows from two facts. First, for a content ID X, the function $H_X =$ HomeNodes(X) returns the same set of nodes regardless of which network node this is executed on, thus, proving that the result is purely a function of the content ID X. Secondly, we note that the routing protcol requires home nodes (either directly hosting content or content links) to periodically send heartbeat messages through the principal links to the rest of the network. This ensures that any node in the network will have the routing layer capabilty to reach atleast one node in this set H_X, thus proving the "routability" of the home node set H_X. This proves the Theorem.

Theorem 2 (Fault Tolerance)**.** Let X be any content ID and let its set of home nodes be given by $H_X =$ HomeNodes(X). Also, let S_n be the set of all nodes (of cardinality n) which form the network that satisfy the above properties and Theorems. Now, if $|H_x| > 1$ and $\forall a, b \in S_n, P(a|b) = P(a)$ where $P(a) =$ probability of node $a \in H_x$, then H_x has fault tolerance with relabilty of $|H_x|$.

The proof of this Theorem can be seen in the following manner. First, the Theorem 1 states that Picolo can do location independent routing for any node $y \in S_n$ where S_n is the set of nodes in the network. Now, for a content ID X if the probability of any node $z \in H_X$ is uncorrelated with any other nodes (they are independent) then by Theorem 1 they can be independently routed by the network. Now, the fault tolerance property follows from the observation that the only remaining way faults (network or node failures) can be correlated is by attempting a hash-prediction (reverse hash attacks) on the cryptographic hash function used in the HomeNodes(X) function, which has been proven to be computationally hard for our choice of SHA-256.

4.1.4. Delegate Routing

There is one practical aspect of routing in a p2p overlay like Picolo that we have not addressed in order to simpify the discussion thus far. This has to do with whether home nodes will actually exist in a sparse network space. We discuss this issue and present a solution that is a drop-in replacement for the HomeNodes(X) function and preserves the above Theorems and properties.

Node and content IDs in Picolo map to a relatively large space, i.e. 2^{256} in size. Let H_X = HomeNodes(X) for content ID X. Given the large ID space and relatively sparse nature of the network nodes (a thousand or a million nodes out of 2^{256}, it is unlikely that nodes in H_X would have an exact match in the network. (this is expected with any content based DHTs routing layer [14, 15]). Nevertheless, we route to each node $y \in H_X$ as if it exists. During this process we might encounter empty neighbor links on the way. In such cases, the algorithm selects an alternate link but does it in a way that is deterministic across the whole network. Routing will terminate when we reach a node which is the best match and the lookup cannot be routed further. This node then, becomes the *Delegate* for the subject node y. We call this *Delegate Routing* where we route the query to each node $y \in H_X$ such that it gets mapped to a node that is a *delegate* for the home node with ID y.

In the routing table, an entry at a level l for an digit k will be empty only if no such nodes exist in the network that have a match. Thus, the same entry will be empty for all nodes in the network. As a result, regardless of where the lookup query originates for a content ID X, it will always compute to the same delegate node y. Delegate routing thus becomes a determinstic routing function computed by the network in a distributed manner. It is important to note that this delegate resolution is the same regardless of where the function computation originates in the network.

Now, we can replace the function HomeNodes(X) with a new function H_X = DelegateNodes(X) that automatically returns a set H_X such that each node in H_X exists in the network. All the analysis, properties and Theorems discussed above extend directly to the DelegateNodes(X) as a drop-in replacement for the HomeNodes(X) function.

4.2. Node Dynamics

We discuss how the network functions when new nodes join or existing nodes leave the network including unexpected node departures.

4.2.1. Node insertion

In this section, we discuss how a new node can join the network and create a routing map for both the principal layer and the secondary layer. The routing map has to be created so that the network properties and key Theorems hold.

Node insertion happens in the following manner. A new node N first creates its 256 bit ID, say, N_{id}, using any credentials it has. Credentials can include node identifier information, owernship certificate information and such. Next, N connects to the core nodes that are available through DNS mechanisms or other bootstrapping methods. It uses a special API call requesting that N be inserted into the network. This API call helps N become part of the network by building its principal routing map correctly and informing other nodes that it is available.

N computes its principal routing maps in the following manner. It uses the Delegate routing function to find the set of delegate nodes to route to the content ID N_{id}. However, at each hop we use the information from the nodes being contacted to construct the neighbor map.

New node notification: The final step is to inform the other nodes of N. This is done by going through the nodes at each level in the routing map that N has constructed and sending them an explicit heartbeat message. This includes any nodes that are delegates for N. This will ensure that new queries will start getting routed to N.

4.2.2. Failures and node departures

Nodes can depart the Picolo overlay network in two ways: First, is the case of a graceful departure, where the departing node send out an explicit notification through the network informing of the departure. This allows the network to adapt before the node leaves, by updating the routing tables and moving content out of the departing node if necessary.

Second is the case of a sudden departure such as a node getting shutdown, failing for various reasons or getting disconnected from the network due to Internet connectivity issues. A node with a degraded Internet connection can also be treated under this scenario. The basic mechanism to recover from such failures is using the heartbeat advertisements as discussed previously. Each link on the overlay network includes heartbeat messages which help update the latency metrics and also result in re-publishing or updating the content list hosted on a node. Upon missing a certain number of messages from a node suspected of failure, neighboring nodes and request an explicit update from such a node. Upon timeout, the neighboring nodes arrive at a consensus that the suspected node has failed and thus update their routing tables accordingly by initiating the departure API calls as for the first case.

4.3. Replication and caching

Structured p2p distributed hash tables can implement lookups in $O(logN)$ hops. Since each hop could add between 100-500 ms of latency, for a network of 10K to 100K nodes this computes to about 4-5 seconds for a lookup (or more). Thus, we require a caching and replication strategy to reduce this cost especially for popular items. The discussion in this section applies to any content or service thats is made available through the p2p system and thus extends beyond Picolo.

There has been significant prior work on caching and replicating strategies for p2p overlay networks. Methods such as Beehive [16] provide a closed-form replication algorithm based on a derived equation that guarantees $O(1)$ lookup but requires full knowledge of the network, such as number of nodes and popularity values for each data item. While this system might be good for analysis and benchmark purposes, its implementation is not pratical for Picolo. Kelips [17] is a probabilistic algorithm that also provides $O(1)$ lookup performance (in the expected case) by dividing the network into $O(\sqrt(N))$ affinity groups each of $\sqrt(N)$ size. They use a gossip protocol to replicate content to all nodes within an affinity group. Work by Gupta et al [18] explore the tradeoff between routing table size and lookup latency. They offer a guaranteed $O(1)$ lookup by maintaining routes to each and every node in the network. Farsite [19] provides a better ttradeoff by using routing tables of $O(dn^{1/3})$ size to route within $O(d)$ hops. Other p2p applications such as PAST [20] and CFS [21] use fixed size caches on intermediate nodes to cache the objects being queried. While they are unable to provide closed-form analytic bounds on query time, their average case performance is reasonably good.

We draw on the above literature to find the right balance between routing table size, cache and replication storage at nodes versus lookup latency. The strategy presented in this section achieves $O(1)$ lookup on average for networks with standard node and popularity dynamics. In other words, we allow for node joins, failures, unexpected departures (including malicious intent) along with changes in service or object popularity. We also allow for "flash crowds", that is, an item, object or service (such as a table-shard, or certain rows in a table) can quickly gain popularity (as given by standard Internet virality models [22]).

- A node caches or replicates an item with a probability that is proportional to the number of queries it is expected to get for that item in the upcoming time interval T.
- The difference between a cache and a replica is application specific. For the database application that the network layer hosts, it depends on whether a particular table or a shard is allowed to have write permissions on the replica.
- When a node caches or replicates an item, it affects the probabilitic demand distribution for the item at nodes that are on the routing path which would have gotten the request has this item not been cached.
- Thus, by repeating this algorithm iteratively, it would converge to the best caching pattern which would reduce lookup times with high probability for all items on the network.

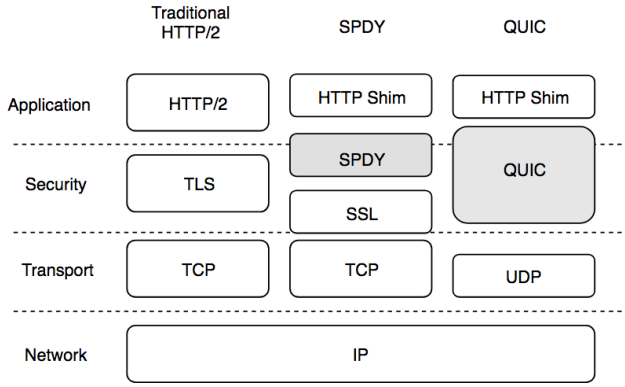

Figure 3: Comparison of network stacks for Traditional HTTP/2, QUIC and SPDY.

Each node maintains these cached links as a part of the secondary or "cache" routing layer which is essentially a lookup map of <Node ID, IP Address> pairs. It is possible for nodes to maintain relatively large caches since they are only limited by the memory and node dynamic characteristics of the network. Cached links that use the methodologies discussed above can significantly reduce lookup latencies. Also the downside of a stale cache link is quite low, since any node can always initiate a principal routing layer based lookup procedure.

4.4. Node Connectivity protocol

We describe the details of the transport protocol layer of a Picolo node. We call this the Node Connectivity Protocol layer. This protocol layer is responsible for connecting to a set of peers, maintaining network connectivity under possibly varying network conditions and allowing for a p2p network programing model (as opposed to client-server programming model which most transport protocols are based off). The protocol layer should also work with nodes that are behind Network Address Translators (NATs).

When a Picolo node starts for the first time (fresh install), it will query a set of "root" servers, similar to the DNS architecture of the Internet which is one of the largest decentralized lookup databases in the world [23]. These root servers populate the nodes with a list of neighboring nodes and content to bootstrap with. It uses algorithms outlined in the previous section to populate its routing table. As part of downloading the node binary, each download is given a certificate signed by the Picolo root servers. This certificate is the seed credential that lets the node become part of the network and is presented to other nodes when it wishes to be part of the core routing layer. With respect to the network layer semantics for making connections, we shall use either the QUIC protocol or the SPDY protocol.

QUIC is a relatively new transport protocol designed by Google [24] that provides encrypted, multi-session transport over UDP instead of TCP for HTTPS traffic. It replaces the traditional HTTPS stack: HTTP/2, TLS, TCP with an integrated protocol layer directly over UDP. The key features include reduction in the head-of-line blocking delay and handshake delays to improve the performance for multi-session connections. However, QUIC is relatively new and the knee-jerk reaction from firewall manufacturers has been to block it until its fully understood. Also it does not work well with NATs that provide STUN protocol [25] support for drilling holes through their firewalls. The network layer would use techniques derived from the STUN protocol to traverse nodes that are behind a NAT. By having a set of nodes (such as ones with a higher stake) behave as super-peers, its possible to craft a p2p version of the STUN protocol to enable other nodes behind a NAT to become part of the network [26].

SPDY (pronounced "SPeeDY"), is a protocol from the Chromium Project at Google [27] to improve network performance for web pages. It requires changes at both end points for the protocol to work and is able to improve load times by about 64% on average. It builds upon previous proposals such as the Stream Control Transmission Protocol (SCTP) [28] that allows multiple sessions to coexist on a single TCP connection.

SPDY can work well in network environments where firewalls are biased towards blocking most ports but the HTTP and HTTPS. SPDY allows bi-directional session initiation to happen and thus is more p2p flavored that other web protocols. It also has multi-session support over an SSL based secure layer that can utilize certificates as credentials among other methods.

4.5. Trust and Governance

4.5.1. Analytics and Debug/Fault diagnosis

Nodes maintain local analytics which are periodically updated to a special distributed system-level analytics database. Analytics include counters for table queries, statistics for various operations (CRUD), latency numbers, route changes and similar network statistics. The analytics database is a special permissioned database that only entities (developers, institutions) with the right credentials can modify. A similar but separate database is used for fault diagnosis, collecting data on Byzantine failures, crash logs and unexpected node departures. A third database is purely used for logs, both at the network level and the database level. Both databases are readable by all nodes who are part of the network. These databases provide the basic mechanism for establishing trust and stake for nodes in the network.

4.5.2. Trust ladder

While we haven't created a detailed specification for how trust and governance work in the network, the basic principles are the following:

- New nodes that obtain a fresh certificate, start at the lowest level of trust. They are able to participate in read-only functionalities of the network, such as basic routing, providing read-only replicas and such.
- Using the network, debug/ fault and analytics logs, the network is able to elevate the trust level of a node. This can be seen as a form of stake. The network can also allow nodes that place a security deposit to start at a certain level of trust.
- Nodes at a higher level of trust gain access to great amounts of functionality. For example, the ability to change the principal routes, or be able to bootstrap new nodes into the network requires a higher level of trust or stake.
- The highest level of trust includes nodes that have the ability to update node binaries, issue certificates, grant access to other nodes and similar acts of governance. While it might be possible for regular nodes to enter this level through a combination of service-based trust and stake, the membership might be managed by a council of network admins or developers.

5. Database Subsystem

In this section, we discuss core database concepts like byzantine paxos, role of timestamps in achieving external consistency, distributed transactions and sharding. Concepts that are most new to a database network built for the decentralized world like distributed query processing, dynamic clustering, data sovereignty and decentralized fine-grained access control are presented.

5.1. Blockchain consensus

One of the key questions to answer in a decentralized environment is that of the *validity, credibility or consensus* on a transaction. For example, when a monetary transaction is made among two or more parties, is there consensus among the involved parties, or among the network as a whole ?

In Picolo, when defining the database schema, it is possible to specify if a consensus is required among the all participating nodes on the network or only among the participating entities in the application or any combination of these. It might also be possible to specify a requirement of validation or permission from entities such as regulatory bodies if they exist in the context of the application.

Once the blockchain grants consensus to a transaction, the credential is captured as a multi-party signed certificate which can be presented to Picolo to authorize the transaction on the database. Picolo can work with any and all existing consensus methods used by the major blockchains today. Also for transactions that are public on a public ledger, it is possible to insert a cryptographic reference to the transaction which would represent a database read operation on Picolo, thereby making it possible to share the transaction as a link (on a ledger or otherwise).

5.2. Database Consensus, replication & sharding

All data on Picolo is replicated for durability and high availability. Storage consumers have the option of choosing a replication factor that suits their needs. They can also choose where to locate their data based on where their users are located to achieve better latency or to comply with regulations like GDPR. This data locality can easily be achieved by instructing the network layer to only allow replicas that belong to a geographic region to join the replica group. Consensus is achieved amongst replicas by a running a variant of paxos that is tolerant to byzantine faults.

5.2.1. Paxos based replication

We modify the algorithms in [29] to make the leader election more frequent and add a slashing condition that punishes byzantine behavior. There are two roles that each replica can take: proposer and acceptor. Proposers initiate changes to the state by proposing new commands to be appended to a sequence from which the replica state is generated and acceptors vote on which sequence to accept. The system moves through different views. A view can be thought of as a discrete time period (in the order of minutes) with a monotonically increasing view number $\mathsf{view}_{\mathsf{num}}$ and has a distinguished proposer called leader. If one imagines that each replica has a number from the set $1..\mathcal{N}$ where $\mathcal{N}$ is the number of replicas, then the leader for a view can simply be chosen as $\mathsf{leader}_{\mathsf{view}} = \mathsf{view}_{\mathsf{num}} \ \% \ \mathcal{N}$. There are two modes in which the consensus process happens: fast and classic.

Fast mode: In fast mode (equivalently, leaderless mode), proposers can directly send commands to acceptors bypassing the leader. This obviates the need for phase 1b messages of classic paxos. In a network where replicas are present in distant geographic locations, the savings could be significant. Note that all messages are digitally signed, so the senders can be uniquely identified. A message is a tuple ($\mathsf{view}_{\mathsf{num}}$, seq) where seq consists of a prefix - the last accepted command sequence, suffixed with new commands. This differs from the classic paxos algorithm where only scalar values are passed around and offers two major advantages:

- Commands need not be exactly similar - commands can appear in differing orders in different replicas as long as they are commutative.
- Proposers don't need a promise from acceptors that they will not accept values with a lower ballot number

In the context of a database, two commands are commutative if they are mutating independent records that don't depend on each other for state calculation. For example, reading a row and writing to another row in a table are commutative operations where as reading and writing to the same row may not be. Even writing to different rows if they have different timestamps is non-commutative if *external consistency* is to be maintained (**§5.3**). The relaxed definition of command similarity helps replicas achieve consensus quicker compared to the usual case. Since we cannot assume synchronicity of the network, messages may appear in different order at different replicas. So as long as they are commutative, we can tolerate the order difference and proceed with the consensus process. The acceptors always accept a sequence with the highest length, so they don't need to send back the ballot number promise.

Classic mode: It is possible that the acceptors are unable to make progress in fast mode *i.e* append new commands to their sequences when they receive concurrent proposals. Since the network is asynchronous and messages may reach out of order, if they are non-commutative and are of the same length, the acceptors cannot agree on the order by themselves. So they fallback to the leader to arbitrage an order for them. They

send their sequences to the leader of the current view $\mathsf{leader}_{\mathsf{view}}$ who then executes a classic ballot to achieve consensus.

Byzantine fault detection: Once acceptors receive new proposals, they first verify if the new sequence contains as a prefix an already accepted sequence by them in the past. If not, they simply reject it. If it does, then they sign their acceptance and multicast it to other acceptors. Other acceptors then perform the same check and if it passes, signal their acceptance by *appending* their signature and multicast it again. This process continues until $\mathcal{N} - f$ acceptors each receive messages with $\mathcal{N} - f$ signatures at which point, the sequence is considered agreed upon and a message is sent back to the proposer indicating consensus. Here $\mathcal{N} \geq 3f + 1$. During this process if an honest acceptor receives a multicast that contains signatures of an acceptor $\mathcal{M}_{\mathcal{A}}$ on two non-commutative sequences (see Fig. 4), then it will trigger a slashing condition (**§6.2**). Since we require at least $\mathcal{N} - f$ acceptors to agree on a proposal, at least one of them is guaranteed to be honest and it will trigger the slashing condition.

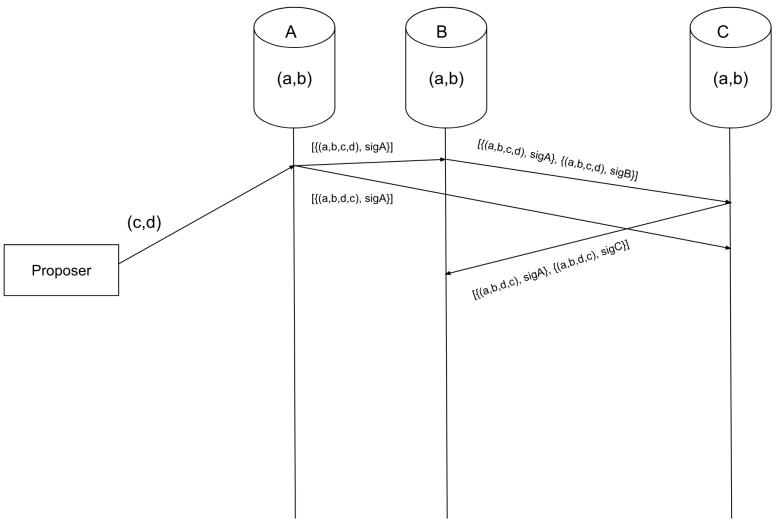

Figure 4: Byzantine fault detection during Paxos. Replica A sending conflicting messages is detected by B and C

5.2.2. Sharding

Picolo automatically partitions data into multiple shards when it grows too big for any one replica. Each shard consists of a set of key ranges (typically 64MB). A key range is the smallest atomic unit that is replicated aka governed by a paxos group. It is also the smallest unit of movement when data from one replica needs to be sharded into distinct replica sets. The process of sharding is a well studied problem in databases and implementations can be readily found, so we omit a detailed discussion here.

5.3. External consistency, Hybrid time & Distributed transactions

Intuitively, external consistency property [1] of a distributed system guarantees that the state changes seen by it are exactly in the order applied to it by an external actor. It is the strongest form of consistency a distributed database system can offer and is notoriously hard to achieve in a system with asynchronous network links. Google's Spanner [30] achieves this by using the TrueTime api which exposes clock uncertainty as an interval. Every transaction in Spanner has a TrueTime timestamp that indicates its time of entry into the system. Maximum clock uncertainty guaranteed by Truetime is 7ms (due to datacenter latencies); which means by making every transaction wait for 7ms before committing, Spanner can guarantee their total ordering system wide. While these low *commit-wait* latencies (7ms) made possible by state of the art datacenters work fine for Spanner, a decentralized network like Picolo has no such luxuries and needs a new trick.

5.3.1. Using timestamps

Picolo uses hybrid time [31] - a combination of physical clocks and logical clocks for timestamping. Each node is assumed to provide an accurate enough physical timestamp by running a daemon process that syncs time with NTP stratum 1 servers like Google's external NTP service. If a node's clock is off by more than 500ms from the average of its paxos group, Picolo automatically kills it and finds a replacement. The logical clock is simply a monotonically increasing number appended to the physical time; so its possible to do a simple lexicographic comparison to determine relative order. For each write transaction, $\mathsf{leader}_{\mathsf{view}}$ assigns a hybrid timestamp to it before proposing it to replicas. So when two requests r1 followed by r2 hit the same paxos group and pass by $\mathsf{leader}_{\mathsf{view}}$, all the replicas are guaranteed to commit requests in the same order irrespective of the order in which they are received. For transactions that span multiple paxos groups, a coordinator (one of the $\mathsf{leader}_{\mathsf{view}}$ of individual paxos groups) is elected to perform a 2 phase commit (2PC) with other leaders. In the prepare phase of 2PC, each leader acquires locks for its corresponding group and replies with its current timestamp. The coordinator then selects the highest timestamp of all the replies and uses it as the timestamp of the transaction during apply phase. This timestamp is also sent back to the initiator of the transaction so that it can pass it to a subsequent causally related transaction (if any). In the case where this propagation is not possible, external consistency cannot be guaranteed and application developers need to employ the commit wait strategy used by Spanner if desired, although at a performance expense.

5.3.2. MVCC

Multi version concurrency control (MVCC) is the practice of storing multiple versions of the same data. In Picolo, each record is uniquely identified by a timestamped key. So keys that differ only by their timestamps represent different versions of the same data. By keeping these different versions, Picolo offers such features as lock-free reads, time travel reads and snapshot isolation. Clients executing read transactions therefore need not acquire any locks since any concurrent writes on the same record have a newer timestamp and won't affect its older timestamped value. Records can also be fetched from the past by specifically executing a read transaction with an older timestamp. For writes however, a lock needs to be acquired on the key being modified. Lock status of keys is stored in a lock table and transactions that mutate data need to check whether the key/range of keys they try to operate on is already present in the lock table, in which case they need to wait. Else, they create an entry in the lock table (thereby acquiring the lock) before proceeding. Similarly, for writes spanning multiple paxos groups, one of the $\mathsf{leader}_{\mathsf{view}}$'s is chosen as the coordinator of a 2PL (two phase locking) process that facilitates the locking of keys/key ranges of each group by its respective leader.

5.4. Data in web 3.0

Our long term goal for Picolo is to make it *the* data network for web 3.0 (**§6.4**). Data in the new web will be solely controlled by data owners and resides in a single shared network. To support such a network, following capabilities must be built:

5.4.1. Distributed query processing

A web 3.0 data network needs to have the capability to execute queries across *all* nodes in the network in response to a client request. Since nodes will have vastly differing schemas, effective mechanisms for searching and query proessing [32, 33, 34] are needed. High level architecture of a Picolo node is depicted in Fig. 5. The schema dictionary depicted in the figure contains metadata to be exposed to the world. Nodes that wish to keep data private should not export the data's metadata to the schema dictionary. Suppose there is a table called users with four columns: username, firstname, lastname and email. The exported metadata entry might look like:

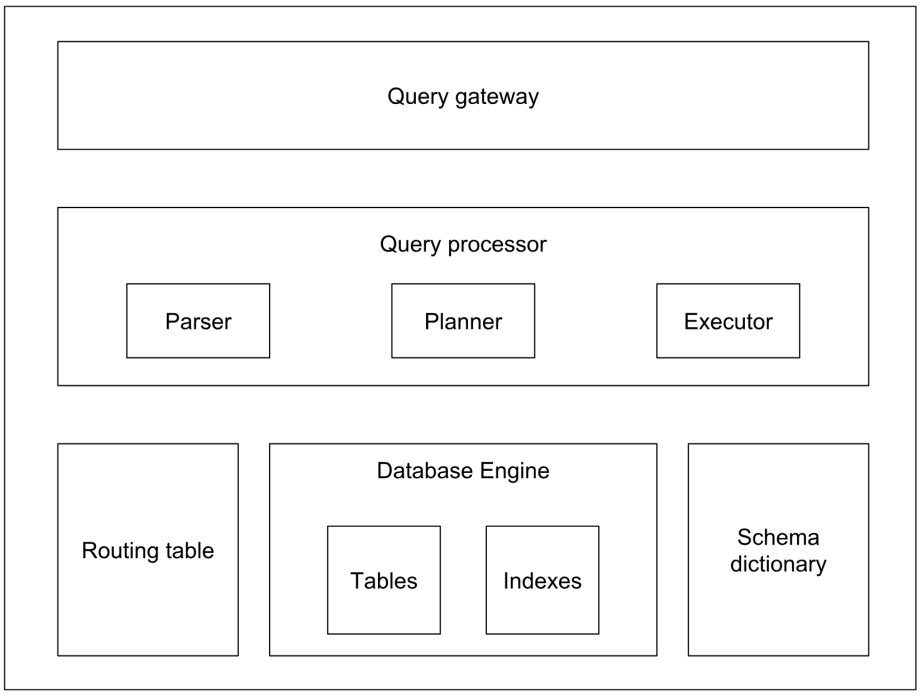

Figure 5: A Picolo node

Entity	Keywords
users	user, people, customer, profile
name	name
firstname	fn, f_name, first_name
lastname	ln, l_name, last_name
email	id, contact

When a query is posed to any node in the system, the node first tries to fulfill it with a local query before passing it along to its neighbors. Remember, nodes in the system host data with heterogeneous schemas. Hence the keywords are used to find semantically similar data by assisting in query reformulations.

Dynamic clustering: Semantic proximity metrics [2] and clustering techniques [35] can be used to find nodes hosting semantically similar schemas. Overtime, these nodes are discovered and are clustered together for better query performance by reducing the number of network hops required.

5.4.2. Data sovereignty

Picolo supports two different schema types: *application controlled* and *user controlled*. Applications can use user controlled schemas to put users in absolute control of data and better comply with regulations like GDPR. For example, a decentralized twitter may want users to have control over their tweets. Users can use any third party client or Picolo's official clients to interact with their tweets, effectively rendering the decentralized twitter just another client to the data albeit with better features.

When semantics don't allow to put user in control of data, application controlled schemas can be used. An example here would be a decentralized ticketing application where users should not be given fine grained control to selectively delete data about which tickets they bought.

Access control: Applications and users may want to share data with other parties but may wish to impose

access controls. There are a few ways of achieving this including building an API on top of the data or using proxy re-encryption techniques. In [36], a secret sharing scheme is used instead, where a party given access to encrypted data collects key shares from key holders, reconstructs the key and decrypts data for further use. Acces rules can be defined by a SQL like declarative language at any granularity desired like at the level of a single row or a cell. An example row level granularity rule looks like:

```
SELECT *
FROM users
WHERE email=foo@bar.com
NODE (SELECT nodeId FROM nodes WHERE domain=application)
```

An example value level (only username is given access to) granularity rule looks like:

```
SELECT username
FROM users
WHERE email=foo@bar.com
NODE (SELECT nodeId FROM nodes)
```

Here `NODE` is a new SQL clause that identifies nodes/actors in the network that have access to the data covered by the rule. Each access rule creates an ecrypted block of data. When there are overlapping rules, multiple encrypted blocks are created with associated keys. The scheme also supports updates to rules - however updates cause data re-encryption and key re-distribution.

5.4.3. Role of encryption

Attribute based encryption (ABE) [37] was first described by Goyal et.al as a way to achieve fine grained access control on data stored with a third party. It allows attaching policies or access structures to cipher texts and allows decrypting them only if the decryption key has attributes that satisfy the access structure. As an example, in a patient-disease database, a policy could be to allow to only decrypt records where the disease is flu and only keys with the attribute *flu* would be able to decrypt them. ABE comes in two forms: ciphertext-policy ABE (CP-ABE) and key-policy ABE (KP-ABE). In CP-ABE, policies are attached to ciphertexts and attributes are attached to keys whereas in KP-ABE, policies are attached to keys and ciphertexts are labelled with attributes. While these techniques sound promising, there don't seem to be many practical implementations of them in databases. Sieve [38] uses KP-ABE to protect user data stored in files with a cloud storage provider. It uses an hybrid encryption scheme where data itself is ecnrypted using a symmetric key and metadata related to the file including its location and the symmetric key is encrypted with KP-ABE. A client first gets the metadata, decrypts it using a key which has a corresponding policy, downloads the file and decrypts it locally with the symmetric key. A similar technique is used in [39]. The ABE schemes used in either of these systems seem to be secure only under a selective-set model which may not be sufficient in all adversarial scenarios. Moreover, key revocation in Sieve requires re-encrypting data which may not scale well while in [39], the effectiveness of key revocation is not discussed in detail.

Another body of research focuses on ABE schemes that are CCA2 secure. In [40] a CCA2 secure KP-ABE scheme is proposed which is based on a large universe construction of another KP-ABE scheme. They overcome the limitations of the underlying scheme by adding a dummy on-the-fly attribute to the decryption procedure which is obtained by running a temporary message through a chameleon hash function. In [41], authors describe a scheme that is adaptively secure under the complexity assumptions of 3-prime subgroup decision problem. Their scheme allows dynamic update of policies tending itself suitable for practical use cases. But similar to Sieve, the data needs to be re-encrypted by the storage provider (albeit without sacrificing confidentiality). EASiER [42] is a system that allows for dynamic policy update without the need for data re-encryption. It uses a minimally trusted proxy that faciliates efficient access revocation and can be used in constructing practical systems that use ABE for access control.

Picolo is the first system that uses ABE for fine-grained access control in a distributed database. It al-

lows data owners to set access rules via the declarative language above and uses a CCA2 secure ABE scheme with proxy assisted revocation. Picolo's construction will be more comprehensively detailed in an upcoming paper.

6. Mechanism design

Since Picolo is an open network and nodes can't be trusted, a robust incentive and disincentive mechanism is required to realize its correct functioning. In this section we discuss how Picolo handles failures and malicious nodes at the database and network layers. Our mechanism design consists of these main pieces:

1. Node stakes and incentives
2. Data availability checks
3. Paxos based byzantine consensus [29]

We aim to devise Casper FFG [43] style slashing conditions based on the above.

Assumptions in the attack model: We assume a standard Byzantine failure model, i.e., the attacker is able to make changes to the messages at the network layer on a node or alter the behavior at the database layer. The attacker is also able to coordinate the behavior of multiple nodes in real-time to achieve a desired attack scenario. We also allow for the adversary to delay communication between honest nodes so long as the adversary has the ability to do so given the topology of the overlay network (i.e. the adversary should be part of the routing path). We also assume that the attacker is computationally bound, i.e., they cannot gather enough computing resources to subvert state-of-the-art cryptographic techniques such as Elliptic curve, crypto-hash functions (such as SHA-256) or encryption schemes such as AES.

Node stake: All nodes in the network need to have a stake in order to have any meaningful participation i.e before they can start hosting large amounts of data. They can either explicitly make a security deposit when they join thereby increasing their stake or can earn rewards overtime by serving large amounts of data (by acting as cache) and use them for making the deposit. While acting as caching nodes, they are not subject to any PoQ checks (**§6.1**).

Byzantine behaviour at the database layer: The central tenet of our design is to use strong cryptographic primitives to push Byzantine behavior at the database layer down to either a DoS style attack vector or to the network layer where we handle it using a combination of crypto-incentives, DHT management and detection. At the database level, each node is responsible for the standard CRUD operations for a given table (or a shard). Each such operation is protected using a cryptographic signature of the entity (dapp or user) that authorizes the request. Thus a malicious node is only able to destroy the data and not alter it. For example, every write request has a signature that certifies the request. Thus, a Merkle proof audit and signature on every table can deter any malicious manipulation of the data stored. The only attack vectors that remain are various versions of DoS style disruption, where a single node or a collection of nodes disrupt the network by delaying messages or destroying the data stored.

Network level Byzantine behavior: An adversary can drop network messages, delete data or both. The routing layer is based on a cryptographic DHT. Thus, it is not possible for the attacker to "choose" to store a particular shard or content. The only way an attacker can do a targeted DoS is by flooding the network with a majority of nodes such that with high probability one of the compromised nodes gets the responsibility for a targeted table or shard. This mapping includes the backup nodes, thus limiting the effectiveness of a targeted attack.

Caching and replication layer incentivizes faster network links and nodes: The caching and replication layer prioritizes faster network links and nodes to serve requests. Thus, nodes that are dead, unresponsive or slow will get fewer requests over time. Thus, a DoS style attack scenario will cause diminishing impact over time. While the network adapts to such a disruption, it is possible for the attacker to gain enough stake and temporarily cause a slowdown. However, their stake (or trust level) would go down and they would be responsible for fewer network functions over time essentially degrading their influence.

Detection: It is be possible to detect such DoS style behavior quickly at the network layer since the overlay topologies tend to share many links at the layer-3 on the Internet. For example, if a node appears to delay messages at the network level while maintaining fast routes and content availability might indicate malicious intent. A detection "service" can be configured to run on nodes that have higher stake or trust. They can detect anomalies between different layers of a potentially malicious node and agree to reduce the trust level of that node. Such a detection service can only be thwarted if the malicious node coordinates its behavior across all observable metrics such that it appears as if it is naturally faulty. For this case, it is indistinguishable from a genuinely faulty node for all practical purposes which is handled at the DHT layer (node addition, deletion, failure modes).

6.1. Data availability checks

All peers (replicas) in a database cluster constantly ping each other (once every 10 seconds) to check if they are still reachable from one another. These checks are important to ensure that the replication factor of data is always maintained and for automatic fail-over. Normally, these pings contain short meaningless data. We enhance these pings with random checks for actual data nodes are supposed to be storing. We term this Proof-of-Query (PoQ) and nodes can prove that they are storing and serving data correctly by responding satisfactorily to PoQ pings. There are two types of PoQs: peer-PoQ or pPoQ and client-PoQ or cPoQ (see Fig. 6)

pPoQ: This is the PoQ that peers (replicas) within a cluster send amongst each other. In addition to ping data, peers randomly query other peers for tiny slivers of data (typically a single column value of a row aka a single cell) and compare the results from each peer against one another and with local data. If there are disparities among the results, then the entire row is fetched along with the digital signature of the row and the public key of the signer. If the signature verification fails for the data returned by a peer, then the validating peer publishes that failure as a violation and triggers the slashing condition. Then the violating peer's deposit is slashed, with a reward going to the validating peer.

cPoQ: This PoQ works similar to above except that it can be triggered by any client of the network, typically the party most interested in the data hosted by the cluster (a dapp). cPoQ acts as a second level security check in case all the peers in the cluster are complicit. It is up to the clients how frequently they want to send cPoQs, the trade off being data transfer from a cluster (network egress) costs money.

Source of randomness: It is important to choose a good source of randomness so as to ensure queries cannot be guessed ahead in time. If one imagines the data set hosted by a cluster as a giant array and the array size is known, a simple secure random number generator that generates numbers between 0 and 1 will work. Verifiers can then simply get the record at index $\lfloor \text{random} * \text{size} \rfloor$

Note on signatures: In our current plan signatures are stored per row. So when a row update occurs on a specific column or group of columns, the signature needs to be recreated for the whole row. This involves fetching the whole row, applying the update, creating the signature on the new row and putting back the whole row. The data overhead with this process might prove to be too much. We are evaluating an alternate scheme where signatures are stored per cell instead using a short signature scheme like BLS [44]. Trade offs between these two schemes need to be more thoroughly evaluated.

6.2. Slashing conditions

There are three conditions, meeting any of which will result in the node losing its stake:

1. Node found to be malicious during a paxos write
2. Node is continuously failing PoQ checks
3. Node not responding to PoQ checks

Malicious behavior during paxos write: We use a modified version of generalized byzantine paxos algorithm [29] for achieving replica consensus (**§5.2**). The modified version reports back nodes in the paxos group

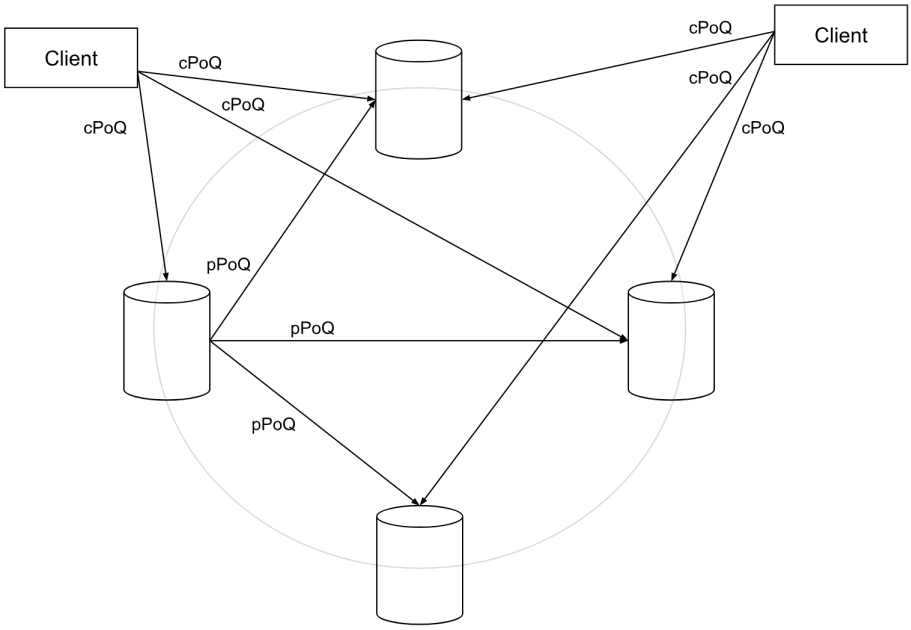

Figure 6: Proof-of-Query pings

that behaved arbitrarily during a write to the initiator of the write (client). Client can then issue cPoQs (**§6.1**) to the reported nodes and check if any data corruption occurred. It may also issue new writes and check if paxos is still reporting back arbitrary behavior of the nodes. It can then publish a proof of arbitrary behavior - simply the wrong query results (that fail signature verification) returned by the malicious nodes to a slashing smart contract that slashes deposits of malicious nodes. Malicious nodes cannot repudiate arbitrary behavior since all messages including responses to cPoQs are digitally signed.

PoQ check failure: Similar to above, if nodes fail PoQ checks i.e return results whose signatures don't match the respective results, initiator of PoQs can publish proofs to the slashing smart contract to trigger deposit slashing. PoQs can also fail with no data returned due to a genuine hardware failure. In this case more PoQs and calls to collect file system statistics are fired to the node and if these calls indicate hardware failure, the node is simply removed from its paxos group and a replacement will be found. The node can then correct the hardware problem and join a new paxos group or simply choose to withdraw its stake and leave the network.

PoQ check unresponsiveness: When a node has not explicitly left the network but stops responding to queries, its deposit is slowly drained. Arguably this method is more punitive than is necessary, the alternate being simply marking the node dead and finding a replacement, but stake draining acts as a deterrent to nodes going silent abruptly. It keeps the network more reliable and performant as replacing a node can be an expensive process. The function that calculates draining rate should take into account these factors:

1. Uptime of the node, t
2. Amount of data on the node, $\mathcal{D}$
3. Current load, ℓ
4. Rewards earned so far, $\mathcal{R}$

The function should be inversely proportional to t, ℓ and $\mathcal{R}$ and directly proportional to $\mathcal{D}$. These relationships make sense because a node with a long uptime might need repair and is genuinely down, a node under heavy load may not be able to respond to PoQs in time as it is busy serving clients, a node that has earned a large amount of rewards so far is trustworthy as it has been serving requests correctly so far and finally since

replacing a node hosting a large amount of data is expensive, it must be penalized for going offline abruptly. Representatively,

$$\mathcal{F}_{drain} \propto \frac{\mathcal{D}}{t * \ell * \mathcal{R}}$$

The exact function to calculate the rate of drain requires experimental evaluations with some trial & error.

6.3. Market design

There are a limited number of tokens (PINTs - Picolo Network Tokens), created at genesis although more can be created in the future. They will be distributed according to a well defined plan amongst investors, developers, partners and network participants. The primary economic activity in the network is that the nodes powering the network are paid by consumers for storage, compute and network bandwidth. And the goal of our market design is to optimize towards a single goal:

"Value of a PINT should increase with usage of the network"

To that end, we propose the following mechanisms:

6.3.1. Work token model

In the utility token model, tokens suffer from the velocity problem. The model also requires that tokens be widely distributed aka into the hands of the consumers which could prove difficult to achieve in practice. Mandating the use of network token to pay for services also introduces friction. The user has to convert fiat to ETH/BTC first, then convert them again to the native network token on a different exchange before he can use the network. Providers on the other hand need to again convert these tokens into more mature/liquid tokens in order to realize gains as quickly as possible as the cost of them holding on to these tokens could be higher than they like [45].

To avoid these problems we use the work token model [46]: network tokens are only required for storage providers to earn the right to join the network. Anyone wishing to become a node on the network need to put up a stake in PINTs in proportion to the resources they'd be providing - more storage provided means more stake required. This stake will be slashed if they are found to be Byzantine (**§6.2**). They can then accept payment from users in any form they prefer including in PINTs. This token model also captures the value created by the network 100 % - when the demand for the network goes up, more providers will want to join the network by purchasing PINTs to use as stake. This puts an organic upward price pressure on the token and vice-versa.

6.3.2. Fair cooperative incentive

There are four types of participants in the network (**§3.2**). The inentives for storage providers and consumers are straight forward - providers get paid by consumers who are benefitting from the service. For the other two types, it is a bit more involved. While the data providers can monetize their data by imposing access controls (**§5.4.2**) and charge for sharing decryption key information, they might not want to do that. They might be genuinely interested in sharing their data for altruistic purposes or are benefitting in a non-monetary form (e.g: by letting developers create innovative services with the data). But this leads to *free riding* problem and *greedy behavior* amongst nodes. Emmanuelle et.al [47] defined the *fair resource sharing* problem and desgined an incentive framework that rewards nodes for acting in a way that maximizes the utility of the network. However, their design depends on a "middleware layer" that tracks various metrics to properly incentivize and punish nodes. In addition to introducing asymmentry into the network, the tracking mechanism depends on nodes truthfully reporting metrics data. Clearly, this strong assumption will not hold well in a Byzantine world as nodes can incorrectly report metrics about their peers in an attempt to damage their participation and access levels as defined in [47]. To solve the above two problems, we propose the following mechanism. First some rules and definitions:

Rule 6.1. Reply: A node should reply to GET messages (**§7**) if its dataset contains records that satisfy the query in GET message.

Rule 6.2. Forward: A node should forward GET messages with positive hop numbers to neighbors.

Definition 6.1. Semantic group: A semantic group is a dynamic cluster (**§5.4.1**) of peers that share semantically similar data.

Definition 6.2. Access level $\mathcal{A}\ell$: Access level of a node is the probability with which its queries will be fulfilled or other queries will be forwarded to by peers in its semantic group.

Definition 6.3. Cooperative: A node is said to be cooperative if it follows rule 6.2.

Definition 6.4. Fair: A node is said to be fair if it follows rules 6.1 and 6.2.

A rational node's objective is to keep its access level high in order to get high quality results to its queries. The access level of each node in a semantic group is tracked by all nodes in the group in a ***non-interactive*** way i.e a node cannot influence what its peers think the access level of a particular node should be. There is no consensus process amongst nodes to determine a common access level value of a node. Each node determines for itself what the access levels of its peers are based on its past interactions with them. For e.g: if a node $\mathcal{A}$ has responded to a node $\mathcal{B}$'s queries satisfactorily in the past, then $\mathcal{A}$'s access level as far $\mathcal{B}$ is concerned will be high and it will respond to $\mathcal{A}$'s queries favorably in the future. Thus,

$$\mathcal{A}_{\mathcal{A}\ell} \propto \sum_{i=0}^{n} \mathcal{N}_i[\mathcal{A}_{a\ell}] + \epsilon$$

where $\mathcal{N}_i$ is a node in the semantic group and $\mathcal{N}_i[\mathcal{A}_{a\ell}]$ is $\mathcal{A}$'s access level at that node and ϵ is a minimum access level all nodes have. A new node can elect to increase its access level (the ϵ) by depositing PINTs to a smart contract. Peers check the smart contract when they receive a GET message from this node for the first time and update their local information. Each node maintains three metrics that will help determine the access levels of its peers:

1. The number of times a peer $\mathcal{A}$ followed 6.1, $\mathsf{num}_{\mathsf{reply}}$
2. The number of times a peer $\mathcal{A}$ followed 6.2, $\mathsf{num}_{\mathsf{fwd}}$
3. The "quality of results" returned by a peer $\mathcal{A}$, qual $(0 < \mathsf{qual} < 1)$

qual is loosely defined and its up to each node how to calculate this value as they are the ultimate consumers of results and its in their best interest to faithfully determine it. Thus, the access level of node $\mathcal{A}$ as per a peer $\mathcal{B}$ can be represented as,

$$\mathcal{A}_{a\ell} \propto \mathsf{num}_{\mathsf{reply}} * \mathsf{num}_{\mathsf{fwd}} * \mathsf{qual}$$

One might notice that there is no explicit definition of a malicious node in this mechanism. This is ok since there is no way in which a node can influence its peers; it may choose to not reply or forward messages from its peers and deliberately set the access levels of its peers to 0 locally but that does not affect the functioning of the semantic group it belongs to. All nodes set the access levels of their peers purely based on their own observation. So its up to each node to choose a behavior aka the manner in which it follows rules 6.1 and 6.2.

6.4. Applications

Picolo can serve as an enabling layer for interesting consumer games and applications. Picolo can also help in solving scalability challenges of blockchains by allowing them to offload on-chain contract data to its network.

Data network for web 3.0: Our long term vision is to become *the* data network of web 3.0. Data in the new web will be free from data silos and resides in a shared open network. It will be under complete control of data owners but shared with third parties in an access controlled way. Big monolithic applications will co-exist with *micro-functions* that act on data *in-situ* to provide innovative new services to people. Applications will not be pre-built but *functionalities* are created on the fly by acting on personal data and by

being context-aware. These functionalities can be crowd-sourced too. For example, a soccer fan might ask a third party *oracle* for a personalized experience to be created for her in expectation of an upcoming World cup game in which her favorite team is playing. The oracle then sends a *function* to act on her data stored on Picolo's network to create the following experience:

1. First it finds out who her favorite team is by looking at her soccer streaming data or tickets purchased in the past
2. Then finds out her friends who support the same team and those who support the opposition by looking at her social graph
3. Looks at her financial data to see how much she'd be willing to spend on the experience
4. Based on the budget estimate, finds a sports bar in the city that will broadcast the game and books tables or book match tickets and a private jet to fly the group to Russia for a live game watching experience in the stadium
5. At the end, splits bills with friends on an app like splitwise and sends notifications and follow ups for payments

This example is intentionally over the board but sheds light on what is possible when data is liberated from silos. Notice that the data never left Picolo. The *function came to data*, performed all operations within the network and only output actions (like booking tickets) to the outside world. Similar applications can be found in areas such as medical research field where independent researchers can easily share and find data and perform computations without ever needing to download it.

Off-chain storage: Blockchain networks like Ethereum don't need to store all contract data on chain. It doesn't make sense to store contract state of every dapp built on the network across thousands of nodes. It can be offloaded to Picolo for easier, faster querying and cheaper storage. It also increases the performance of the app since storing data on Ethereum consumes a lot of gas and since each block has a max gas limit, there is only so many transactions packed into a single block thereby affecting performance. As we have seen with Cryptokitties, if a dapp goes viral, it becomes slow and unusable due to limited throughput of Ethereum. Other networks like Enigma [48] can also use Picolo to safely store their "secret contract" state of secure computations and avoid bloating blocks.

Blockchain analytics: Picolo can be used to examine the state of blockchains, index and store particularly useful sets of data. Any enterprising developer can find sets of data that are valuable, mine and store them on Picolo. They can then sell access to the data by encrypting it and imposing appropriate access controls (**§5.4.2**). As a concrete example, one might write a program to monitor the contract state of Cryptokitties and keep track of stats like how many kitties are being created, their times of creation, how often they are sired with other kitties, how often they are changed hands, which kitty is the most popular sire, how quickly the giveBirth method is being called, who are the most frequent callers of that method etc. Picolo can also be used to index data from other storage networks like IPFS and Storj.

6.5. Attacks

There are four types of participants in the network (**§3.2**) and a few attacks on/from each of these are discussed below. Storj [7] also describes a few attacks that are relevant to a storage network which we touch upon. Below list is by no means exhaustive and we foresee the emergence of a living document dedicated to describing attacks and their mitigations.

Cheating provider: A storage provider who tries to pretend that they are hosting data without actually doing so. This kind of "passive" attack is mitigated by cPoQ (**§6.1**).

Lying consumer: A storage consumer may report that a provider is failing cPoQs when it actually isn't. To resolve this, consumers are required to report details of the supposedly failed cPoQ, the provider information and give read access to the record that cPoQ is supposed to have returned. Then anyone can check whether the cPoQs are actually failing and determine whether to slash the provider's deposit or not. Practically speaking, a committe consisting of other providers can be verifiably chosen at random to validate this claim

which then arrives at a consensus [49].

DDoS on the network: One simple solution to prevent this attack would be to require all requests burn a miniscule amount of PINTs per each request to the network. Another solution could be to adaptively drop GET messages with high hop numbers when network is under load. Since most genuine messages already know where the data they want is located and have small hop numbers, this adaptive dropping won't affect them. As such, this remains an open problem and more sophisticated solutions will be explored as the network develops.

Selective dropper: A storage provider may selectively drop messages from consumers. If this continues for an extended period, a consumer can simply replace this provider with a new one. Since data is replicated across multiple providers, replacement of one should not create any disruption.

Egregious egress: A storage provider may untruthfully report serving a shockingly high amount of data to consumers. Since egress earns them money, it is expected that any rational provider tries this attack. Since load is evenly spread across all replicas hosting a data set, a replica reporting a high sigma deviation can be detected and counter measures can be taken.

Free rider: A peer in a semantic group (defn 6.1) may not reply to or forward requests. It just joined the network to gather information about the network like what queries are being sent by which clients or gather data from peers while they route it to clients. This attack is mitigated by fair cooperative incentive (**§6.3.2**)

Greedy Ganesh: A peer in a semantic group may choose not to forward any requests to peers or forward them with client information stripped so that they can't reply to clients directly. Then once the peers return results and ask it to route them to a client, it will just combine them with results from its local database and reply to the client as if the whole result set originated from it. Clients then see this peer as a high quality peer and send future requests to it. This leads to the formation of data islands as clients will not be able to discover valuable data hosted by other peers.

Sybil, Google attack & Honest Geppetto: Sybil attacks involve the creation of large amounts of nodes in an attempt to disrupt network operation by hijacking or dropping messages. These attacks are mitigated by requiring nodes to put up stake before they can join the network. Storj paper [7] argues that the Honest Geppetto attack can be mitigated by analyzing relationships between nodes like whether they are operated by the same entity, but it doesn't provide any practical techniques for such an analysis. So relying on the security deposit might be the only solution in near term.

Hostage Bytes: Redundant storage combined with stake slashing conditions will mitigate this attack. Although collusion across multiple malicious nodes is difficult in practice, we need to explore a more robust solution to this attack.

Cheating owner: Storage consumers might evade payment when its due; this can be mitigated by a threshold payment scheme where if the outstanding amount exceeds a certain value, payment is mandated failing which will prevent the consumer from accessing data.

Faithless farmer: Non public data should be access controlled via encryption to prevent this attack. (**§5.4.2**)

7. MX Protocol

Nodes in the network need to speak the same language for efficient discovery and communication of data. Hence the following message exchange scheme (see Fig. 7) is proposed similar to [50]. Note that the exact mapping between the following messages and underlying transport protocol is not discussed here and may change depending on the final transport protocol chosen (QUIC vs TCP)

PUT: A PUT message contains the query to be run, an optional list of nodes to run the query against and an optional max number of hops (needed in case of an empty node list). This is used for creating or updating data in the system.

GET: A GET message contains the query to be run, max number of hops, the number of results to retrieve, the mode of retrieval (pull vs push) and a transaction identifier. Client sends this to a server to retrieve results that match the query. Parameters in GET can be varied depending on application needs - a streaming application may choose the push mode in which server pushes data to the client as it becomes available up until the specified number is met. A latency sensitive application may choose to retrieve a small number of results in a batch in one pull.

SEND: Servers respond to each GET message with one or more SEND messages with results. In pull mode, there is only one SEND message followed by an END message where as in push mode there are multiple SEND messages followed by an END message.

END: Servers send END messages to clients to indicate that they have finished sending all results in response to a particular GET message.

CLOSE: A client can send a CLOSE message to the server to indicate that it no longer is interested in the remaining query results and close the transaction. It doesn't have to wait until all the results are retrieved.

OK: Server sends this message to a client as a positive acknowledgment to a client's message.

ERROR: Server sends this message to a client as a negative acknowledgment to a client's message.

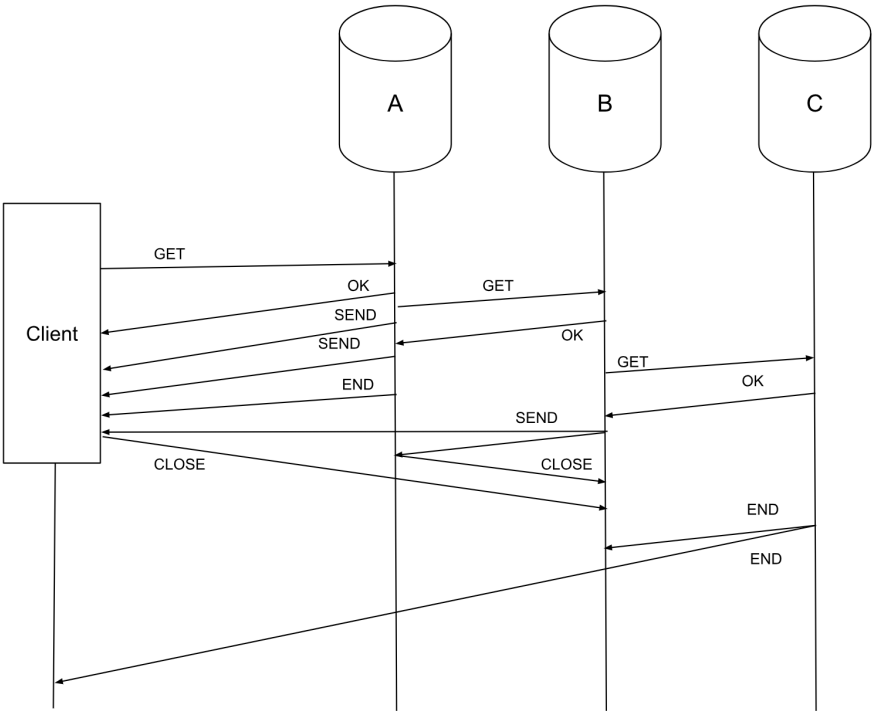

Figure 7: GET message flow, time increasing from top to bottom

8. Conclusions and Future work

We described a database network that serves as the backbone for web 3.0 data. The network is made up of homogeneous nodes that self-organize into a p2p network based on DHTs and run a database software that provides features such as SQL interface, external consistency, auto-sharding of data and self-replication in case of failures. The network is incentive-compatible and has mechanisms in place for rewarding desirable behavior and punishing malicious intent. The network can be used by blockchain apps to store their contract state or normal apps to store generic application data and put the data owner in complete control. We discussed mechanisms to ensure data confidentiality, integrity, availability and a scheme to provide fine grained access control. We listed various attacks that are possible on the network and their respective mitigation/prevention measures.

We leave the practical construction and performance analysis of attribute-based encryption scheme for a future paper. Methods to detect access control leakage, more robust defenses against DDos and Sybil attacks are left as a future exercise. A more generic access control framework based on functional encryption [51] that promises increased privacy and seems to be a step towards the holy grail of fully homomorphic encryption [52] will also be explored in the future.

9. References

[1] David K. Grifford. Information storage in a decentralized computer system. http://bitsavers.org/pdf/xerox/parc/techReports/CSL-81-8_Information_Storage_in_a_Decentralized_Computer_System.pdf.

[2] Wee Siong Ng, Beng Chin Ooi, Kian-Lee Tan, and Aoying Zhou. Peerdb: A p2p-based system for distributed data sharing. http://www.inf.ufpr.br/eduardo/ensino/ci763/papers/peerdb.pdf.

[3] Ryan Huebsch, Brent Chun, Joseph M. Hellerstein, Boon Thau Loo, Petros Maniatis, Timothy Roscoe, Scott Shenker, Ion Stoica, and Aydan R. Yumerefendi. The architecture of pier: an internet-scale query processor. http://www.huebsch.org/papers/CIDR05.pdf.

[4] Alon Y. Halevy, Zachary G. Ives, Jayant Madhavan, Peter Mork, Dan Suciu, and Igor Tatarinov. The piazza peer data management system. http://www.cis.upenn.edu/~zives/research/piazza-tkde.pdf.

[5] Protocol Labs. Filecoin: A Decentralized Storage Network. https://filecoin.io/filecoin.pdf.

[6] Juan Benet. Ipfs - content addressed, versioned, p2p file system. https://ipfs.io/ipfs/QmR7GSQM93Cx5eAg6a6yRzNde1FQv7uL6X1o4k7zrJa3LX/ipfs.draft3.pdf.

[7] Shawn Wilkinson et al. Storj - a peer-to-peer cloud storage network. https://storj.io/storj.pdf.

[8] David Vorick and Luke Champine. Sia: Simple decentralized storage. https://sia.tech/sia.pdf.

[9] Bigchaindb 2.0 the blockchain database. https://www.bigchaindb.com/whitepaper/bigchaindb-whitepaper.pdf.

[10] Neeraj Murarka and Pavel Bains. Technical paper. https://bluzelle.com/wp-content/uploads/2017/10/Bluzelle-Technical-Paper.pdf.

[11] Mediachain an open, universal media library. http://www.mediachain.io/.

[12] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a common api for structured peer-to-peer overlays. In *IPTPS*, 2003.

[13] RFC 1518. An architecture for ip address allocation with cidr. https://tools.ietf.org/html/rfc1518.

[14] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01. ACM, 2001.

[15] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*. Springer-Verlag, 2001.

[16] Venugopalan Ramasubramanian and Emin Gün Sirer. Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04. USENIX Association, 2004.

[17] Indranil Gupta, Ken Birman, Prakash Linga, Al Demers, and Robbert van Renesse. Kelips: Building an efficient and stable p2p dht through increased memory and background overhead. In *Peer-to-Peer Systems II*. Springer Berlin Heidelberg, 2003.

[18] Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. One hop lookups for peer-to-peer overlays. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03. USENIX Association.

[19] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '00. ACM, 2000.

[20] P. Druschel and A. Rowstron. Past: a large-scale, persistent peer-to-peer storage utility. In *Proceedings Eighth Workshop on Hot Topics in Operating Systems*, 2001.

[21] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01. ACM, 2001.

[22] Sharad Goel, Ashton Anderson, Jake Hofman, and Duncan Watts. The structural virality of online diffusion. 62, 07 2015.

[23] History of the root server system. `https://www.icann.org/en/system/files/files/rssac-023-04nov16-en.pdf`.

[24] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17. ACM, 2017.

[25] RFC 5389. Session traversal utilities for nat (stun). `https://tools.ietf.org/html/rfc5389`.

[26] Z. Zhang, X. Wen, and W. Zheng. A nat traversal mechanism for peer-to-peer networks. In *2009 International Symposium on Intelligent Ubiquitous Computing and Education*, May 2009.

[27] Google's Chromium Project. Spdy: An experimental protocol for a faster web. `https://www.chromium.org/spdy/spdy-whitepaper`.

[28] RFC4960. Stream control transmission protocol. `https://tools.ietf.org/html/rfc4960`.

[29] Miguel Pires, Srivatsan Ravi, and Rodrigo Rodrigues. Generalized paxos made byzantine and less complex. `https://arxiv.org/pdf/1708.07575.pdf`.

[30] James C. Corbett et. al. Spanner: GoogleâĂŹs globally-distributed database. https://static.googleusercontent.com/media/research.google.com/en//archive/spanner-osdi2012.pdf.

[31] David Alves, Todd Lipcon, and Vijay Garg. Technical report: Hybridtime - accessible global consistency with high clock uncertainty. http://users.ece.utexas.edu/~garg/pdslab/david/hybrid-time-tech-report-01.pdf.

[32] George Kokkinidis and Vassilis Christophides. Semantic query routing and processing in p2p database systems: The ics-forth sqpeer middleware. http://www.dcs.bbk.ac.uk/selene/reports/sqpeer.pdf.

[33] Peter Triantafillou and Theoni Pitoura. Towards a unifying framework for complex query processing over structured peer-to-peer data networks. https://pdfs.semanticscholar.org/91ca/8c26f621a3fc04ed45f68ade474d43b6e3fd.pdf.

[34] DONALD KOSSMANN. The state of the art in distributed query processing. http://www.cse.ust.hk/zsearch/qualify/DistributedSearch/The%20state%20of%20the%20art%20in%20distributed%20query%20processing.pdf.

[35] Verena Kantere, Dimitrios Tsoumakos, Timos K. Sellis, and Nick Roussopoulos. Groupeer: Dynamic clustering of p2p databases. *Inf. Syst.*, 34:62–86, 2009.

[36] Angela Bonifati, Ruilin Liu, and Hui (wendy) Wang. Distributed and secure access control in p2p databases. https://hal.inria.fr/hal-01056665/document.

[37] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. https://eprint.iacr.org/2006/309.pdf.

[38] Frank Wang, James Mickens, Nickolai Zeldovich, and Vinod Vaikuntanathan. Sieve: Cryptographically enforced access control for user data in untrusted clouds. https://www.usenix.org/system/files/conference/nsdi16/nsdi16-paper-wang-frank.pdf.

[39] Shivaramakrishnan Narayan, Martin GagnÃĺ, and Reihaneh Safavi-Naini. Privacy preserving ehr system using attribute-based infrastructure. http://courses.cs.vt.edu/cs6204/Privacy-Security/Papers/Crypto/Privacy-EHR-ABE.pdf.

[40] Weiran Liu, Jianwei Liu, Qianhong Wu, Bo Qin, and Yunya Zhou. Practical direct chosen ciphertext secure key-policy attribute-based encryption with public ciphertext test. In Mirosław Kutyłowski and Jaideep Vaidya, editors, *Computer Security - ESORICS 2014*, pages 91–108, Cham, 2014. Springer International Publishing.

[41] Zuobin YING, Hui LI, Jianfeng MA, Junwei ZHANG, and Jiangtao CUI. Adaptively secure ciphertext-policy attribute-based encryption with dynamic policy updating. http://engine.scichina.com/publisher/scp/journal/SCIS/59/4/10.1007/s11432-015-5428-1?slug=full%20text.

[42] Sonia Jahid, Prateek Mittal, and Nikita Borisov. Easier: Encryption-based access control in social networks with efficient revocation. http://hatswitch.org/~nikita/papers/easier-asiaccs11.pdf.

[43] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. https://arxiv.org/pdf/1710.09437.pdf.

[44] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. https://www.iacr.org/archive/asiacrypt2001/22480516.pdf.

[45] Vitalik Buterin. On medium-of-exchange token valuations. https://vitalik.ca/general/2017/10/17/moe.html.

[46] Kyle Samani. New models for utility tokens. https://multicoin.capital/2018/02/13/new-models-utility-tokens/.

[47] Emmanuelle Anceaume, Maria Gradinariu, and Aina Ravoaja. Incentive for p2p fair resource sharing. https://pdfs.semanticscholar.org/e17f/ff4829dd45f894751d14e0531084e9de68a4.pdf.

[48] Guy Zyskind, Oz Nathan, and Alex Sandy Pentland. Enigma: Decentralized computation platform with guaranteed privacy. https://s3.amazonaws.com/enigmaco-website/uploads/pdf/enigma_full.pdf.

[49] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. https://people.csail.mit.edu/nickolai/papers/gilad-algorand-eprint.pdf.

[50] K. Berket, A. Essiari, D. Gunter, and W. Hoschek. Peer-to-peer i/o (p2pio) protocol specification version 0.6. http://dst.lbl.gov/~hoschek/firefish/p2pio-spec/spec.pdf.

[51] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. Cryptology ePrint Archive, Report 2010/543, 2010. https://eprint.iacr.org/2010/543.

[52] Craig Gentry. A fully homomorphic encryption scheme. http://cs.au.dk/~stm/local-cache/gentry-thesis.pdf.

[53] Napster. https://en.wikipedia.org/wiki/Napster.

[54] P. Mockapetris and K. J. Dunlap. Development of the domain name system. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88. ACM, 1988.

[55] Ian Clarke. A distributed decentralised information storage and retrieval system. *Master's thesis, Univ. of Edinburgh*, 1999.

[56] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability*. Springer-Verlag, 2001.

[57] Gnutella p2p network. https://en.wikipedia.org/wiki/Gnutella.

[58] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01. ACM, 2001.

[59] B. Y. Zhao, Ling Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, Jan 2004.

[60] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97. ACM, 1997.

[61] P. F. Tsuchiya. The landmark hierarchy: A new hierarchy for routing in very large networks. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88. ACM, 1988.

[62] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01. Springer-Verlag, 2002.

[63] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02. ACM, 2002.

[64] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03. USENIX Association, 2003.

[65] M. Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table. In *Peer-to-Peer Systems II*. Springer Berlin Heidelberg, 2003.

[66] Moni Naor and Udi Wieder. A simple fault tolerant distributed hash table. In *Peer-to-Peer Systems II*. Springer Berlin Heidelberg, 2003.

[67] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The design of a large-scale event notification infrastructure. In Jon Crowcroft and Markus Hofmann, editors, *Networked Group Communication*. Springer Berlin Heidelberg, 2001.

[68] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. *SIGPLAN Not.*, 2000.

[69] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole, Jr. Overcast: Reliable multicasting with on overlay network. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00. USENIX Association, 2000.

A. EIP-1729. Introduce SQL semantics to EVM <WIP>

We are contemplating an EIP (Ethereum Improvement Proposal) to give Ethereum dapp developers a new choice for data storage. Not all data need to be replicated on every full node of Ethereum as the need for strongest security maybe an overkill for most smart contracts. Such data can be stored on Picolo network at a contract defined replication factor. Specifically, the EIP plans to introduce SQL primitives like selections, projections, aggregates etc directly into solidity (feasibility currently being evaluated).

Picolo may also be used by "stateless clients" and "state minimized clients" to query for data that is normally hosted by "archival nodes" in the current implementation or in the upcoming sharded implementation. Picolo's SQL capabilities make it easy for clients to construct complex queries that are not currently possible in the simple key-value lookups provided by the EVM (Ethereum Virtual Machine).

```
<Mapping of SQL commands to EVM/eWASM opcodes to be defined here.  Or maybe just a library
contract or other ‘‘listener’’ mechanism will be easier to implement>
```

B. Background on p2p networks

Since a fast production quality peer-peer network is a critical layer for any decentralized service architecture, we present a summary of the past work in this area for educational purposes.

Napster [53] was one of the first popular services that provided much of the original inspiration for p2p systems although its database was centralized. DNS is an example of a widely deployed distributed and largely decentralized key-value database that powers every lookup and interaction on the Internet [54]. DNS relies on special root servers to bootstrap the lookup protocol. The Freenet [55, 56] and the Gnutella [57] p2p systems were popular in the previous decade for file sharing. Both systems were designed for sharing of large files over a longer duration of time. Content reliabilty including lookup reliability and network latency goals were necessary in this enviroment.

The second generation of p2p systems include research driven projects such as Chord [14], Content Addressable Network (CAN) [58], Pastry [15], Tapestry [59] and Kademlia. Chord along with CAN, Tapestry and Pastry developed the concept of distributed hash tables (DHTs) as a fundamental mechanism for content-based addressing. They built over the scalability and self-organizing properties of both FreeNet and Gnutella by providing a definite answer to a lookup query in a bounded number of network hops. In fact, these protocols are able to locate content within $O(logN)$ where N is the number of nodes in the system. From an API perspective, these overlays provide a key-based routing (KBR) interface that supports deterministic routing of messages to a live node that has the responsiblity for the "value" corresponding to the given key. These systems also support high level APIs such as Dynamic Object Location and Routing (DOLR) [12]. Most DHT's use the concept of consistent hashing to distribute load evenly among the nodes. Consistent hashing refers to a technique which creates 'buckets' of items such that a small change in the buckets does not lead to a large amount of rehashing.

Chord uses such hash functions to map nodes and content uniformly to a circular 160-bit namespace. Chord improves the scalability of consistent hashing by removing the requirement that every node knows about every other node. Each node only maintains about $O(logN)$ state information about other nodes in an N node network. When nodes join or leave, they require $O(log^2N)$ messages to keep the network updated. CAN routes messages in a d-dimensional space where each node maintains a routing table with $O(d)$ entries and any node can be reached in $O(dN^{1/d})$ routing hops. CAN's routing table does not grow with network size, but the number of routing hops grows faster than $logN$. When compared to Chord and CAN, Pastry and Tapestry take network distances into account when constructing overlay topologies. While Chord and CAN use shortest overlay hops and other runtime heuristics, both Tapestry and Pastry construct locally optimal routing tables to reduce any routing inefficiencies.

Pastry and Tapestry share some similarities to the work by Plaxton et al [60] and to the routing layer in the Landmark hierarchy [61]. The approach consists of routing based on address prefixes or otherwise called prefix-based routing. However, both Pastry and Tapestry include an ability to self-organize the network structure and achieve network locality in content mapping which also lends support for replication. In addition, Tapestry also allows some application-based locality management by "publishing" location pointers throughout the network for efficiently locating content and services.

Kademlia [62] is another p2p DHT-based routing system that uses prefix-based routing by arranging 160-bit IDs (node IDs and content IDs) in a binary tree style data-structure for efficient routing. It uses an XOR-based distance metric for building the routing table and for the routing algorithm itself. In terms of its performance and other features, it is very similar to the above systems such as Chord, CAN, Pastry and Tapestry, but it offers simplicity in its routing and lookup algorithms which make it attrative for implementation. IPFS [6] uses a version of Kademlia for locating files for decentralized applications.

Other notable systems include Viceroy [63] which provides logarithmic hops through nodes with constant degree routing tables. SkipNet [64] uses a multidimensional skip-list data structure to support overlay routing, maintaining both a DNS-based namesapce for operational locality and a randomized namespace for network locality. Other overlay proposals such as Koorde [65] and Naor et al [66] attain lower bounds on local routing

state but oversimplify some of the other features.

The third generation of p2p research includes building applications on top of these DHT systems, validating them as novel infrastructures or tuning them for specific use cases. For example, applications such as PAST [20] and SCRIBE [67] are built on top of Pastry. Decentralized file storage application project OceanStore [68] was built on top of Tapestry, while CFS [21] was build on top of Chord. FarSite [19] uses a conventional distributed directory service and could be built on top of Pastry. Another example of an overlay network is the Overcast System [69], which provides reliable single-source multicast streams.