Blocktree: A Distributed Computing Environment

Matthew Carr

October 6, 2023

Abstract

This document is a proposal for a distributed computing environment called Blocktree. The system is designed around the actor model, and it uses actors to encapsulate resources and provide services. The platform is responsible for orchestrating these actors on a set of native operating system processes. The persistent state for the system is stored in a global distributed filesystem implemented using this actor runtime. High availability is achieved using the Raft consensus protocol to synchronize the state of files between processes. All data stored in the filesystem is secured with strong integrity and optional confidentiality protections. Well-known cryptographic constructions are used to provide these protections, the system does not attempt to innovate in terms of cryptography. A network block device interface allows for fast low-level read and write access to file sectors, with full support for client-side encryption. The system's trust model allows for mutual TLS authentication between all processes, without the need to trust a third-party certificate authority. By integrating these ideas into a single platform, the system aims to advance the status quo in the security and reliability of software systems.

1 Introduction

Blocktree is an attempt to extend the Unix philosophy that everything is a file to the entire distributed system that comprises modern IT infrastructure. The system is organized around a global distributed filesystem which defines security principals, resources, and their authorization attributes. This filesystem provides a language for access control that can be used to securely grant access to resources, even those owned by different organizations. The system provides an actor runtime for orchestrating services. Resources are represented as actors and actors are executed by runtimes in different operating system processes. Each process has its own credentials which authenticate it as a unique security principal, and which specify the filesystem path where it's located. A process has authorization attributes which determine the set of processes that it may communicate with. TLS authentication is used to secure connections between processes. Messages addressed to actors in a different process are forwarded over these connections, while messages delivered to actors in the same process are delivered with zero-copying.

The single global Blocktree filesystem is partitioned into disjoint domains of authority. Each domain is controlled by a root principal. As is the case for all principals, a root principal is authenticated by a public-private signing key pair and is identified by the base64url encoded hash of its public signing key. The domain of authority for a given absolute path is determined by its first component, which is the identifier of the root principal that controls the domain. Because there's no meaning to the directory '/', a directory consisting of only a single component equal to a root principal's identifier is referred to as the root directory of the domain. The root principal delegates its authority to write files to subordinate principals by issuing them certificates which specify the path that the authority of the subordinate is limited to. File data is signed for authenticity and a certificate chain is contained in its metadata. This certificate chain must lead back to the root principal and consist of certificates with correctly scoped authority in order for the file be valid. Given the path of a file and the file's contents, this allows the file to be validated by anyone without the need to trust a third-party. Blocktree paths are called self-certifying for this reason. This construction was independently discovered by the author, but a similar system was previously used in the Self-certifying File System (SFS) [4].

One of the major challenges in distributed systems is managing persistent state. Blocktree solves this issue with its distributed filesystem. Files are broken into segments called sectors. The sector size of a file can be

configured when it's created, but can't be changed later. Reads and writes of individual sectors are guaranteed to be atomic. The sectors which comprise a file and its metadata are replicated by a set of processes running the sector service. These service providers are responsible for storing the sectors of files that are contained in the directory containing the runtime in which it's running. The actors providing the sector service in a given directory coordinate with one another using the Raft protocol [5] to synchronize the state of the sectors they store. By partitioning the data in the filesystem based on directory, the system can scale beyond the capabilities of a single consensus cluster. Associated with every file is a Merkle tree of sector hashes, which allows sectors to be verified without reading the entire contents of a file. Encryption can be optionally applied to sectors, and when it is, the key is managed by the system. The cryptographic mechanisms used to implement these protections are described in section 3.

One of the design goals of Blocktree is to facilitate the creation of composable distributed systems. A major challenge to building such systems is the difficulty is isolating bugs when they inevitably occur. Research into session types (a.k.a. Behavioral Types) promises to bring the safety benefits of type checking to actor communication ([1] chapter 9). Blocktree integrates a session typing system that allows protocol contracts to be defined that specify the communication protocol of a set of actors. This model allows the state space of the actors participating in a computation to be defined, and the state transitions which occur to be specified based on the types of messages received. These contracts are used to verify protocol adherence statically and dynamically. This system is implemented using compile time code generation. It frees the developer from dealing with the numerous failure modes that can occur in a communication protocol.

Blocktree is implemented in the Rust programming language. It is currently tested on Linux, but running it on other Unix-like operating systems should be straight-forward. FUSE support from the host kernel is required to mount the filesystem. The system's source code is licensed under the Affero GNU Public License Version 3. The project's homepage is `https://blocktree.systems`. Anyone interested in contributing to development is welcome to submit a pull request to `https://gogs.delease.com/Delease/Blocktree`. If you have larger changes or architectural suggestions, please submit an issue for discussion prior to investing your time in an implementation.

The remainder of this document is structured as follows:

- Section 2 describes the actor runtime, services, and runtime discovery.
- Section 3 discusses the filesystem, its concurrency semantics and implementation.
- Section 4 details the cryptographic mechanisms used to secure file data.
- Section 5 is a set of examples describing ways that Blocktree can be used to build systems.
- Section 6 provides some concluding remarks.

2 Actor Runtime

Building scalable fault tolerant systems requires us to distribute computation over multiple computers. Rather than switching to a different programming model when an application scales beyond the capacity of a single computer, it's beneficial in terms of programmer time and program simplicity to begin with a model that enables multi-computer scalability. Fundamentally, all communication over a network involves the exchange of messages. So if we wish to build scalable fault-tolerant systems, it makes sense to choose a programming model built on message passing, as this will ensure low impedance with the underlying networking technology.

That is why Blocktree is built on the actor model and why its actor runtime is at the core of its architecture. The runtime can be used to spawn actors, register services, dispatch messages immediately, and schedule messages to be delivered in the future. Messages can be dispatched in two ways: with `send` and `call`. A message is dispatched with `send` when no reply is required, and with `call` when exactly one is. The Rust `Future` returned by `call` can be awaited to obtain the reply. If a timeout occurs while waiting for the reply, the `Future` completes with an error. The name `call` was chosen to bring to mind a remote procedure call, which is the primary use case this method was intended for. Awaiting replies to messages serves as a simple way to synchronize a distributed computation.

Executing actions at some point in the future or at regular intervals are common tasks in computer systems. Blocktree facilitates this by allowing messages to be scheduled for future delivery. The schedule may specify a one time delivery at a specific instant in time, or a repeating delivery with a given period. These scheduling modes can be combined so that you can specify an anchoring instant and a period whose multiples will be added to this instant to calculate each delivery time. For example, a message could be scheduled for delivery every morning at 3 AM. Messages scheduled in a runtime are persisted in the runtime's file. This ensures scheduled messages will be delivered even if the runtime is restarted. If a message has been delivered and the schedule is such that it will never be delivered again, it is removed from the runtime's file. If a message is scheduled for delivery at a single instant in time, and that delivery is missed, the message will be delivered as soon as possible. But, if a message is periodic, any messages which were missed due to a runtime not being active will never be sent. This is because the runtime only persists the message's schedule, not every delivery. This mechanism is intended for periodic tasks or delaying work to a later time, not for building hard realtime systems.

The actor runtime is implemented using the Rust asynchronous runtime tokio [`https://tokio.rs`]. Actors are spawned as tasks in the tokio runtime, and multi-producer single consumer channels are used for message delivery. Because actors are just tasks, they can do anything a task can do, including awaiting other `Future`s. Because of this, there is no need for the actor runtime to support short-lived worker tasks, as any such use-case can be accomplished by awaiting a set of `Future`s. This allows the runtime to focus on providing support for services. Using tokio also means that the actor runtime has access to a high performance multi-threaded runtime with evented IO. This asynchronous programming model ensures that resources are efficiently utilized, and is ideal for a system focused on orchestrating services which may be used by many clients.

2.1 Services

One of the challenges in building actor systems is supervising and managing actors' lifecycles. This is handled in Erlang [1] through the use of supervision trees, but Blocktree takes a different approach, one inspired by Microsoft's Orleans framework [2]. Orleans introduced the concept of virtual actors, which are purely logical entities that exist perpetually. In Orleans, one does not need to spawn actors nor worry about respawning them should they crash, the framework takes care of spawning an actor when a message is dispatched to it. This model also gives the framework the flexibility to deactivate actors when they are idle and to load balance actors across different computers. In Blocktree, a similar system is used when messages are dispatched to services. The Blocktree runtime takes care of routing these messages to the appropriate actors, spawning them if needed. A service must be registered in a runtime before messages can be routed to it. The actors which are spawned based on this registration are called *service providers* of the service. Services which directly use operating system resource, such as those that listen on network sockets, are often started immediately after registration so they're available to external clients.

Blocktree uses services to represent a logical collection of actors which implement some kind of computational service for other actors in the system or external clients. A service is identified by a Blocktree path. Only one service implementation can be registered in a particular runtime, though this implementation may be used to spawn many actors as providers for the service, each associated with a different ID. The runtime spawns a new actor when it finds no service provider associated with the ID in message it's delivering. Some services may only have one service provider in a given runtime, as is the case for the sector and filesystem services. The `scope` and `rootward` field in a service name specify the set of runtimes to which a message may be delivered. They allow the sender to express their intended recipient, while still affording enough flexibility to the runtime to route messages as needed. If `rootward` is `false`, the message is delivered to a service provider in a runtime that is directly contained in `scope`. If `rootward` is `true`, the parent directories of scope are searched, working towards the root of the filesystem tree, and the message is delivered to the first provider of `service` which is found. When there are multiple service providers to which a given message could be delivered, the one to which it is actually delivered is unspecified to allow the runtime to balance load. Delivery will occur to at most one recipient, even in the case that there are multiple potential recipients. In order to contact other runtimes and deliver messages to them their network endpoints (IP addresses and UDP ports) need to be known. This is achieved by maintaining a file with a runtime's endpoint in the same directory as the runtime. The runtime is granted write permissions on the file, and it is updated by `bttp` when it begins listening on a new endpoint. The port a `bttp` server uses to listen for unicast connections is uniformly randomly selected from the set of ports in the dynamic range (49152-65535)

which are unused on the server's host. Use of a random port allows many different `bttp` servers to share a single IP address and makes Blocktree more resistent to censorship. The services which are allowed to be registered in a given runtime are specified in the runtime's file. The runtime reads this list and uses it to deny service registrations for unauthorized services. The list is also read by other runtime's when they're searching for service providers.

Messages can be addressed to services or specific actors. When addressed to a specific actor, the message contains an *actor name*, which is a pair consisting of the path of the runtime hosting the actor and the `Uuid` identifying the specific actor in that runtime. When addressed to a service, a message is dispatched using a *service name*, which contains the following fields:

1. `service`: The path identifying the receiving service.
2. `scope`: A filesystem path used to specify the intended recipient.
3. `rootward`: A boolean describing whether message delivery is attempted towards or away from the root of the filesystem tree. A value of `false` indicates that the message is intended for a runtime directly contained in the scope. A value of `true` indicates that the message is intended for a runtime contained in a parent directory of the scope and should be delivered to a runtime which has the requested service registered and is closest to the scope.
4. `id`: An identifier for a specific service provider.

The ID can be a `Uuid` or a `String`. It is treated as an opaque identifier by the runtime, but a service is free to associate additional meaning to it. Every message has a header containing the name of the sender and receiver. The receiver can be an actor or service name, but the receiver is always an actor name. For example, to open a file, a message is dispatched with `call` using the service name of the filesystem service. The reply contains the name of the file actor spawned by the filesystem service which owns the opened file. Messages are then dispatched to the file actor using its actor name to read and write to the file.

The filesystem is itself implemented as a service. A filesystem service provider can be passed messages to delete files, list directory contents, open files, or perform other standard filesystem operations. When a file is opened, a new actor is spawned which owns the newly created file handle and its name is returned to the caller in a reply. Subsequent read and write messages are sent to this actor. The filesystem service does not persist any data itself, its job is to function as an integration layer, conglomerating sector data from many different sources into a single unified interface. The sector service is what is ultimately responsible for storing data and maintaining the persistent state of the system. It stores sector data in the local filesystem of each computer on which it's registered. The details of how this is accomplished are described in the next section.

2.2 Transporting Messages

Messages can be forwarded between actor runtimes using a secure transport called `bttp`. This transport is implemented using the QUIC protocol [3], which integrates TLS for security. A `bttp` client may connect anonymously or using credentials. If an anonymous connection is attempted, the client has no authorization attributes associated with it. Only runtimes which grant others the execute permission allow connections from such clients. If these permissions are not granted in the runtime's file, anonymous connections are rejected. When a client connects with credentials, mutual TLS authentication is performed as part of the connection handshake, which cryptographically verifies the credentials of each runtime. These credentials contain the filesystem paths where each runtime is located. This information is used to securely route messages between runtimes. The `bttp` server is always authenticated during the handshake, even when the client is connecting anonymously. Because QUIC supports the concurrent use of many different streams, it serves as an ideal transport for a message oriented system. `bttp` uses different streams for independent messages, ensuring that head of line blocking does not occur. Note that although data from separate streams can arrive in any order, the protocol does provide reliable in-order delivery of data in any given stream. The same stream is used for sending the reply to a message dispatched with `call`. Once a connection is established, messages may flow both directions (provided both runtimes have execute permissions for the other), regardless of which runtime is acting as the client or the server.

When a message is sent between actors in the same runtime it is delivered into the queue of the recipient without any copying, while ensuring immutability (i.e. move semantics). This is possible thanks to the Rust ownership system, because the message sender gives ownership to the runtime when it dispatches the message, and the runtime gives ownership to the recipient when it delivers it.

2.3 Communication Security Model

A runtime is represented in the filesystem as a file. Among other things, this file contains the authorization attributes associated with the runtime's security principal. The certificate used by the runtime to authenticate is also contained in this file, so other runtimes are able to locate it and the public key contained within it. The metadata of the file contains authorization attributes just like any other file (e.g. UID, GID, and mode bits). In order for a principal to be able to send a message to an actor in the runtime, it must have execute permissions for this file. Thus communication between runtimes can be controlled using simple filesystem permissions. Permissions checking is done during the `bttp` handshake. Note that it is possible for messages to be sent in one direction in a `bttp` connection but not in the other. In this situation replies are permitted but unsolicited messages are not. An important trade-off which was made when designing this model was that messages which are sent between actors in the same runtime are not subject to any authorization checks. This was done for two reasons: performance and security. By eliminating authorization checks messages can be more efficiently delivered between actors in the same process, which helps to reduce the performance penalty of the actor runtime over directly using a `tokio::Task`. Security is enhanced by this decision because it forces the user to separate actors with different security requirements into different operating system processes, which ensures all of the process isolation machinery in the operating system will be used to isolate them.

2.4 Actor Ownership

As in other actor systems, it is convenient to represent resources in Blocktree using actors. This allows the same security model used to control communication between actors to be used for controlling access to resources, and for resources to be shared by many actors. For instance, a Point-to-Point Protocol connection could be owned by an actor. This actor could forward traffic delivered to it in messages over this connection. The set of actors which are able to access the connection is controlled by setting the filesystem permissions on the file for the runtime executing the actor owning the connection.

The concept of ownership in programming languages is very useful for ensuring that resources are properly released when the object using them dies. Because actors are used for encapsulating resources in Blocktree, a similar system of ownership is employed. An actor is initially owned by the actor that spawned it. It can only have a single owner, but the owner can grant ownership to another actor. An actor is not allowed to own itself, though it may be owned by the runtime. When the owner of an actor returns, the actor is sent a message instructing it to return. If it does not return after a timeout, it is interrupted. This is the opposite of how supervision trees work in Erlang. Instead of the parent receiving a message when the child returns, the child receives a message when the parent returns. Service providers spawned by the runtime are owned by it. They continue running until the runtime chooses to reclaim their resources, which can happen because they are idle or the runtime is overloaded. Note that ownership is not limited to a single runtime, so distributed resources can be managed by owning actors in many different runtimes. In this case the connection to the remote runtime serves as a proxy for the remote owner, so the owned actors will be ordered to return if this connection dies or the remote owner dies.

While the actor runtime can be a convenient way of implementing new systems, a backwards compatibility mechanism is needed to allow existing systems to operate in the context of Blocktree. Containers have become the standard unit of deployment for modern applications, which makes them both useful and straight-forward to support in Blocktree. To execute a container in the actor runtime, it must be owned by a supervising actor. This actor is responsible for starting the container and managing the container's kernel resources. Logically, it owns all such resources, including all spawned operating system processes. When the actor halts, all of these resources are destroyed. All network communication to the container is controlled by the supervising actor. The supervisor can be configured to bind container ports to host ports, as is commonly done today, but it can also be used to encapsulate traffic to and from the container in Blocktree messages. These messages are routed to other actors based on the configuration of the supervisor. This essentially creates a VPN for containers, ensuring

that regardless of how insecure their communication is, they will be safe to communicate over any network. This network encapsulation system could be used in other actors as well, allowing a lightweight and secure VPN system to built.

2.5 Runtime Discovery Over the Network

While it's possible to resolve runtime paths to network endpoints when the filesystem is available, another mechanism is needed to allow the filesystem service providers to be discovered. This is accomplished by allowing runtimes to query one another to learn of other runtimes. Because queries are intended to facilitate message delivery, the query fields and their semantics mirror those used for addressing messages:

1. `service` The path of the service whose providers are sought. Only runtimes with this service registered will be returned.

2. `scope` The filesystem path relative to which the query will be processed.

3. `rootward` Indicates if the query should search for runtimes from `scope` toward the root.

As long as at least one other runtime is known, a query can be issued to learn of more runtimes. A runtime which receives a query may not be able to answer it directly. If it cannot, it returns the endpoint of the next runtime to which the query should be sent.

In order to bootstrap the discovery processes, another mechanism is needed to find the first peer to query. There were several possibilities explored for doing this. One way is to use a blockchain to store this data. This idea may be worth revisiting in the future, but the author wanted to avoid the complexity of implementing a new proof of work blockchain. Instead, two independent mechanisms are used, one that can discover runtimes over the internet as long as their path is known, and another that can discover runtimes on the local network even when the discoverer doesn't know their paths.

When the path to a runtime is known, DNS is used to resolve SRV records using a fully qualified domain name (FQDN) derived from the path's root principal identifier. The SRV records are resolved using the name `_bttp._udp.<FQDN>`, where `<FQDN>` is the FQDN derived from the root principal's identifier. One SRV record may be created for each of the filesystem service providers in the root directory. Each record contains the UDP port and hostname where a runtime is listening. Every runtime is configured with a search domain that is used as a suffix in the FQDN. The leading labels in the FQDN are computed by base32 encoding the binary representation of the root principal's identifier. If the encoded string is longer than 63 bytes (the limit for each label in a hostname), it is separated into the fewest number of labels possible, working from left to right along the string. A dot followed by the search domain is concatenated onto the end of this string to form the FQDN. This method has the advantages of being simple to implement and allowing runtimes to discover each other over the internet. Implementing this system would be facilitated by hosting DNS servers in actors in the same runtimes as the root filesystem service providers. Then, records could be dynamically created which point to these runtimes. These runtimes would also need to be configured with static IP addresses, and the NS records for the search domain would need to point to them. It is also possible to build such a system without hosting DNS inside of Blocktree, by using a dynamic DNS service. The downside of using DNS is that it couples Blocktree with a centrally administered, albeit distributed, system.

Because this mechanism requires knowledge of the root principal of a domain to perform discovery, it will not work if a runtime doesn't know its own root principal because it's starting up for the first time and has no credentials. This runtime needs a way to discover other runtimes so it can connect to the filesystem and sector services. This issue is solved by using link-local multicast addressing to discover the runtimes on the same network as the discoverer. When a `bttp` server starts listening for unicast traffic, it also listens for UDP datagrams on port 50142 at addresses 224.0.0.142 and FE02::142, if the IPv4 or IPv6 networking stacks are available, respectively. If the host is attached to a dual-stack network, the server listens on both addresses. When a runtime is attempting to discover other runtimes, it sends out datagrams to these endpoints. Each `bttp` server replies with its unicast endpoint and filesystem path (as specified in its credentials). If the server is available at both IPv4 and IPv6 unicast addresses, it's at the server's discretion which address to respond with. It may even respond with an IPv4 endpoint to an IPv4 datagram, and IPv6 endpoint to an IPv6 datagram. Once a client has discovered the `bttp`

servers on its network, it can route messages to them, such as the provisioning requests which are used to obtain new credentials. Provisioning is described in the Cryptography section. Note that port 50142 is in the dynamic range, so it doesn't need to registered with the Internet Assigned Names and Numbers Authority (IANA). Both addresses 224.0.0.142 and FE02::142 are currently unassigned. but they will need to be registered with IANA if Blocktree is widely adopted.

To allow runtimes which are not permitted to execute the root directory to query for other runtimes, authorization logic which is specific to queries is needed. If a process is connected with credentials and the path in the credentials contains the scope of the query, the query is permitted. If a process is connected anonymously, its query will only be answered if the query scope and all of its parent directories, grant others the execute permission. Queries from authenticated clients can be authorized using only the information in the handshake and query, but anonymous queries require knowledge of filesystem permissions, some of which may not be known to the answering runtime. When authorizing an anonymous query, an answering runtime should check that that the execute permission is granted on all directories that it's responsible for storing. If all these checks pass, it should forward the querier to the next runtime as usual.

2.6 Protocol Contracts

To facilitate the creation of composable systems, a protocol contract checking system based on session types has been designed. This system models a communication protocol as a directed graph representing state transitions based on the types of received messages. The protocol author defines the states that the actors participating in the protocol can be in using Rust traits. These traits define handler methods for each message type the actor is expected to handle in that state. A top-level trait which represents the entire protocol is defined that contains the types of the initial state of every party in the protocol. A macro is used to generate the message handling loop for each of the parties to the protocol, as well as enums to represent all possible states that the parties can be in and the messages that they exchange. The generated code is responsible for ensuring that errors are handled when a message of an unexpected type is received, eliminating the need for ad-hoc error handling code to be written by application developers.

Let's explore how this system can be used to build a simple pub-sub communications protocol. In this protocol, there will be a server which handles `Pub` and `Sub` messages, and there will be a client which sends `Sub` when it starts up and processes the resulting `Pub` messages as they arrive. The state-transition graph for the protocol is shown in figure 1. The solid edges indicate state transitions and are labeled with the message type which triggered the transition. The dashed edges indicate message delivery and are labeled with the type of the message being delivered. Although `Runtime` is not the state of any actor in the system, it is included in the graph because it is the logical sender of the `Activate` and `Pub` messages. `Activate` is delivered by the runtime to pass a reference to the runtime and provide the actor's `Uuid`. `Pub` messages are dispatched by actors outside the graph and are routed to actors in the `Listening` state by the runtime. Note that the runtime itself doesn't have any notion of the state of an actor, it just delivers messaging using the rules described previously. Only an actor can tell whether a message is expected or not given its current state. Each of the actor states are modeled by the following Rust traits:

```
pub struct ClientInit {
  type AfterActivate: Subed;
  type Fut: Future<Output = Result<Self::AfterActivate>>;
  fn handle_activate(self, msg: Activate) -> Self::Fut;
}

pub struct Subed {
  type AfterPub: Subed;
  type Fut: Future<Output = Result<Self::AfterPub>>;
  fn handle_pub(self, msg: Envelope<Pub>) -> Self::Fut;
}

pub struct ServerInit {
```

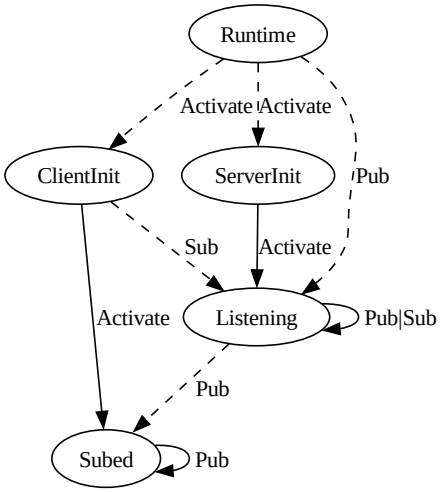

Figure 1: The state-transition graph for a simple pub-sub protocol.

```
  type AfterActivate: Listening;
  type Fut: Future<Output = Result<Self::AfterActivate>>;
  fn handle_activate(self, msg: Activate) -> Self::Fut;
}

pub struct Listening {
  type AfterSub: Listening;
  type SubFut: Future<Output = Result<Self::AfterSub>>;
  fn handle_sub(self, msg: Envelope<Sub>) -> Self::SubFut;

  type AfterPub: Listening;
  type PubFut: Future<Output = Result<Self::AfterPub>>;
  fn handle_pub(self, msg: Envelope<Pub>) -> Self::PubFut;
}
```

The definition of `Activate` is as follows:

```
pub struct Activate {
  rt: &'static Runtime,
  act_id: Uuid,
}
```

A static reference can be given to a runtime because a runtime is required to live for the entire lifetime of a process. This allows simple references to be passed around, avoiding the complexity of lifetimes and the overhead of reference counting. The `Envelope` type is a wrapper around a message which contains information about who sent it and a method that can be used to send a reply. In general a new actor state, represented by a new type, can be returned by a messaging handling method. The protocol itself is represented by the trait:

```
pub trait PubSubProtocol {
  type Server: ServerInit;
  type Client: ClientInit;
```

```
}
```

By modeling this protocol independently of any implementation of it, we allow for many different interoperable implementations to be created. We can also isolate bugs in these implementations because unexpected or malformed messages are checked for by the generated code.

2.7 Future Work

Currently, the actor runtime only supports actors implemented in Rust. A WebAssembly (Wasm) plugin system is planned which allows actors to be implemented in any language which can be compiled to Wasm. This work is blocked pending the standardization of the WebAssembly Component Model, which promises to provide an interface definition language which will allow type safe actors to be defined in many different languages. Once Wasm support is added, it will make sense to use the filesystem to distribute compiled actor modules, as the strong integrity protection it provides make it an ideal way to securely distribute software.

Any computer system of even moderate complexity needs an interface for viewing and controlling the state of the system. The modern cross-platform way to accomplish this is by creating a web GUI. Blocktree will include a service called `btconsole` which provides such an interface. It will be used to view and modify the filesystem, modify runtime attributes, and even register new services. The aim is to provide an interface which makes complicated network management tasks simple, and make networks more secure by providing a single pane of glass which shows their configuration.

3 Filesystem

The responsibility for serving data in Blocktree is shared between the filesystem and sector services. Most actors will access the filesystem through the filesystem service, which provides a high-level interface that takes care of the cryptographic operations necessary to read and write files. The filesystem service relies on the sector service for actually persisting data. The individual sectors which make up a file are read from and written to the sector service, which stores them in the local filesystem of the computer on which it's running. A sector is the atomic unit of data storage and the sector service only supports reading and writing entire sectors at once. File actors spawned by the filesystem service buffer reads and writes until there is enough data to fill a sector. Because cryptographic operations are only performed on full sectors, the cost of providing these protections is amortized over the size of the sector. Thus there is tradeoff between latency and throughput when selecting the sector size: a smaller size means less latency but a larger one enables more throughput.

3.1 The Sector Service

A file has a single metadata sector, a Merkle sector, and zero or more data sectors. The sector size of a file can be specified when it's created, but cannot be changed later. Every data sector contains the ciphertext of the number of bytes equal to the sector size, but the metadata and Merkle sectors contain a variable amount of data. The metadata sector contains all of the filesystem metadata associated with the file. In addition to the usual metadata present in any Unix filesystem, cryptographic information necessary to verify and decrypt the contents of the file is also stored. The Merkle sector of a file contains a Merkle tree over the data sectors of a file. The hash function used by this tree can be configured at file creation, but cannot be changed later.

When sector service providers are contained in the same directory they connect to each other to form a consensus cluster. This cluster is identified by a `u64` called the cluster's *generation*. Every file is identified by a pair of `u64`, its generation and its inode. The sectors within a file are identified by an enum which specifies which type they are, and in the case of data sectors, their 0-based index.

```
pub enum SectorKind {
  Meta,
  Merkle,
  Data(u64),
}
```

The byte offset in the plaintext of the file at which each data sector begins can be calculated by multiplying the sector's index by the sector size of the file. The `SectorId` type is used to identify a sector.

```
pub enum SectorId {
  generation: u64,
  inode: u64,
  sector: SectorKind,
}
```

The sector service persists sectors in a directory in its local filesystem, with each sector stored in a different file. The scheme used to name these files involves security considerations, and is described in the next section. When a sector is updated, a new local file is created with a different name containing the new contents. Rather than deleting the old sector file, it is overwritten by the creation of a hardlink to the new file, and the name that was used to create the new file is unlinked. This method ensures that the sector file is updated in one atomic operation. The sector service also uses the local filesystem to persist the replicated log it uses for Raft. This file serves as a journal of sector operations.

Communication with the sector service is done by passing it messages of type `SectorMsg`.

```
pub struct SectorMsg {
  id: SectorId,
  op: SectorOperation,
}

pub enum SectorOperation {
  Read,
  Write(WriteOperation),
}

pub enum WriteOperation {
  Meta(Box<FileMeta>),
  Data {
    meta: Box<FileMeta>,
    contents: Vec<u8>,
  }
}
```

Here `FileMeta` is the type used to store metadata for files. Note that updated metadata is required to be sent when a sector's contents are modified, because it contains updated integrity information.

A generation of sector service providers uses the Raft protocol to synchronize the state of the sectors it stores. The message passing interface of the runtime enables this implementation and the sector service's requirements were important considerations when designing this interface. The system currently replicates all data to each of the service providers in the cluster. Additional replication methods are planned as future enhancements (e.g. erasure encoding and distribution via consistent hashing), allowing for different tradeoffs between data durability and storage utilization.

The creation of a new generation of the sector service is accomplished with several steps. First, a new directory is created to contain the generation. Next, one or more runtimes are provisioned in this directory, using a procedure which is described in the next section. Provisioning produces files for each of the runtimes stored in the new directory. The sector service provider in each of the runtimes uses the filesystem service (which connects to the parent generation) to find the other runtimes hosting the sector service in the directory and messages them to establish a fully-connected cluster. Finally, the service provider which is elected leader contacts the generation in the root directory and requests a new generation number. Once this number is known, it is stored in the superblock for the generation, which is the file identified by the new generation number and inode 2. The superblock is not contained in any directory and cannot be accessed outside the sector service. The superblock also keeps track of the next inode to assign to a new file.

To prevent malicious actors from writing invalid data, the sector service must cryptographically verify all write messages. The process it uses to do this involves several steps:

1. The certificate chain in the metadata that was sent in the write message is validated. It is considered valid if it ends with a certificate signed by the root principal and the paths in the certificates are correctly nested, indicating valid delegation of write authority at every step.

2. Using the last public key in the certificate chain, the signature in the metadata is validated. This signature covers all of the fields in the metadata.

3. The new sector contents in the write message are hashed using the digest function configured for the file and the resulting hash is used to update the file's Merkle tree in its Merkle sector.

4. The root of the Merkle tree is compared with the integrity value in the file's metadata. The write message is considered valid if and only if there is a match.

This same logic is used by file actors to verify the data they read from the sector service, except they don't modify the Merkle tree during verification, they just compare computed hashes to those contained in the nodes on the path from the sector's leaf node to the root. Only once a write message is validated is it shared with the sector service provider's peers in its generation. Although the data in a file is encrypted, it is still beneficial for security to prevent unauthorized principals from gaining access to its ciphertext. To prevent this, a sector service provider checks a file's metadata to verify that the requesting principal actually has a readcap (to be defined in the next section) for the file. This ensures that only principals that are authorized to read a file can access its sectors.

3.2 The Filesystem Service

The sector service is relied upon by the filesystem service to read and write sectors. Filesystem service providers communicate with the sector service to open files and perform filesystem operations. These providers spawn file actors that are responsible for verifying and decrypting the information contained in sectors. They use the credentials of the runtime they're hosted in to decrypt sector data using information contained in file metadata. File actors are also responsible for encrypting and integrity protecting data written to files. In order for these actors to produce a signature over the root of the file's Merkle tree, it maintains a copy of the tree in memory. This copy is read from the sector service when the file is opened. While this does mean duplicating data between the sector and filesystem services, this design was chosen to reduce the network traffic between the two services, as the entire Merkle tree does not need to be transmitted on every write. Encapsulating all cryptographic operations in the filesystem service and file actors allows the computer storing data to be different from the computer encrypting it. This approach allows client-side encryption to be done on more capable computers and low powered devices to delegate encryption to a storage server.

A major advantage of using file actors to access file data is that they can be accessed over the network from a different runtime as easily as they can be from the same runtime. One complication arising from this approach is that file actors must not outlive the actor which caused them to be spawned. This is handled in the filesystem service by making the actor who opened the file the owner of the file actor. When a file actor receives notification that its owner returned, it flushes any buffered data and returns.

Some of the information stored in metadata needs to be kept in plaintext to allow files to be verified and decrypted, but most of it is encrypted using the same key as the file's contents. The file's authorization attributes, its size, and its access times are all encrypted. The table storing the file's extended attributes (EAs) is also encrypted. Cache control information is included in this area as well. It specifies the number of seconds that a file may be cached as a u32. The filesystem service uses this information to evict sectors from its cache when they've been cached for longer than this threshold.

The filesystem service uses an `Authorizer` type to make authorization decisions. It passes this type the authorization attributes of the principal accessing the file, the attributes of the file, and the type of access (read, write, or execute). The `Authorizer` returns a boolean indicating if access is permitted or denied. These access control checks are performed for every message processed by the filesystem service, including opening a file. A file actor only responds to messages sent from its owner, which ensures that it can avoid the overhead of performing access control checks as these were carried out by the filesystem service when it was created. The file actor is

configured when it is spawned to allow read only, write only, or read write access to a file, depending on what type of access was requested by the actor opening the file.

3.3 Filesystem Event Publishing

Often when building distributed systems it's convenient to publish information about events to any interested party. To facilitate this pattern, the sector service allows actors to subscribe for notifications when a file is written to. The sector service maintains a list of actors which are currently subscribed, and when it commits a write to its local storage, it sends each of them a notification message identifying the sector written (but not the written data). By using different files to represent different events, a simple notification system can be built. Because the contents of a directory may be distributed over many different generations, this system does not support the recursive monitoring of directories. Although this system lacks the power of `inotify` in the Linux kernel, it does provides some of its benefits without incurring much of a performance overhead or implementation complexity. For example, this system can be used to implement streaming replication. This is done by subscribing to writes on all the files that are to be replicated, then reading new sectors as soon as notifications are received. These sectors can then be written into replica files in a different directory. This ensures that the contents of the replicas will be updated in near real-time.

3.4 External Access to the Filesystem

Being able to access the filesystem from actors allows a programmer to implement new applications using Blocktree, but there is an entire world of existing applications which only know how to access the local filesystem. To allow these applications to access Blocktree, a FUSE daemon called `btfuse` was written which allows a Blocktree directory to be mounted to a directory in the local filesystem. This daemon can directly access the sector files in a local directory, or it can connect over the network to filesystem or sector service provider. This FUSE daemon could be included in a system's initrd to allow it to mount its root filesystem from Blocktree, opening up many interesting possibilities for hosting machine images in Blocktree. A planned future enhancement is to develop a Blocktree filesystem driver which actually runs in kernel space. This would reduce the overhead associated with context switching back and forth from user space to kernel space, increasing the performance of the system.

3.5 Future Work

Because of the strong integrity protection afforded to sectors, it is possible for peer-to-peer distribution of sector data to be done securely. Implementing this mechanism is planned as a future enhancement to the system. The idea is to base the design on bit torrent, where the generation responsible for a file acts as a tracker for that file, and the file actors accessing it communicate with one another directly using the information provided by the sector service. This could allow the system to scale elastically to a much larger number of concurrent reads by reducing the load on the sector service.

4 Cryptography

This section describes the cryptographic mechanisms used to integrity and confidentiality protect files as well as procedures for obtaining credentials. These mechanisms are based on well-established cryptographic constructions.

4.1 Integrity Protection

A file is integrity protected by a digital signature over its metadata. The metadata contains an integrity field which contains the root node of the Merkle tree over the file's contents. This allows any sector in the file to be verified with a number of hash function invocations that is logarithmic in the size of the file. It also allows the sectors of a file to be verified in any order, enabling random access. The hash function used in the Merkle tree can be configured when the file is created. Currently, SHA-256 is the default, and SHA-512 is supported. A file's metadata also contains a certificate chain, and this chain is used to authenticate the signature over the metadata. In Blocktree, the certificate chain is referred to as a *writecap* because it grants the capability to write

to files. This term comes from Tahoe: The Least-Authority Filesystem [7]. The certificates in a valid writecap are ordered by their paths, the initial certificate contains the longest path, the path in each subsequent certificate must be a prefix of the one preceding it, and the final certificate must be signed by the root principal. These rules ensure that there is a valid delegation of write authority at every link in the chain, and that the authority is ultimately derived from the root principal specified by the absolute path of the file. By including all the information necessary to verify the integrity of a file in its metadata, it is possible for a requestor who only knows the path of a file to verify that the contents of the file are authentic.

4.2 Confidentiality Protection

Confidentiality protection of files is optional but when it is enabled, a file's sectors are individually encrypted using a symmetric cipher. The key to this cipher is randomly generated when a file is created. A different IV is generated for each sector by hashing the index of the sector with a randomly generated IV for the entire file. A file's key and IV are encrypted using the public keys of the principals to whom read access is allowed. The resulting ciphertext is referred to as a *readcap*, as it grants the capability to read the file. This term is also from Tahoe [7]. These readcaps are stored in a table in the file's metadata. Each entry in the table is identified by a byte string that is derived from the public key of the principal who owns the entry's readcap. The byte string is computed by calculating an HMAC of the the principal's public key. The HMAC is keyed with a randomly generated salt that is stored in the file's metadata. An identifier for the hash function that was used in the HMAC is included in the byte string so that it can be recomputed later. When the filesystem service accesses the file, it recomputes the HMAC using the salt, its public key, and the hash function specified in each entry of the table. It can then identify the entry which contains its readcap, or that such an entry does not exist. This mechanism was designed to prevent offline correlation attacks on the readcap table, as it's stored in plaintext in local filesystems. The file key and IV are also encrypted using the keys of the file's parents. Note that there may be multiple parents of a file because it may be hard linked to several directories. Each of the resulting ciphertexts is stored in another table in the file's metadata. The entries in this table are identified by an HMAC of the parent's generation and inode numbers, where the HMAC is keyed using the file's salt. By encrypting a file's key and IV using the key and IV of its parents, it is possible to traverse a directly tree using only a single public key decryption. The file where this traversal begins must contain a readcap owned by the accessing principal, but all subsequent accesses can be performed by decrypting the key and IV of a child using the key and IV of a parent. Not only does this allow traversals to use more efficient symmetric key cryptography, but it also means that it suffices to grant a readcap on a single directory in order to grant access to the entire tree rooted at that directory.

Because it is not possible to change the key used by a file after it's created, a file must be copied in order to rotate the key used to encrypt it. Similarly, revoking a readcap is accomplished by creating a copy of the file and adding all the readcaps from the original's metadata except for the one being revoked. While it's certainly possible to remove a readcap from the metadata table, this is not supported because the readcap holder may have used custom software to save the file's key and IV while it had access to them, so data written to the same file after revocation could potentially be decrypted by it. By forcing the user to create a new file, they are forced to re-encrypt the data using a fresh key and IV, which cannot be known to the principal whose readcap was revoked.

4.3 Obfuscation of Sector Files in the Local Filesystem

From an attacker's perspective, not every file in a domain is equally interesting. They may be particularly interested in reading the root directory, or they may have identified the inode of a file containing kompromat. To make offline identification of which files sectors in the local filesystem belong to, an obfuscation mechanism is used. This works by generating a random salt for each generation of the sector service, and storing it in the generation's superblock. It is hashed along with the inode and the sector ID to produce the name to use for the sector file in the local filesystem. These files are organized into different subdirectories according to the value of the first two digits in the hex encoding of the resulting hash, the same way git organizes object files. This simple method makes it more difficult for an attacker to identify the files each sector belongs to while still allowing the sector service efficient access.

4.4 Credential Storage and Provisioning

Processes need a way to securely store their credentials. They accomplish this by using a credential store, which is a type that implements the trait `CredStore`. A credential store provides methods for using a process's credentials to encrypt, decrypt, sign, and verify data, but it does not allow them to be exported. A credential store also provides a method for generating new root credentials. Because root credentials represent the root of trust for an entire domain, it must be possible to securely back them up from one credential store to another. Root credentials can also be used to perform cryptographic operations without exporting them. A password is set when the root credentials are generated, and this same password must be provided to use, export, and import them. When root credentials are exported from a credential store they are confidentiality protected using multiple layers of encryption. The outer layer is encryption by a symmetric key cipher whose key is derived from the password. The public storage key of the receiving credential store must also be provided when root credentials are exported. This public key is used to perform encrypt a randomly generated key and IV which is used to perform the inner encryption of the root credentials with a symmetric cipher, ensuring that only the intended credential store is able to import them. Currently there are two `CredStore` implementors in Blocktree, one which is used for testing and one which is more secure. The first is called `FileCredStore`, and it uses a file in the local filesystem to store credentials. A symmetric cipher is used to protect the root credentials, if they are stored, but it relies on the security of the underlying filesystem to protect the process credentials. For this reason it is not recommended for production use. The other credential store is called `TpmCredStore`, and it uses a Trusted Platform Module (TPM) 2.0 [6] to store credentials. The TPM is used to generate the process's credentials in such a way that they can never be exported from the TPM (this is a feature of TPM 2.0). A randomly generated cookie is needed to use these credentials. The cookie is stored in a file in the local filesystem which has its permissions set to prevent others from accessing it. Thus this type also relies on the security of the local filesystem, but an attacker would need to steal the TPM and this cookie in order to steal a process's credentials.

The term provisioning is used in Blocktree to refer to the process of acquiring credentials. A command line tool call `btprovision` is provided for provisioning credential stores. This tool can be used to generate new process or root credentials, create a certificate request using them, issue a new writecap, and finally to import the writecap. When setting up a new domain, `btprovision` can create a new sector storage directory in the local filesystem and write the new process's files to it. It's also capable of connecting to the filesystem service if it's already running.

While manual provisioning is necessary to bootstrap a domain, an automatic method is needed to make this process more ergonomic. When a runtime starts, it checks its configured credential store to find the writecap to use to authenticating to other runtimes. If one is not stored, the runtime can choose to request a writecap from the filesystem service. This is done by dispatching a message with `call` to the filesystem service without specifying a scope. Because the message doesn't contain a path, theres no root directory to begin discovery at. So, the runtime resorts to using link-local discovery to find other runtimes. If one is discovered, the runtime connects to it anonymously and sends it a writecap request. This request includes a copy of the runtime's public key and, optional, a path where the runtime would like to be located. This path is purely advisory, the filesystem service is free to place the runtime in any directory it sees fit. The filesystem service creates a new file for the new runtime containing its public key and marks it as pending. The reply to the runtime contains the path of the file created for it. The operators of the domain can then use `btconsole` or `btprovision` to view the request and approve it at their discretion. The approving operator uses their credentials and the public key in the new process's file to issue a new writecap and stores it in the file. Authorization attributes (e.g. UID and GID) are also assigned and written into the file. Note that a process's file is normally not writeable by the process itself, so as to prevent it from setting its own authorization attributes. Once these data have been written, the runtime can read its file to retrieve its new writecap, which it stores in its credential store for later use. The runtime can avoid polling its file for changes if it subscribes to write notifications. To communicate using its new authorization attributes, the runtime must break existing connections and reconnect using its new writecap. Note that this procedure requires the new runtime to be on the same LAN as a filesystem service provider.

The procedure for creating a new domain is straight-forward, and all the steps can be performed using `btprovision`.

1. Generate the root credentials for the new domain.
2. Generate the credentials for the first runtime.

3. Create a writecap request using the runtime credentials.

4. Approve the request using the root credentials.

5. Import the new writecap into the credential store of the first runtime.

The first runtime is configured to host the sector and filesystem services, so that subsequent runtimes will have access to the filesystem. After that, additional runtimes on the same LAN can be provisioned using the automatic process.

4.5 Access Control Examples

Up till now the focus has been on authentication and authorization of processes, but it bears discussing how user based access control can be accomplished with Blocktree. Because credentials are locked to the device on which they're created, a user will be associated with at least as many principals as they have devices. But, all of these principals can be configured to have the same authorization attributes (UID, GID, SELinux context, etc.), giving them the same permissions. It makes sense to provision all of the runtimes associated with a user in one place and the natural place is the user's home directory. Although every one of the user's processes needs to be provisioned, this is not a huge limitation because a single runtime can host many different actors, implementing many different applications. Managing the users in a domain is facilitated by putting their home directories in a single user directory for the domain. Runtimes hosting the sector service on storage servers could then be provisioned in this directory to provide the sector and filesystem services for the users' home directories. It would be at the administrator's discretion whether to enable client-side encryption. If they wanted to enable it, the principal of at least one of a user's runtimes would need to be issued a readcap for the user's home directory. This runtime could then directly access the sector service in the domain's user directory. By moving encryption onto the user's computer, load can be shed from the storage servers. Note that this setup does require all of the user's runtimes to be able to communicate with the runtime whose principal was issued the readcap.

To illustrate how Blocktree can be used to enable collaboration between enterprises, consider a situation where two companies wish to partner on the development of a product. To facilitate their collaboration, they want to have a way to securely share data. One of the companies is selected to host the data and accepts the cost and responsibility of serving it. The host company creates a directory which is used to store the data. The other company will connect to the filesystem service in the host company's domain to access data in the shared directory. Each of the principals in the other company which need to access the data are provisioned in the shared directory. During the provisioning process, assigns authorization attributes to these principals. Once the principals have their writecaps, they can access the data in the shared directory by sending messages to the filesystem service with the shared directory as the `scope` field and the `rootward` field set to true. This setup gives the hosting company a lot of control over the data. If the other company wishes to protect its investment, it should use one of its provisioned principals to subscribe to write events on the shared directory and all of its files, so that it can copy written sectors out of the host company's domain as soon as they're written. Although it's not possible to directly subscribe to writes on the contents of a directory, by monitoring a directory for changes, it's possible to begin monitoring files as soon as they're created.

5 Example Systems

This section contains examples of systems that could be built using Blocktree. The hope is to illustrate how this platform can be used to implement existing applications more easily and to make it possible to implement systems which are currently out of reach.

5.1 A distributed AI execution environment.

Neural networks are just vector-valued functions with vector inputs, albeit very complicated ones with potentially billions of parameters. But, just like any other computation, these functions can be conceptualized as computational graphs. Imagine that you have a set of computers equipped AI accelerator hardware and you have a neural network that is too large to be processed by any one of them. By partitioning the graph into small enough

subgraphs, we can break the network down into pieces which can be processed by each of the accelerators. The full network can be stitched together by passing messages between each of these pieces.

Let us consider how this could be accomplished with Blocktree. We begin by provisioning a runtime on each of the accelerator machines, each of which will have a new accelerator service registered. Messages will be sent to the accelerator service describing the computational graph to execute, as well as the name of the actor to which the output is to be sent. When such a message is received by an accelerator service provider, it spawns an actor which compiles its subgraph to a kernel for its accelerator and remembers the name of the actor to send its output to. An orchestrator actor will be responsible for partitioning the graph and sending these messages. Ownership of the actors spawned by the accelerator service is given to the orchestrator actor, ensuring that they will all be stopped when the orchestrator returns. When one of the spawned actors stops, it unloads the kernel from the accelerator's memory and returns it to its initial state. Note that the orchestrator actor must have execute permissions on each of the accelerator runtimes in order to send messages to them. The orchestrator dispatches messages to the accelerator service in reverse order of the flow of data in the computational graph, so that it can tell each service provider where its output should be sent. The actors responsible for the last layer in the computational graph send their output to the orchestrator. To begin the computation, the actors which are responsible for input are given the filesystem path of the input data. The orchestrator learns of the completion of the computation when it receives the output from final layer. It can then save these results to the filesystem and return, thus ensuring all resources are released. Because inference and training can both be modeled by computational graphs, this same procedure can be used for both.

5.2 A decentralized social media network.

One of the original motivations for designing Blocktree was to create a platform for a social network that puts users in full control of their data. In the opinion of the author, the only way to actually accomplish this is for users to host the data themselves. One might think it is possible to use client-side encryption to solve the privacy problem, but this does not solve the full problem. While it is true that good client-side encryption will prevent the service provider from reading the user's data, the user could still loose everything if the service provider goes out of business or simply decides to stop offering its service. Similarly, putting data in a federated system, such as Mastodon or Nostr, also puts the user at risk of loosing their data if the server they use is permanently shutdown. To have real control the user must host the data themselves. Then they decide how its encrypted, how its served, and whether it continues to exist.

Let's explore how Blocktree can be used to build a social media platform which provides this control. To participate in this network each user will need to setup their own domain by generating new root credentials and provisioning at least one runtime to host the social media service. A technical user could do this on their own hardware by reading the Blocktree documentation, but a non-technical user might choose to purchase a new router with Blocktree pre-installed. By connecting this router directly to their WAN, the user ensures that the services running on it will always have direct internet access. The user can access the `btconsole` web GUI via the router's WiFi interface to generate their root credentials and provision new runtimes on their network.

A basic function of any social network is keeping track of a user's contacts. This is accomplished by maintaining the contacts as files in a well-known directory in the user's domain. Each file in the directory is named using the user defined nickname for the contact and its contents include the root principal of the contact as well as any additional user defined attributes, such as physical address or telephone number. The root principal is used to route messages to the social media service in the contact's domain. When a user adds a new contact, a connection message is sent to it, which the contact can choose to accept or reject. If accepted, the contact creates an entry in its contacts directory for the user. The contact's social media service will then accept future direct messages from the user. When the user sends a direct message to the contact, its runtime dispatches the message to the social media service in the contact's domain. Once delivered, the contact's social media service stores the message in a directory for the user's correspondence, sort of like an mbox directory but where messages are sorted into directories based on sender rather than receiver.

Note that this procedure only works if a contact's root principal can be resolved using the search domain configured in the user's runtime. We can ensure this is the case by configuring the runtime to use a search domain that operates a Dynamic DNS (DDNS) service and by arranging with this service to create the correct records to resolve the root principal. The author intends to operate such a service to facilitate the use of Blocktree by

home users, but a more long-term solution is to implement a blockchain for resolving root principals. Only then would the system be fully decentralized.

Making public posts is accomplished by creating files in a directory with the HTML contents of the post. This file, the directory containing it, and all parents of it, would be configured to allow others to read, and in the case of directories, execute them. At least one runtime with the filesystem service registered would need to have the execute permission granted to others to allow anyone to access these files. When someone wanted to view the posts of another user, they would dispatch messages to the filesystem service in the other user's domain to read these files from the well-known posts directory.

Of course user's would not be using a file manager to interact with this social network, they would use their browsers as they do now. This web interface would be served by the social media service in their domain. A normal user who has a Blocktree enabled router would just type in a special hostname into their browser to open this interface. Because the router provides DNS services to their network, it can generate the appropriate records to ensure this name resolves to the address where the social media service is listening. The social media service would be responsible for sending messages to other users' domains to get their posts, and to the filesystem in their own domain to display the user's direct messages. All this file data would be used to populate the web interface. It's not hard to see how the same system could be used to serve any type of media: text, images, video, immersive 3D worlds. All of these can be stored in files in the filesystem, and so all of them are accessible to Blocktree actors.

One issue that must be addressed with this design is how it will scale to a large number of users accessing it at once. In other words, what happens if a user goes viral? Currently, the way to solve this would be to add more computers to the user's network which run the sector and filesystem services. This is not ideal as it means the user would need to buy more hardware. A better solution would be implement peer-to-peer distribution of sector data in the filesystem and sector services. This would reduce the load on the user's computers and allow their follows to share the posted data with each other. This work is planned as a future enhancement.

5.3 A smart lock.

The access control language provided by Blocktree's filesystem can be used for more than just authorizing access to data. To illustrate this point, consider a smart lock installed on the front door of a company's office building. When the company first got the lock they used NFC to configure it to connect to their WiFi network. The lock then used link-local runtime discovery to perform automatic provisioning. An IT administrator accessed `btconsole` to approve the provisioning request and position the lock in a specific directory in the company's domain. Permission to actuate the lock is granted if a principal has execute permission on the lock's file. To verify the physical presence of an employee, NFC is used for the authentication handshake. When an employee presses their NFC device, for instance their phone, to the lock, it generates a nonce and transmits it to the device. The device then signs the nonce using the credentials it used during provisioning in the company's domain. It transmits this signature to the lock along with the path to the principal's file in the domain. The lock then reads this file to obtain the principal's authorization attributes and its public key. It uses the public key to validate the signature presented by the device. If this is successful, it then checks the authorization attributes of the principal against the authorization attributes on its own file. If execute permissions are granted, the lock actuates, allowing the employee access. The administrators of the company's domain create a group specifically for controlling physical access to the building. All employees with physical access permission are added to this group, and the group is granted execute permission on the lock, rather than individual users.

5.4 A traditional three-tier web application.

While it's hoped that Blocktree will enable interesting and novel applications, it can also be used to build the kind of web applications that are common today. Suppose that we wish to build a three-tier web application. Let's explore how Blocktree could help.

First, we should consider which database to use. A traditional SQL database would be desirable, preferably one which is open source and not owned by a large corporation with dubious motivations. These constraints lead us to choose Postgres, but Postgres was not designed to run on Blocktree. However, Postgres does have a container image available on docker hub, we can create a service to run this container image in our domain. But

Postgres stores all of its data in the local filesystem of the machine it runs on. How can we ensure this does not become a single point of failure? First, we should create a directory in our domain to hold the Postgres cluster directory. Then we should procure at least three storage servers and provision runtimes hosted on each of them in this directory. The sector service is registered on each of the runtimes, so all the data stored in the directory will be replicated on each of the servers. Now, the Postgres service should be registered in one and only one of these runtimes, as Postgres requires exclusive access to its database cluster. `btfuse` will be used to mount the Postgres directory to a path in the local filesystem and the Postgres container will be configured to access it. We now have to decide how other parts of the system are going to communicate with Postgres. We could have the Postgres service setup port forwarding for the container, so that ordinary network connection can be used to talk to it. But we will have to setup TLS if we want this to be secure. The alternative is to use Blocktree as a VPN and proxy network communications in messages. This is accomplished by registering a proxy service in the same runtime as the Postgres service and configuring it to allow traffic it receives to pass to the Postgres container's IP address on TCP port 5432.

In a separate directory, several runtimes are provisioned which will host the webapp service. This service will use axum to serve the static assets to our site, including the Wasm modules which make up our frontend, as well as our site's backend. In order to do this, it will need to connect to the Postgres database. This is accomplished by registering the proxy service in each of the runtimes hosting the webapp service. The proxy service is configured to listen on TCP 127.0.0.1:5432 and forwards all traffic to the proxy service in the Postgres directory. The webapp can then use the `tokio-postgres` crate to establish a TCP connection to 127.0.0.1:5432 and it will end up talking to the containerized Postgres instance.

Although the data in our database is stored redundantly, we do still have a single point of failure in our system, namely the Postgres container. To handle this we can implement a failover service. It will work by calling the Postgres service with heartbeat messages. If too many of these timeout, we assume the service is dead and start a new instance one of the other runtimes in the Postgres directory. This new instance will have access to all the same data as the failed instance, including its journal file. Assuming it can complete any in progress transactions, the new service will come up after a brief delay and the system will recover.

5.5 A realtime geo-spacial environment.

If we are to believe science fiction, the natural evolution of human-computer interaction is the development of a persistent virtual world that we use to communicate, conduct business, and enjoy our leisure. This kind of system has been a dream for a long time, but as it's grown closer to becoming a reality, the popular consciousness has shifted against it. People are rightly horrified by the idea of giving control over their virtual worlds to the same social media company that has a track record of causing societal harm. But this technology does not need to be dystopian. If an open system can be built, which actually works, it can prevent the market from accepting a closed system designed to lock in user attention and monetize them relentlessly. These systems are the future, it's only a question of who will own them.

Let's explore how Blocktree can be used to build such a system. The world we're going to render will be a planet with a roughly spherical surface and a configurable radius ρ, which is a `u32` value whose units are meters. We'll use latitude (ϕ) and longitude (λ) in radians to describe the locations of points on the surface. Both ϕ and λ will take `f64` values. The elevation of a point will be given by h, which is the deviation from ρ. h is measured in meters and takes values in `i32`. So, the distance from the center of the planet to the point (ϕ, λ, h) is $\rho + h$.

The data describing how to render a planet consists of its terrain mesh, terrain textures, and the objects on its surface. This could represent a very large amount of data for a planet with realistic terrain populated by many structures. To facilitate sharding this information over many different servers, the planet is broken into disjoint regions, each of which is stored in its own directory. A single top-level directory represents the entire planet, and contains a manifest describing it. This manifest specifies the planet's name, its radius, its rotational period, the size limit of its regions in MB, as well as any other global attributes. This top-level directory also contains the texture for the sky box to render the view of space from the planet. In the future it may be interesting to explore the creation of more dynamic cosmic environments, but a simple sky box has the advantage of being efficient. The data in a planet is recursively broken into the fewest number of regions such that the amount of data in each regions is less than the configured threshold. When a regions grows too large it is broken into four new regions by cutting it along the centerline parallel to the ϕ axis, and the one parallel to the λ axis. In other words,

it is divided in half north to south and east to west. The four new regions are stored in four subdirectories of the original region's directory named SE, SW, NE, and NW, depending on their location relative to the original region. The data in the old region is then moved into the appropriate directories. The directory tree of a planet essentially forms a quadtree, albeit one which is built up progressively.

In the leaf directories of this tree the actual data for a region are stored in two files, one which describes the terrain and the other which describes objects. It's expected that the terrain will rarely be modified, but that the objects may change regularly. The terrain file contains the mesh vertices in the region as well as its textures. It is organized as an R-tree to allow for efficient spacial queries based on player location. The region's objects file is also organized as an R-tree. It contains all of the graphical data for the objects to be rendered in the region, including meshes, textures, and shaders.

The creation of a shared virtual world must involve players collaboratively building persistent structures. This is allowed in a controlled way by defining plot objects. A plot is like a symbolic link, it points to a file whose contents contain the data used to render the plot. This mechanisms allows the owner of the planet to delegate a specific area on its surface to another player by creating a plot defining that area and pointing it to a file owned by the other player. The other player can then write meshes, textures, and shaders into this file to describe the contents of the plot. If the other player wishes to collaborate with others on the construction, they can grant write access on the file to a third party. This is not unlike the ownership of land in the real world.

To facilitate viewing the planet from many distances, each interior node in the planet's directory tree contains a reduced level of detail (LOD) version of the terrain contained in it. For example, the top-level directory contains the lowest LOD mesh and textures for the terrain. This LOD would be suitable for rendering the planet as a globe on a shelf, or as it would appear from a high orbit. By traversing the directory tree, the LOD can be increased as the player travels closer to the surface. This system assists with rendering an animation where the player appears to approach and land upon the planet's surface.

By dividing the planet's data into different leaf directories, it becomes possible to provision computers running the sector and filesystem services in each of them. This divides the storage and bandwidth requirements for serving the planet over this set of servers. In addition to serving these data, another service is needed to keep track of player positions and execute game logic. Game clients address their messages using the directory of the region their player is located in, and set `rootward` to true. These messages are delivered to the closest game server to the region the player is in, which may be located in the region's directory or higher up the tree. When a player approaches the border of a region, its game client begins dispatching messages to the adjacent directories as well.

6 Conclusion

There have been many attempts to create a distributed Unix over the years. Time has shown that this is a difficult problem, but time has not diminished its importance. IT systems are more complex now than ever, with many layers of abstraction that have built up over time. As users, we've suffered greatly from systems which were never designed to be secure on the hostile internet that exists today. Security has been bolted onto these systems (HTTPS, STARTTLS, DNSSEC) in a backwards compatible way, which results in weakened protections. What's worse, the entire trust model of the web relies on the ludicrous idea that there is a distinguished group of certificate authorities who have the power to secure our communications. We need to take a different approach. Data should be certified by its path, it must always be transported between processes in an authenticated manner, and user code should never have to care how this is accomplished.

The typical internet user stores all their important data in the cloud with third-party service providers. They do this because of the convenience of being able to access this information from anywhere, and because of the perceived safety in having a large internet company look after it for them. This convenience comes at the price of putting users at the mercy of these companies. Take email for example, a service which is near universally used for account recovery, password reset, and login verification. If a service provided decided to stop providing a user access to their email, the user would be effectively cut off from their online life, which is effectively their entire life. There is no technical reason for things to be this way. Blocktree allows users to host their own services in their own domain. If we can make setting up an email or social media server as simple as clicking a button in a web GUI, there will be no convenience advantage to using cloud services. If more users begin hosting their own services, the internet will become more distributed, which will make it more resistent to disruption and

censorship.

Cloud computing has also driven changes in the way businesses acquire computing resources. It's common for startups to rent all of their computing resources from one large cloud provider and there are compelling economic and technical reasons to do this. But, as a firm grows they often experience growing pains as their cloud bills grow with them. If the firm has not developed their software with a multi-cloud, or hybrid approach in mind, they may face the prospect of major changes in order to bring their application on-prem or to a rival cloud. By developing their application on Blocktree, businesses have a single platform to target which can run on rented computers in the cloud just as easily as servers in their own data center. This ensures the choice to rent or buy can be made on a purely economic basis. Blocktree is not the only system that provides flexibility. The portability of containers is one of the reasons they've become so popular, and they have their place, but they are a lower-level abstraction which require developers to solve many of the problems that Blocktree could handle for them.

Ransomware attacks and data breaches are embarrassingly common. There are many reasons for this, from the reliance on passwords for authentication, to the complexity of the software supply chain, but it's clear that as IT professionals we need to do more to keep the systems under our protection safe. Blocktree helps us do this by solving many of the difficult problems involved with securing communication on a hostile network. It takes a true zero-trust approach, ensuring that all communications between processes is authenticated using public key cryptography. Data at rest is also secured with encryption and integrity protection. No security system can prevent all attacks, but by putting these mechanisms together in an easy to use platform, we can advance the status quo for secure computing.

When Unix was first developed in the 1970's, its authors could not have foreseen the applications that would be enabled by their system. Although there have been many different kinds of Unices over the years, the core programming model, built around the filesystem, has remained since the beginning. It's a testament to the importance of this abstraction that it's persisted for so long. No designer can foresee all the ways that their abstractions will be used, but they can try to build them in such a way that as much choice is left to the user as possible. By making the actor model, and messaging passing, the core of Blocktree, it is hoped that low overhead communication between distributed components can be achieved. By using this system to provide a global distributed filesystem, it is hoped that the interoperable sharing of data can be achieved. And by using protocol contracts to constrain actor communication, it is hoped that structure and safety can bring order to distributed computation. While it's possible to see some of the applications that can be built from these abstractions, their composability and the creativity of developers will lead to systems that cannot be foreseen.

References

[1] Joe Armstrong. "Making reliable distributed systems in the presence of software errors". PhD thesis. Royal Institute of Technology, Stockholm, Sweden, 2003.

[2] Phil Bernstein et al. *Orleans: Distributed Virtual Actors for Programmability and Scalability*. Tech. rep. MSR-TR-2014-41. Mar. 2014. URL: https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/.

[3] Adam Langley et al. "The QUIC Transport Protocol: Design and Internet-Scale Deployment". In: 2017.

[4] David Mazières and M. Frans Kaashoek. "Escaping the Evils of Centralized Control with Self-Certifying Pathnames". In: *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*. EW 8. Sintra, Portugal: Association for Computing Machinery, 1998, pp. 118–125. ISBN: 9781450373173. DOI: 10.1145/319195.319213. URL: https://doi.org/10.1145/319195.319213.

[5] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, June 2014, pp. 305–319. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[6] *TPM 2.0 Library*. URL: https://trustedcomputinggroup.org/resource/tpm-library-specification/.

[7] Zooko Wilcox-O'Hearn and Brian Warner. *Tahoe – The Least-Authority Filesystem*. Cryptology ePrint Archive, Paper 2012/524. https://eprint.iacr.org/2012/524. 2012. URL: https://eprint.iacr.org/2012/524.