Pleiades v3 Technologies

Sienna Lloyd [sienna@linux.com](mailto:sienna@linux.com)

Pleiades is a collection of different types, layers, and aspects of technologies that enable it to operate successfully. Right now, Pleiades is at v2 - it’s already gone several early rewrites after validating assumptions, design patterns, etc. This doc describes technologies that are being targeted for the v3 rewrite.

The technologies listed below are grouped by category but not necessarily any specific order.

# Programming Language

Pleiades v1 and v2 both use Go. Go is a very powerful language and allowed Pleiades to go through many quick iterations of technology. It’s concurrency model made it easy to design Pleiades’ monolithic but modular architecture, and enabled high-throughput on most workloads. However, Go has also been exceedingly limiting due to it’s memory management model.

The value of Go for Pleiades was Sienna’s familiarity with the ecosystem, a large and diverse CNCF ecosystem to pull libraries from, well-respected CNCF vendors with excellent reference libraries, and a vibrant community. However, Go’s memory management model has been extremely limiting when it comes to performance. Due to Go’s GC and automatic memory management, it’s incredibly difficult to determine where objects are allocated, what their lifecycle is, and nil pointer dereferences are difficult to debug in a massive monolith. Go’s parametric generics are simply too basic and don’t allow for covariance, and can’t realistically be used effectively. Go also uses Plan9 assembly, which is impossible to write due to a near complete lack of documentation. Go is useful for infrastructure applications, but not low-level infrastructure.

Pleiades v3 is targeting a complete rewrite in Rust. Rust’s memory management, lifetimes, generics, and general typing system are substantially stronger than Go’s, and it supports intrinsics through LLVM. Rust’s memory management model guarantees faster performance due to memory ownership and lifetimes, and it’s threading model is much more robust than Go’s goroutines. This does create an extensive overhead as large swaths of the existing Pleiades v2 code base comes from 3rd parties, and several of the more important subsystems will need to be either completely rewritten from scratch, ported, or alternatives found. Overall Rust’s performance characteristics, memory management, and LLVM integration make it a much more suitable language for Pleiades v3.

# Networking

The core networking stack of Pleiades v3 will be based off QUIC as it provides faster connection times and zero-blocking streams. Pleiades v2 is currently a mixture of gRPC and varying TCP implementations.

## QUIC

QUIC is the underlying networking technology. It’s based off Google’s SPDY protocol, and was ratified by IETF with [RFC 9000](https://www.rfc-editor.org/rfc/rfc9000.html). QUIC provides 0-RTT handshakes, multiple streams per connection, full TLS connection security, ordered bidi streaming, passive latency monitoring, and connection migrations. It is incredibly performant and is the underlying technology of HTTP/3.

The reference implementation that Pleiades will likely use for Pleiades v3 is [quiche](https://github.com/cloudflare/quiche). Cloudflare’s networking backbone is some of the best in the industry, and their Rust implementation is the most used. For more information on QUIC, see Cloudflare’s [landing page](https://cloudflare-quic.com/)

## RPC & Messaging

While gRPC provides useful RPC functionality, it is HTTP-based, and being at the mercy of the ecosystem is miserable. Pleiades v2 uses it and it is a major lesson-learned for the project. gRPC is incredibly slow, and the ecosystem is driven by *largest consumer needs*, so HTTP/3 won’t come for years, and even then it’s still HTTP. Pleiades v2 also embeds [NATS](https://nats.io).

Pleiades v3 will no longer use gRPC in any part of it’s code, but two variations of QUIC. It’s expected that there will be two layers of RPC-style networking: one for the very low-level raft, gossip, and kvstore subsystems; and the other for higher-order application functionality such as messaging, queuing, and pubsub. Being at the mercy of the gRPC ecosystem is hellish, miserable, and generally a great way to let your technology rot. That being said, protocol buffers are very powerful, and useful for encoding and framing, so those will stay.

### RPC

At the lowest level, Pleiades v3 will implement a custom QUIC-based protocol with protocol buffers using magic bytes and protobuf framing architectures to determine framing, routing, and message passing. While QUIC supports ordered bidi streaming on a per-stream basis, due to short-term complexity, its likely that each stream handler will maintain two streams per protocol type for ease of simplicity. The use of magic bytes is primarily to annotate and notify changes in configurations, routing, etc., and will be limited as protobufs are fully framed.

### Messaging

For higher-order application messaging needs, Pleiades v2 currently uses embedded NATS as the queuing and pubsub messaging provider. NATS is incredibly powerful, incredibly heavy, and also written in Go. Going outside of Go, the only major message queuing platform that seems to be a good fit is [ZeroMQ](https://zeromq.org/). zmq is a very powerful solution in C++, and there are several Rust bindings for it, and one [full-Rust implementation](https://github.com/zeromq/zmq.rs). Sienna has an [open issue](https://github.com/zeromq/zmq.rs/issues/181) to ask about the project status, but she may just fork it and maintain it either way. The only limitation of zmq (bindings or native) is that currently there are no QUIC socket implementations.

## Network Interfacing

Pleiades v3 will likely define a standard set of network traits that each library can implement to leverage the networking library in an RPC-adjacent manner. This is dependent on Rust’s memory model, and whether or not it’s a good design pattern.

For the protocol buffer implementation, right now [quick-protobuf](https://crates.io/crates/quick-protobuf) is attractive because it’s low-level and uses clone-on-write. It also doesn’t require protoc or other external tools, which is extremely attractive.

# Clustering & Automation

Pleiades v2 uses powerful libraries from well-respected tech companies to manage clustering, membership, and other varying things for it’s autonomy. However, nearly all these technologies are in Go and must be ported with non-trivial modifications, mostly to networking.

## Gossip

Pleiades v2 targeted [Serf](https://github.com/hashicorp/serf) for it’s internal gossiping structure. The value of Serf was it’s mixture of [SWIM](https://www.cs.cornell.edu/projects/Quicksilver/public_pdfs/SWIM.pdf) and the [Vivaldi network tomography](https://sites.cs.ucsb.edu/~ravenben/classes/276/papers/vivaldi-sigcomm04.pdf) system. However, Serf is written in Go and by Hashicorp, who are relicensing all of their new versions and products with BSL moving foward. So something has to be done.

The value of SWIM + network tomography is it’s infectious gossiping, clustering, and location awareness. This allows Pleiades to loosely model a force-directed graph, where changes are rolled out through loosely connected nodes, and the network tomography allows for location-based clustering without needing to define locations. Pleiades v3 will keep SWIM the Vivaldi network tomography functionality for it’s gossiping patterns. This is an essential architecture piece which allows Pleiades v3 to remain an automated constellation mesh database (re: fully-autonomous system). For the curious, Hashicorp published a [configurable convergence simulator](https://www.serf.io/docs/internals/simulator.html) for SWIM; it’s worth a look-see if you have performance questions.

However, there is a substantial bit of work which needs to be done to port Hashicorp’s reference implementation and extensions. Hashicorp’s [Lifeguard extensions](https://arxiv.org/abs/1707.00788) to SWIM is relatively minor, but the Vivaldi implementation contains several extensions from the [Network Coordinates in the Wild](https://www.usenix.org/legacy/events/nsdi07/tech/full_papers/ledlie/ledlie_html/index_save.html) USENIX paper and IBM Research’s [Euclidean Embedding](https://dominoweb.draco.res.ibm.com/492D147FCCEA752C8525768F00535D8A.html) paper. While these extensions can be trusted due to long-term production usage, they also make it harder when referencing the papers.

The reference SWIM implementation (with Lifeguard) is called [memberlist](https://github.com/hashicorp/memberlist). memberlist bundles it’s own networking and custom packet implementation for the protocol, but the Pleiades v3 port can’t have either. The network messages need to be protobufs, and the networking implementation will need to support the QUIC-based stack that Pleiades will use. Otherwise, the rest of the functionality should be a relatively straightforward port using Rust’s stdlib.

The Vivaldi implementation sits in the Serf [tree](https://github.com/hashicorp/serf/tree/master/coordinate), and is fairly straightforward. Most of the structs can be converted to protobufs for external interfacing, but otherwise this library is fairly straightforward.

## Clocks

Pleiades v1 and v2 do not implement clocks, and this is a major design flaw. During prototyping, none of the major versions ever made it far enough to need a clock. However, for ranges to work with atomic transactions, Pleiades v3 will need accurate clocks. The OS will always sync with the system clock, but Pleiades v3 will port CockroachDB’s [hybrid logical clock (HLC) implementation](https://github.com/cockroachdb/cockroach/blob/master/pkg/util/hlc/doc.go) as it’s implementation. Lamport clocks have too much skew for a tightly-knit system like Pleiades, and CockroachDB’s HLC implementation is stable in production. This port should be very straightforward.

## Lifecycle Automation

Pleiades v1 contained no dependency injection, but Pleiades v2 uses Uber’s [fx](https://github.com/uber-go/fx) framework. This gives a baseline DI framework that’s good enough to handle startup and shutdown events. However, fx is overcomplicated and not worth porting, so Pleiades v3 can use whatever is most popular in the Rust ecosystem. DI is imperative as Pleiades is a modular monolith, and there needs to be both control of background services and also DI for things like service-to-service clients, network handlers, etc.

Pleiades v1 and v2 did not contain lifecycle workflows, but Pleiades v3 will contain a small workflow engine to keep the internal lifecycle events manageable. As each node is fully autonomous in the constellation, the ability to handle complex workflows in for CREs is important. This will help the project maintainers advance the constellation’s internals without completely rewriting large swaths of internal logic every time there’s a logic change to a CRE workflow. Daniel Gerlag’s [Workflow Core](https://github.com/danielgerlag/workflow-core) is an excellent embeddable workflow engine in C# and is the reference implementation for Pleiades v3 workflow engine. Not all features or functionality will be needed, so the port will primarily be just the workflow engine and enough netcode to keep it controllable and observable.

## Config Automation

Pleiades v1 used Steve Francia’s [viper](https://github.com/spf13/viper) and [cobra](https://github.com/spf13/cobra) libraries, whereas Pleiades v2 only used viper and a custom port of Mitchell Hashimoto’s [CLI library](https://github.com/mitchellh/cli). Viper is a very powerful but overcomplicated configuration management library that provides the core configuration, but it is like trying to use a shotgun on work that needs a scalpel. For the CLI and configuration library, whatever is both popular for Rust and also minimal will likely be the correct decisions to integrate. Ideally, these would be external dependencies instead of ported code.

# Storage

Storage in Pleiades v1 was an absolute mess (hey, it was a prototype lol), and Pleiades v2’s storage layer is better, but still not great. Pleiades v3 aims to fix that by having a single, unified storage layer that provides local, shard, and global storage opportunities to all consumers (with caveats). Both Pleiades v1 and v2 are built on Raft, but Pleiades v2 uses multi-raft.

## Disk Storage

Pleiades v1 and v2 both use [bbolt](https://github.com/etcd-io/bbolt), which is a Go-based port of Howard Chu’s LMDB. Bbolt is a great embedded database for some workloads, but it is not enough for Pleiades. Bbolt is based on b+tree indexing, which is useful for lightning fast reads, but horrible at writes. To support more complex workloads, Pleiades v3 needs to use a Log-Structured-Merge-Tree (LSM) database, where reads and writes are a bit more balanced.

The initial target replacement for bbolt was CockroachDB’s [Pebble](https://github.com/cockroachdb/pebble/blob/master/docs/rocksdb.md), which is a [customized port](https://github.com/cockroachdb/pebble/blob/master/docs/rocksdb.md) of Facebook’s [RocksDB](https://github.com/facebook/rocksdb). RocksDB is based off Google’s [LevelDB](https://github.com/google/leveldb). However, as Pleiades v3 is no longer targeting Go, Pebble is no longer a good fit for use. TiKV’s storage engine uses RocksDB, which is mind-blowingly optimized for modern storage hardware, and Pleiades v3 will likely use TiKV’s [RocksDB bindings](https://github.com/tikv/rust-rocksdb). As all databases, at their lowest levels, are just disk-based hash maps, using RocksDB is totally normal and saves a bunch of effort.

Pleiades v3 will use a single instance of RocksDB per node to store local, shard, and global data.

## Raft

Pleiades v1 used Hashicorp’s raft implementation and Pleiades v2 uses a multi-raft library called [dragonboat](https://github.com/lni/dragonboat). Dragonboat is *impressively performant*, but it’s in Go and by a very discreet maintainer who uses the handle lni. With Pleiades v3 being Rust-based, there are two real options for Rust-based raft: port dragonboat (not ideal) or fork & modify TiKV’s [raft-rs](https://github.com/tikv/raft-rs) (not ideal). Realistically, those are the two options, and neither are ideal. Porting dragonboat will be heavily error prone because it contains **extensive** Go-specific performance modifications, and that’s not really ideal. However, raft-rs uses the [prost](https://github.com/tokio-rs/prost) protobuf library, which is heavy and slow compared to quick-protobuf, and it includes it’s own networking.

Pleiades v2’s raft architecture was heavily influenced by dragonboat, and dragonboat has more bells and whistles than raft-rs. TiKV’s internal multi-raft implementation, [raftstore-v2](https://github.com/tikv/tikv/tree/master/components/raftstore-v2), is complex and built on top of raft-rs using TiKV’s [placement drivers](https://github.com/tikv/pd) and [regions](https://tikv.github.io/tikv-dev-guide/understanding-tikv/scalability/region.html) (their version of CockroachDB’s range keys). Pleiades v3 can’t really consume raftstore-v2 as-is because we don’t support regions, but range keys, and our multi-raft architectures are *wildly different*.

Sienna’s assertion is that we should fork & modify raft-rs to implement our networking changes and use the fork for now. Ideally, we’ll submit the modifications back to raft-rs, but due to our custom network protocol, it’s unlikely the modification will be welcome. So long as our networking modifications are minimal and isolated, it should be fairly easy to pull in patches from upstream raft-rs as needed. As raft is nearly 10 years old now, it’s unlikely to go through major changes, so this is a fairly safe decision, it just comes with extensive maintenance burdens.

## Ranges (re: sharding)

Pleiades v1 used no sharding (but also didn’t use multi-raft), and Pleiades v2 uses various hashing algorithms for sharding. Architecturally, Pleiades v3 will leverage CockroachDB’s range key architecture in conjunction with it’s gossip fabric. As this required a full rewrite regardless of the Rust migration, there are minimal changes here.

## Transactions & MVCC

Pleiades v2 contained atomic transaction support, but Pleiades v3 does not currently have any transaction support planned. It is possible that v3 will contain atomic transaction support, but it is not guaranteed. RocksDB does contain pessimistic and optimistic transaction support, so it is possible that atomic transactions will continue to exist in Pleiades v3.

Pleiades v2 contained support for atomic MVCC operations through bbolt, but Pleiades v3 will not. RocksDB contains WAL and two-phase commit functionality, which allows for similar operations, but is not quite the same. Continued MVCC support is unplanned for Pleiades v3.

# Administration

Pleiades v2 had no administration layer, and administration for Pleiades v3 was planned via direct integration with SWIM. Pleiades v2 contained several fabric CLI commands, and Pleiades v3 will contain similar constructs for the time being.

## Authentication

Neither versions of Pleiades contains authentication as they were architectural prototypes, but Pleiades v3 will lay the core foundation required for fine-grained authorization. Originally, Pleiades v3 was going to bundle [OpenFGA](https://openfga.dev) for authentication, but that has changed now that v3 will be a full bottom-up rewrite. Aside from some nice-to-have TLS functionality to make things easier to work with, authentication is going to be put on hold until at least v3.1, but possibly later.