The Internals of Veilid

A New Distributed Application Framework

Christien ‘DilDog’ Rioux
Katelyn ‘Medus4’ Bowden

www.veilid.com
What is Veilid?
The Veilid Mission

We exist to develop, distribute, and maintain a privacy focused communication platform and protocol for the purposes of defending human and civil rights.

“Fight for the things that you care about, but do it in a way that will lead others to join you.” - Ruth Bader Ginsburg
Others have come before us...

**Tor**
Privacy-oriented Networking

**IPFS**
Distributed Data Storage

Other Efforts (Check them out too!)
- NOSTR - social media specific, still ‘federated’ relay vs client
- Scuttlebutt.nz - social publication system, no ip privacy
- Holepunch.to - similar app framework concept, no ip privacy
- Not mentioned - a lot of ‘web3’ ‘dApps’ that require buying some ‘coin’
Veilid is an open-source peer-to-peer mobile-first networked application framework.

Veilid is conceptually similar to IPFS + Tor, but faster and designed from the ground-up to provide all services over a privately routed network.

Veilid enables development of fully-distributed applications without a 'blockchain' or a 'transactional layer' at their base.

Veilid can be included as part of user-facing applications or run as a 'headless node' for power users who wish to help build the network.
Veilid Design Goals

Security First
Written In Rust
Memory and type-safety

Runs Everywhere
Linux, Mac, Windows
Android, iOS
and browser WASM

Standard Protocols
UDP
TCP
Websockets

All In Network
No External Services
Avoid DNS
No STUN/TURN

Privacy Focused
Strong cryptography
IP privacy is built-in
Nodes != Identity

Resilient
Low Latency
High Node Churn
Switching Networks
Building a community of applications

Not everything needs to be centralized

Stop being dependent on corporate systems
Networking
Nodes

All Veilid applications running veilid-core are 'nodes' and they are all equal in the eyes of the network.

No nodes are 'special'

All nodes help each other out, regardless of the application hosting them.

Nodes are only limited by the resources they bring and the configuration of the network they are on.

Applications directly linking in veilid-core

Linux, Mac, Windows, Android, iOS, and Web Apps for Everyone.
FFI and WASM Bindings for Flutter / Dart.
Rust applications can directly use veilid-core.
Native bindings for other languages are welcome!

Headless nodes running veilid-server

Linux, Mac and Windows for 'power users'.
Can be controlled via JSON API for simpler apps.
Python development via veilid-python package.
Admins and devs can use veilid-cli to control server.

<table>
<thead>
<tr>
<th>Node Id</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLD0:6-FYr7P7pB7B8U-JntxH5S7qTCM03LhVpQ17dJuwL8M</td>
<td>UDP:170.64.186.46:5150</td>
</tr>
<tr>
<td>VLD0:7DyNMc134kHrLhX0m6YQoAdf3VAv3YvF1V7ceNQkc</td>
<td>UDP:161.35.164.16:5150</td>
</tr>
<tr>
<td>VLD0:ccxhF4LuhFXMA-878uF306F59vo01lF0D-656C10e</td>
<td>UDP:159.69.163.27:5150</td>
</tr>
<tr>
<td>VLD0:nsOYjM6F7cVh0yUX50s2fS4tFCYy538yh/hk43JMc</td>
<td>UDP:159.233.237.80:5150</td>
</tr>
<tr>
<td>VLD0:0xJf3s_AEXvFQDvlypYNMs3nMqBEdYlKpyy0k0ucl</td>
<td>UDP:157.230.215.0:5150</td>
</tr>
<tr>
<td>VLD0:pntFXY8y0C0UrI13etUXs82t3SFBhjd7Y-Ttiz0w</td>
<td>UDP:170.64.186.46:5150</td>
</tr>
</tbody>
</table>

Connected to [::1]:5959 | TUNs: 18.75kB/s Up: 6.38kB/s No Tunnels
Protocols

Low level protocols supported by Veilid are kept simple, to minimize complications.

Everything uses framed RPC operations up to 64KB in size.

Protocol support is extensible and may add WebRTC and other specialized protocols in the future.

DNS is only used one time during ‘bootstrap’ and not required.

SSL is optional and only for HTTPS.

Websockets for Veilid Webapps.

UDP

Fast, unsequenced, unreliable datagrams
Chunked into MTU-sized pieces and reassembled by Veilid
Support for out-of-order delivery
No retransmission or acknowledgment

TCP

Sequenced, reliable streams
In-line framing
All the usual TCP guarantees

Websockets

Sequenced, reliable streams
Support for HTTP and HTTPS delivery
All nodes speak Websockets
Browsers can directly contact any other node on the network.
Every node has a 256-bit public key 'node id'. Nodes arrange their routing table with a 'distance' metric. Routing tables are 'buckets' like Kademlia DHT.
Bootstrapping

Bootstrap nodes aren’t ‘special’. Any node can bootstrap a Veilid network. Networks can be ‘keyed’ to keep nodes off that don’t have the key. You can join the ‘big Veilid network’ or make your own isolated network.

Ask Bootstraps To ‘Find Self’
A single initial DNS TXT record request returns some bootstrap nodes that are known to exist. Those are asked to return nodes that are ‘close’ to your own node.

Public Address Detection
Nodes are often behind various forms of NAT. Validating one’s own public ‘Dial Info’ is essential for publishing one’s Node Info and answering Find Node requests.

Peer Minimum Refresh
Nodes in your routing table are asked to return nodes that are near you as well. Finding nodes close to your own is always harder than finding nodes far away, so we focus on that with our requests.

Network Class Detection
Determining NAT type and what mechanisms can be used to achieve connectivity. Direct connection techniques like reverse connections and UDP hole punching may be inappropriate for some network classes.

Relay Configuration
Low-capability network classes may require the use of Inbound or Outbound relays in order to achieve reachability. Nodes help each other out to the best of their ability and incur no penalty for not being able to assist other nodes.

Public Address Detection
Nodes are often behind various forms of NAT. Validating one’s own public ‘Dial Info’ is essential for publishing one’s Node Info and answering Find Node requests.

Ping Validation
Nodes come and go, change address, and are unreliable. Checking routing table nodes for proof-of-life is done with exponential backoff. Nodes are removed from the routing table on a LIFO basis.

Veilid
All devices are welcome and treated fairly

You can use the public Veilid Network or build your own

Nodes help each other like mutual aid for connectivity
Cryptography
Strong, appropriate, cryptography choices are essential to the functioning of Veilid.

Veilid provides applications guarantees about how data is handled on the wire and at rest.

Cryptosystems were chosen that work well together and provide a balance of speed and cryptographic hardness.

**Authentication** is Ed25519

Elliptic curve 25519 was chosen to provide public/private key authentication and signing capabilities.

**Key Exchange** is x25519

Curve 25519 has a DH function that allows nodes to generate a symmetric key to communicate privately.

**Encryption** is XChaCha20-Poly1305

ChaCha20 with a 192-bit extended nonce is a fast authenticated stream cipher with associated data (AEAD).

**Message Digest** is BLAKE3

BLAKE3 is an extremely fast cryptographic hash that is highly parallelizable and as strong as SHA3-256 and over 17 times faster.

**Key Derivation** is Argon2

Password hash generation should be slow and resistant to GPU attacks. Argon2 was the winner of the 2015 Password Hashing Competition.
Upgrading

Nothing lasts forever

And cryptography is no exception. As computing power improves and cryptographic attacks evolve, weaknesses in cryptosystems are inevitable.

Veilid has ensured that upgrading to newer cryptosystems is streamlined and minimally invasive to app developers, and handled transparently at the node level.

Multiple Routing Tables

Because changing cryptosystems changes node ids, there will be different distance measurements between nodes, necessitating a separate routing table per cryptosystem. We support this today.

Typed Keys

Cryptographic keys, signatures, and hashes are all tagged with their cryptosystem to ensure that we know exactly how they were generated and how they should be used and persisted.

Migration Support

Reading persisted data will automatically use the correct cryptosystem and will default to always writing it back using the newest/best cryptosystem. This allows for data to be easily migrated just by reading it and writing it back to storage.

Simultaneous Cryptosystems

While transitioning cryptosystems, nodes can respond to other nodes using either the old system or the new one, or both.
Secure Storage

Device-level secret storage APIs are available for all platforms

Encrypted table store APIs are exposed to applications to make safe data storage easy

Device data keys can also be password protected

Apps never need to write anything to disk unencrypted

Protected Store

Device-level Secret Storage

- MacOS / iOS Keychain
- Android Keystore
- Windows Protected Storage
- Linux Secret Service

- New Rust Crate: keyring-manager

Table Store

Encrypted Key-Value Database

- SQLITE on Native
- IndexedDB in Browser

Device Key can be protected from backup dumping attacks

- New Rust Crate: keyvaluedb

Record Store

Distributed Hash Table Storage

- Encrypted + Authenticated
- Subkey support
- LRU distributed cache
- Per-key multi-writer schemas

Block Store

Content-addressable Data Distribution

- Take What You Give model
- Connect and share cloud storage
- Bittorrent-like sharding

“COMING SOON”
**On The Wire**

Data is encrypted at rest and on the wire
Everything is authenticated and encrypted between nodes
All node information is signed

<table>
<thead>
<tr>
<th>All Protocols Same Encryption</th>
<th>Encrypted And Signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each low-level protocol uses the same message and receipt encapsulation. No protocol is special and all protocols offer the same safety guarantees.</td>
<td>Messages between nodes are signed by the sender and encrypted for only the receiver. Messages can be relayed without decryption and authentication covers the entire contents including headers.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Everything Is Timestamped</th>
<th>Node Information Is Signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Envelopes include timestamps and unique nonces and reject old or replayed messages.</td>
<td>When a node publishes routing table entries they are signed. No node can lie about another node's dial info, capabilities, availability, or replay old node info when newer info is available.</td>
</tr>
</tbody>
</table>
Everything is end-to-end encrypted

All storage is encrypted at rest

Your data is protected even if you lose your device
RPC Protocol
**RPC Schema**

Strong, appropriate, cryptography choices are essential to the functioning of Veilid.

Veilid provides applications guarantees about how data is handled on the wire and at rest.

Cryptosystems were chosen that work well together and provide a balance of speed and cryptographic hardness.

- **Schema Language is Cap’n Proto**
  
  Cap’n Proto is designed for deserialization speed and schema evolution. Flexible and well supported in Rust.

- **RPC is fully in-schema and documented**
  
  Both ‘Question/Answer’ and ‘Statement’ RPC modes are supported. All schema fields are documented.

- **RPC fully supports Private Routing**
  
  All private routing structures are expressed in the RPC schema itself, no magic encrypted blobs.

- **Schema Evolution is built-in**
  
  Fields can be added and removed with full backward and forward compatibility. New features won’t break older Veilid nodes.

- **RPC Schema is cryptography-independent**
  
  As cryptosystems change, the language spoken by Veilid nodes remains the same.
FindNodeQ

Finding Nodes from other nodes’ routing tables is a functional primitive for Veilid networking.

A node that sends a FindNodeQ RPC question will receive a FindNodeA RPC answer within the allowed RPC latency.

The question asks a node to find nodes ‘close’ to a key in hash space that meet some capability criteria.

The answer returns a list of nodes and their Signed Node Info.
Finding Nodes from other nodes’ routing tables is a functional primitive for Veilid networking.

A node that sends a FindNodeQ RPC question will receive a FindNodeA RPC answer within the allowed RPC latency.

The question asks a node to find nodes ‘close’ to a key in hash space that meet some capability criteria.

The answer returns a list of nodes and their Signed Node Info.
Distributed Hash Tables are a way of storing data in records that have keys that are close to nodes in the network.

**DHT Is Just ‘Search’**

It may look complicated, but all of the DHT algorithms out there are just ‘search’ algorithms. Finding data that is stored on some node somewhere out there.

**Improving Search**

We built a better DHT by making both search and data locality more relevant. Veilid synchronizes popular data when nodes come and go from the network.
Veilid DHT is built using GetValue and SetValue RPC operations. Nodes can opt out of DHT storage if they do not want to participate.

Veilid DHT records have schemas that define subkeys that are individually addressable and can have multiple writers.

DHT record subkeys have sequence numbers and are eventually consistent across multiple writes and background synchronizations.
The DHT gives you full control over your data

Our DHT is not based on a blockchain or a coin

Popular data becomes more available automatically
Private Routing
Veilid Routes are a combination of source and destination private routing. Because no node can trust any other node to pick the whole route, both source and destination must participate.
Compiled Routes

Private Routes are published as a ‘private destination’ and Safety Routes are allocated locally and combined together with a Private Route to form a Compiled Route.

A's Private Route

B's Safety Route To A

ENC=Authenticated Encryption With Nonce
DEC=Authenticated Decryption With Nonce
PK=Public Key
SK=Secret Key
DH=Diffie-Hellman Symmetric Key
ADDR=IP Address
X=Nonce
### Secure Envelopes

Each node hop only knows about the next one. This is similar to onion routing, but assumes that the source is fully in control of the Safety Route and the destination is fully in control of the Private Route.

- **Envelope from B to N5**
- **Envelope from N5 to N6**
- **Envelope from N6 to N7**
- **Envelope from N7 to N8**
- **Envelope from N8 to A**
Toward The Future

Private routing is a balance of performance and security. Applications can make use of higher node hop counts if they desire. Future private routing advancements will be transparent to users.

Per-Hop Payload Keying

Ensuring that there is nothing common between packets at each hop will reduce the risk of mass data collection being able to deanonymize routes.

Elimination of Hop Counting

Currently the protocol keeps an internal hop count that is not necessary. Efforts should be made to ensure that individual nodes don’t know how far along in a route they are.

Simplify Directionality

Routes are currently bidirectional, but are allocated directionally. We may be able to simplify our allocation mechanism by enforcing bidirectionality. Bidirectional routes are faster, but directional routes could provide more anonymity.

Hop Caching

Route hop NodeInfo could be cached to save on-the-wire size as well as speed things up.

Increasing Hop Count

Currently the default is one hop chosen by the Safety Route, and one hop chosen by the Private Route, which leads to three hops total once compiled.

It may be important to increase hop count to 2 for users with critical safety needs and to protect from nation-state-level deanonymization where appropriate.

Existing research (on Tor) suggests that our existing hop count should be sufficient and provide comparable anonymity, but this should be revisited.
IP Privacy means your location is safe too

Users don’t have to do anything to use it

No IP address means no tracking, collection, or correlation
Veilid is written in Rust
Crates are published and you can use them today!
Power User Quick Start

Just read the README.md and clone the repository from GitLab and get started right away!

```
echo "deb [arch=amd64 signed-by=/usr/share/keyrings/veilid-packages-keyring.gpg] http://packages.veilid.net/apt stable main" > /etc/apt/sources.list.d/veilid.list

yum-config-manager --add-repo https://packages.veilid.net/rpm/veilid-rpm-test-repo.repo
```

git clone --recursive-submodules git@gitlab.com:veilid/veilid.git

### Veilid Server

In order to run the `veilid-server` locally:

```
cd ./veilid-server
cargo run
```

In order to see what options are available:

```
cargo run -- --help
```

### Veilid CLI

In order to connect to your local `veilid-server`:

```
cd ./veilid-cli
cargo run
```

Similar to `veilid-server`, you may see CLI options by typing:

```
cargo run -- --help
```
Veilid has first-class FFI+JS Plugin support for Dart/Flutter and example code to get you started!
Veilid has an easy no-compile way to get started learning the Veilid API with Python
How You Can Help
Veilid is an open-source initiative, designed and implemented in the open. Come join our team and contribute to its growth! Be part of this!

**Coders And Hackers**

We can use more low-level programmers and protocol experts. Platform experts. We want this system to work well for everyone and be a strong foundation for general computing and application development.

**App Developers**

You can get started writing a Veilid app today! Got a game idea? Want to port something from a centralized system to a decentralized one? Let’s make this happen!

**Usability Experts**

We want to make sure that Veilid and Veilid apps are accessible to everyone. Everyone should be able to make use of Veilid without even realizing they’re doing it.

**Open Source + Governance**

Open source projects deserve to be managed in the open too. We’ve got an open RFC process for our design and an MPL-2.0 license that ensures that free and commercial entities can contribute safely and legally.
Find Us Online!

Web: www.veilid.com

Twitter: @veilidnetwork

Mastodon: @veilidnetwork

Discord: veilid.com/discord

GitLab: gitlab.com/veilid
See It Live Tonight

Release Party at 8pm!