@push.rocks/smartvpn

Documentation for @push.rocks/smartvpn

- readme.md for @push.rocks/smartvpn
- changelog.md for @push.rocks/smartvpn

readme.md for @push.rocks/smartvpn

A high-performance VPN solution with a **TypeScript control plane** and a **Rust data plane daemon**. Enterprise-ready client authentication, triple transport support (WebSocket + QUIC + WireGuard), and a typed hub API for managing clients from code.

□□ **Noise IK** mutual authentication — per-client X25519 keypairs, server-side registry □□ **Triple transport**: WebSocket (Cloudflare-friendly), raw **QUIC** (datagrams), and **WireGuard** (standard protocol) □□ **ACL engine** — deny-overrides-allow IP filtering, aligned with SmartProxy conventions □□ **PROXY protocol v2** — real client IPs behind reverse proxies (HAProxy, SmartProxy, Cloudflare Spectrum) □□ **Per-transport metrics**: active clients and total connections broken down by websocket, QUIC, and WireGuard □□ **Hub API**: one `createClient()` call generates keys, assigns IP, returns both SmartVPN + WireGuard configs □□ **Real-time telemetry**: RTT, jitter, loss ratio, link health — all via typed APIs □□ **Unified forwarding pipeline**: all transports share the same engine — TUN (kernel), userspace NAT (no root), or testing mode □□ **Destination routing policy**: force-target, block, or allow traffic per destination with nftables integration ⚡ **Handshake-driven WireGuard state**: peers appear as "connected" only after a successful WireGuard handshake, and auto-disconnect on idle timeout

Issue Reporting and Security

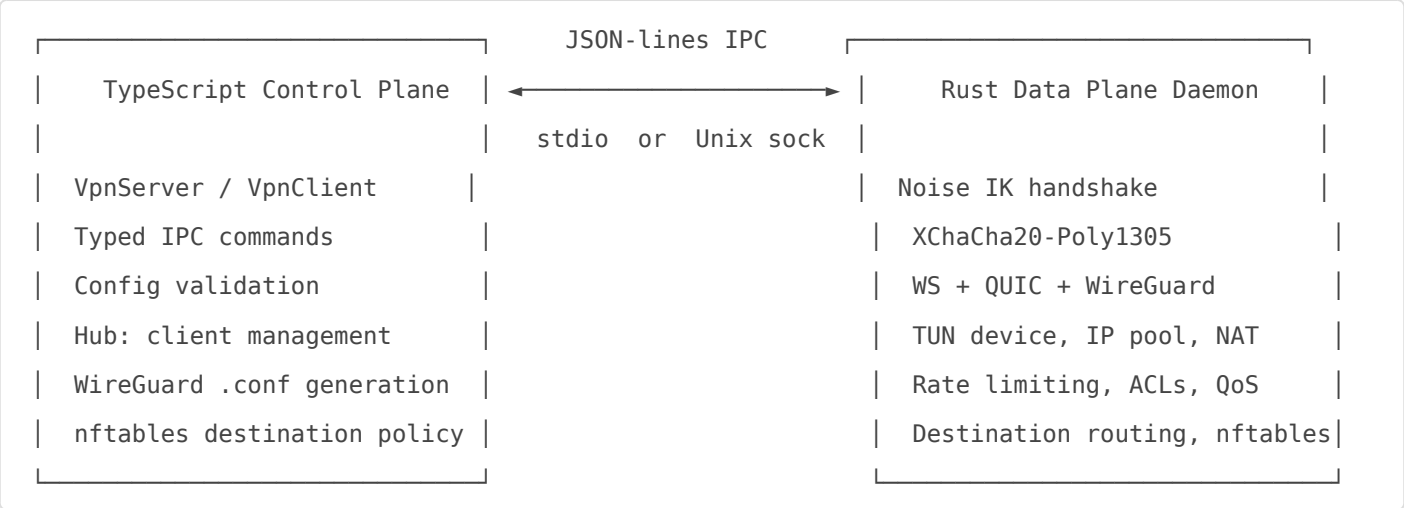
For reporting bugs, issues, or security vulnerabilities, please visit community.foss.global/. This is the central community hub for all issue reporting. Developers who sign and comply with our contribution agreement and go through identification can also get a code.foss.global/ account to submit Pull Requests directly.

Install □□

```
pnpm install @push.rocks/smartvpn
# or
npm install @push.rocks/smartvpn
```

The package ships with pre-compiled Rust binaries for **linux/amd64** and **linux/arm64**. No Rust toolchain required at runtime.

Architecture



Split-plane design — TypeScript handles orchestration, config, and DX; Rust handles every hot-path byte with zero-copy async I/O (tokio, mimalloc).

IPC Transport Modes

The bridge between TypeScript and Rust supports two transport modes:

Mode	Use Case	How It Works
stdio	Development, testing	Spawns the Rust daemon as a child process, communicates over stdin/stdout
socket	Production	Connects to an already-running daemon via Unix domain socket, with optional auto-reconnect

```
// Development: spawn the daemon
const server = new VpnServer({ transport: { transport: 'stdio' } });

// Production: connect to running daemon
const server = new VpnServer({
  transport: {
    transport: 'socket',
```

```
    socketPath: '/var/run/smartvpn.sock',
    autoReconnect: true,
    reconnectBaseDelayMs: 100,
    reconnectMaxDelayMs: 5000,
    maxReconnectAttempts: 10,
  },
});
```

Quick Start □□

1. Start a VPN Server (Hub)

```
import { VpnServer } from '@push.rocks/smartvpn';

const server = new VpnServer({ transport: { transport: 'stdio' } });
await server.start({
  listenAddr: '0.0.0.0:443',
  privateKey: '<server-noise-private-key-base64>',
  publicKey: '<server-noise-public-key-base64>',
  subnet: '10.8.0.0/24',
  transportMode: 'all',       // WebSocket + QUIC + WireGuard simultaneously (default)
  forwardingMode: 'tun',      // 'tun' (kernel), 'socket' (userspace NAT), or 'testing'
  wgPrivateKey: '<server-wg-private-key-base64>', // required for WireGuard transport
  enableNat: true,
  dns: ['1.1.1.1', '8.8.8.8'],
});
```

2. Create a Client (One Call = Everything)

```
const bundle = await server.createClient({
  clientId: 'alice-laptop',
  serverDefinedClientTags: ['engineering'],  // trusted tags for access control
  security: {
    destinationAllowList: ['10.0.0.0/8'],        // can only reach internal network
    destinationBlockList: ['10.0.0.99'],           // except this host
```

```
    rateLimit: { bytesPerSec: 10_000_000, burstBytes: 20_000_000 },
  },
});

// bundle.smartvpnConfig  → typed IVpnClientConfig, ready to use
// bundle.wireguardConfig → standard WireGuard .conf string
// bundle.secrets         → { noisePrivateKey, wgPrivateKey } — shown ONCE
```

3. Connect a Client

```
import { VpnClient } from '@push.rocks/smartvpn';

const client = new VpnClient({ transport: { transport: 'stdio' } });
await client.start();

const { assignedIp } = await client.connect(bundle.smartvpnConfig);
console.log(`Connected! VPN IP: ${assignedIp}`);
```

Features

Enterprise Authentication (Noise IK)

Every client authenticates with a **Noise IK handshake** (`Noise_IK_25519_ChaChaPoly_BLAKE2s`). The server verifies the client's static public key against its registry — unauthorized clients are rejected before any data flows.

- Per-client X25519 keypair generated server-side
- Client registry with enable/disable, expiry, tags
- Key rotation with `rotateClientKey()` — generates new keys, returns fresh config bundle, disconnects old session

Triple Transport

Transport	Protocol	Best For
WebSocket	TLS over TCP	Firewall-friendly, Cloudflare compatible

Transport	Protocol	Best For
QUIC	UDP (via quinn)	Low latency, datagram support for IP packets
WireGuard	UDP (via boringtun)	Standard WG clients (iOS, Android, wg-quick)

The server runs **all three simultaneously** by default with `transportMode: 'all'`. All transports share the same unified forwarding pipeline (`ForwardingEngine`), IP pool, client registry, and stats — so WireGuard peers get the same userspace NAT, rate limiting, and monitoring as WS/QUIC clients. Clients auto-negotiate with `transport: 'auto'` (tries QUIC first, falls back to WS).

□□ Per-Transport Metrics

Server statistics include per-transport breakdowns so you can see exactly how many clients use each protocol:

```
const stats = await server.getStatistics();

// Aggregate
console.log(stats.activeClients);       // total connected clients
console.log(stats.totalConnections);    // total connections since start

// Per-transport active clients
console.log(stats.activeClientsWebsocket);   // currently connected via WS
console.log(stats.activeClientsQuic);        // currently connected via QUIC
console.log(stats.activeClientsWireguard);   // currently connected via WireGuard

// Per-transport total connections
console.log(stats.totalConnectionsWebsocket);
console.log(stats.totalConnectionsQuic);
console.log(stats.totalConnectionsWireguard);
```

WireGuard connection state is handshake-driven — registered WireGuard peers do NOT appear as "connected" until their first successful WireGuard handshake completes. They automatically disconnect after 180 seconds of inactivity or when boringtun reports `ConnectionExpired`. This matches how WebSocket/QUIC clients behave: they appear on connection and disappear on disconnect.

□□ ACL Engine (SmartProxy-Aligned)

Security policies per client, using the same `ipAllowList` / `ipBlockList` naming convention as `@push.rocks/smartproxy`:

```
security: {
  ipAllowList: ['192.168.1.0/24'],          // source IPs allowed to connect
  ipBlockList: ['192.168.1.100'],            // deny overrides allow
  destinationAllowList: ['10.0.0.0/8'],      // VPN destinations permitted
  destinationBlockList: ['10.0.0.99'],        // deny overrides allow
  maxConnections: 5,
  rateLimit: { bytesPerSec: 1_000_000, burstBytes: 2_000_000 },
}
```

Supports exact IPs, CIDR, wildcards (`192.168.1.*`), and ranges (`1.1.1.1-1.1.1.100`).

□□ PROXY Protocol v2

When the VPN server sits behind a reverse proxy, enable PROXY protocol v2 to receive the **real client IP** instead of the proxy's address. This makes `ipAllowList` / `ipBlockList` ACLs work correctly through load balancers.

```
await server.start({
  // ... other config ...
  proxyProtocol: true,                          // parse PP v2 headers on WS connections
  connectionIpBlockList: ['198.51.100.0/24'],     // server-wide block list (pre-handshake)
});
```

Two-phase ACL with real IPs:

Phase	When	What Happens
Pre-handshake	After TCP accept	Server-level `connectionIpBlockList` rejects known-bad IPs — zero crypto cost
Post-handshake	After Noise IK identifies client	Per-client `ipAllowList` / `ipBlockList` checked against real source IP

- Parses the PP v2 binary header from raw TCP before WebSocket upgrade
- 5-second timeout protects against stalling attacks
- LOCAL command (proxy health checks) handled gracefully
- IPv4 and IPv6 addresses supported
- `remoteAddr` field on `IVpnClientInfo` exposes the real client IP for monitoring
- **Security**: must be `false` (default) when accepting direct connections — only enable behind a trusted proxy

□□ Destination Routing Policy

Control where decrypted VPN client traffic goes — force it to a specific target, block it, or allow it through. Evaluated per-packet before per-client ACLs.

```
await server.start({
  // ...
  forwardingMode: 'socket', // userspace NAT mode
  destinationPolicy: {
    default: 'forceTarget',                    // redirect all traffic to a target
    target: '127.0.0.1',                       // target IP for 'forceTarget' mode
    allowList: ['10.0.0.0/8'],                 // these destinations pass through directly
    blockList: ['10.0.0.99'],                  // always blocked (deny overrides allow)
  },
});
```

Policy modes:

Mode	Behavior
`'forceTarget'`	Rewrites destination IP to `target` — funnels all traffic through a single endpoint
`'block'`	Drops all traffic not explicitly in `allowList`
`'allow'`	Passes all traffic through (default, backward compatible)

In **TUN mode**, destination policies are enforced via **nftables** rules (using `@push.rocks/smartnftables`). A 60-second health check automatically re-applies rules if they're removed externally.

In **socket mode**, the policy is evaluated in the userspace NAT engine before per-client ACLs.

□□ Socket Forward Proxy Protocol

When using `forwardingMode: 'socket'` (userspace NAT), you can prepend **PROXY protocol v2 headers** on outbound TCP connections. This conveys the VPN client's tunnel IP as the source address to downstream services (e.g., SmartProxy):

```
await server.start({
  // ...
  forwardingMode: 'socket',
  socketForwardProxyProtocol: true,  // downstream sees VPN client IP, not 127.0.0.1
```

```
});
```

□□ Packet Forwarding Modes

SmartVPN supports three forwarding modes, configurable per-server and per-client:

Mode	Flag	Description	Root Required
TUN	`'tun'`	Kernel TUN device — real packet forwarding with system routing	□ Yes
Userspace NAT	`'socket'`	Userspace TCP/UDP proxy via `connect(2)` — no TUN, no root needed	□ No
Testing	`'testing'`	Monitoring only — packets are counted but not forwarded	□ No

```
// Server with userspace NAT (no root required)
await server.start({
  // ...
  forwardingMode: 'socket',
  enableNat: true,
});

// Client with TUN device
const { assignedIp } = await client.connect({
  // ...
  forwardingMode: 'tun',
});
```

The userspace NAT mode extracts destination IP/port from IP packets, opens a real socket to the destination, and relays data — supporting both TCP streams and UDP datagrams without requiring `CAP_NET_ADMIN` or root privileges.

□□ Telemetry & QoS

- **Connection quality**: Smoothed RTT, jitter, min/max RTT, loss ratio, link health (`healthy` / `degraded` / `critical`)
- **Adaptive keepalives**: Interval adjusts based on link health (60s → 30s → 10s)
- **Per-client rate limiting**: Token bucket with configurable bytes/sec and burst
- **Dead-peer detection**: 180s inactivity timeout (all transports)

- **MTU management**: Automatic overhead calculation (IP+TCP+WS+Noise = 79 bytes)
- **Per-transport stats**: Active client and total connection counts broken down by websocket, QUIC, and WireGuard

□□ Client Tags (Trusted vs Informational)

SmartVPN separates server-managed tags from client-reported tags:

Field	Set By	Trust Level	Use For
`serverDefinedClientTags`	Server admin (via `createClient` / `updateClient`)	□ Trusted	Access control, routing, billing
`clientDefinedClientTags`	Client (reported after connection)	⚠ Informational	Diagnostics, client self-identification
`tags`	*(deprecated)*	—	Legacy alias for `serverDefinedClientTags`

```
// Server-side: trusted tags
await server.createClient({
  clientId: 'alice-laptop',
  serverDefinedClientTags: ['engineering', 'office-berlin'],
});

// Client-side: informational tags (reported to server)
await client.connect({
  // ...
  clientDefinedClientTags: ['macOS', 'v2.1.0'],
});
```

□□ Hub Client Management

The server acts as a **hub** — one API to manage all clients:

```
// Create (generates keys, assigns IP, returns config bundle)
const bundle = await server.createClient({ clientId: 'bob-phone' });

// Read
const entry = await server.getClient('bob-phone');
const all   = await server.listRegisteredClients();
```

```
// Update (ACLs, tags, description, rate limits...)
await server.updateClient('bob-phone', {
  security: { destinationAllowList: ['0.0.0.0/0'] },
  serverDefinedClientTags: ['mobile', 'field-ops'],
});

// Enable / Disable
await server.disableClient('bob-phone'); // disconnects + blocks reconnection
await server.enableClient('bob-phone');

// Key rotation
const newBundle = await server.rotateClientKey('bob-phone');

// Export config (without secrets)
const wgConf = await server.exportClientConfig('bob-phone', 'wireguard');

// Remove
await server.removeClient('bob-phone');
```

□□ WireGuard Config Generation

Generate standard `.conf` files for any WireGuard client:

```
import { WgConfigGenerator } from '@push.rocks/smartvpn';

const conf = WgConfigGenerator.generateClientConfig({
  privateKey: '<client-wg-private-key>',
  address: '10.8.0.2/24',
  dns: ['1.1.1.1'],
  peer: {
    publicKey: '<server-wg-public-key>',
    endpoint: 'vpn.example.com:51820',
    allowedIps: ['0.0.0.0/0'],
    persistentKeepalive: 25,
  },
});
// → standard WireGuard .conf compatible with wg-quick, iOS, Android
```

Server configs too:

```
const serverConf = WgConfigGenerator.generateServerConfig({
  privateKey: '<server-wg-private-key>',
  address: '10.8.0.1/24',
  listenPort: 51820,
  enableNat: true,
  natInterface: 'eth0',
  peers: [
    { publicKey: '<client-wg-public-key>', allowedIps: ['10.8.0.2/32'] },
  ],
});
```

□□ System Service Installation

Generate systemd (Linux) or launchd (macOS) service units:

```
import { VpnInstaller } from '@push.rocks/smartvpn';

const unit = VpnInstaller.generateServiceUnit({
  binaryPath: '/usr/local/bin/smartvpn_daemon',
  socketPath: '/var/run/smartvpn.sock',
  mode: 'server',
});
// unit.platform    → 'linux' | 'macos'
// unit.content     → systemd unit file or launchd plist
// unit.installPath → /etc/systemd/system/smartvpn-server.service
```

You can also call `generateSystemdUnit()` or `generateLaunchdPlist()` directly for platform-specific options like custom descriptions.

□□ Events

Both `VpnServer` and `VpnClient` extend `EventEmitter` and emit typed events:

```
server.on('client-connected', (info: IVpnClientInfo) => {
  console.log(`${info.registeredClientId} connected from ${info.remoteAddr} via
${info.transportType}`);
});
```

```
server.on('client-disconnected', ({ clientId, reason }) => {
  console.log(`${clientId} disconnected: ${reason}`);
});

client.on('status', (status: IVpnStatus) => {
  console.log(`State: ${status.state}, IP: ${status.assignedIp}`);
});

// Both server and client emit:
server.on('exit', ({ code, signal }) => { /* daemon process exited */ });
server.on('reconnected', () => { /* socket transport reconnected */ });
```

Event	Emitted By	Payload
`status`	Both	`IVpnStatus` — connection state changes
`error`	Both	`{ message, code? }`
`client-connected`	Server	`IVpnClientInfo` — full client info including transport type
`client-disconnected`	Server	`{ clientId, reason? }`
`exit`	Both	`{ code, signal }` — daemon process exited
`reconnected`	Both	`void` — socket transport reconnected

API Reference □□

Classes

Class	Description
`VpnServer`	Manages the Rust daemon in server mode. Hub methods for client CRUD, telemetry, rate limits, WireGuard peer management.
`VpnClient`	Manages the Rust daemon in client mode. Connect, disconnect, status, telemetry.
`VpnBridge<T>`	Low-level typed IPC bridge (stdio or Unix socket). Handles spawn, connect, reconnect, and typed command dispatch.

Class	Description
`VpnConfig`	Static config validation and JSON file I/O. Validates keys, addresses, CIDRs, MTU, etc.
`VpnInstaller`	Generates systemd/launchd service files for daemon deployment.
`WgConfigGenerator`	Generates standard WireGuard `.conf` files (client and server).

Key Interfaces

Interface	Purpose
`IVpnServerConfig`	Server configuration (listen addr, keys, subnet, transport mode, forwarding mode, clients, proxy protocol, destination policy)
`IVpnClientConfig`	Client configuration (server URL, keys, transport, forwarding mode, WG options, client-defined tags)
`IClientEntry`	Server-side client definition (ID, keys, security, priority, server/client tags, expiry)
`IClientSecurity`	Per-client ACLs and rate limits (SmartProxy-aligned naming)
`IClientRateLimit`	Rate limiting config (bytesPerSec, burstBytes)
`IClientConfigBundle`	Full config bundle returned by `createClient()` — includes SmartVPN config, WireGuard .conf, and secrets
`IVpnClientInfo`	Connected client info (IP, stats, authenticated key, remote addr, transport type)
`IVpnServerStatistics`	Server stats with per-transport breakdowns (activeClientsWebsocket/Quic/Wireguard, totalConnections*)
`IVpnConnectionQuality`	RTT, jitter, loss ratio, link health
`IVpnMtuInfo`	TUN MTU, effective MTU, overhead bytes, oversized packet stats
`IVpnKeypair`	Base64-encoded public/private key pair
`IDestinationPolicy`	Destination routing policy (forceTarget / block / allow with allow/block lists)
`IVpnEventMap`	Typed event map for server and client EventEmitter

Server IPC Commands

Command	Description
`start` / `stop`	Start/stop the VPN listener
`createClient`	Generate keys, assign IP, return config bundle
`removeClient` / `getClient` / `listRegisteredClients`	Client registry CRUD
`updateClient` / `enableClient` / `disableClient`	Modify client state
`rotateClientKey`	Fresh keypairs + new config bundle
`exportClientConfig`	Re-export as SmartVPN config or WireGuard `.conf`
`listClients` / `disconnectClient`	Manage live connections
`setClientRateLimit` / `removeClientRateLimit`	Runtime rate limit adjustments
`getStatus` / `getStatistics` / `getClientTelemetry`	Monitoring (stats include per-transport breakdowns)
`generateKeypair` / `generateWgKeypair` / `generateClientKeypair`	Key generation
`addWgPeer` / `removeWgPeer` / `listWgPeers`	WireGuard peer management

Client IPC Commands

Command	Description
`connect` / `disconnect`	Manage the tunnel
`getStatus` / `getStatistics`	Connection state and traffic stats
`getConnectionQuality`	RTT, jitter, loss, link health
`getMtuInfo`	MTU and overhead details

Transport Modes □□

Server Configuration

```
// All transports simultaneously (default) — WS + QUIC + WireGuard
{ transportMode: 'all', listenAddr: '0.0.0.0:443', wgPrivateKey: '...', wgListenPort: 51820 }

// WS + QUIC only
{ transportMode: 'both', listenAddr: '0.0.0.0:443', quicListenAddr: '0.0.0.0:4433' }

// WebSocket only
```

```
{ transportMode: 'websocket', listenAddr: '0.0.0.0:443' }

// QUIC only
{ transportMode: 'quic', listenAddr: '0.0.0.0:443' }

// WireGuard only
{ transportMode: 'wireguard', wgPrivateKey: '...', wgListenPort: 51820, wgPeers: [...] }
```

All transport modes share the same `forwardingMode` — WireGuard peers can use `'socket'` (userspace NAT) just like WS/QUIC clients.

Client Configuration

```
// Auto (tries QUIC first, falls back to WS)
{ transport: 'auto', serverUrl: 'wss://vpn.example.com' }

// Explicit QUIC with certificate pinning
{ transport: 'quic', serverUrl: '1.2.3.4:4433', serverCertHash: '<sha256-base64>' }

// WireGuard
{ transport: 'wireguard', wgPrivateKey: '...', wgEndpoint: 'vpn.example.com:51820', ... }
```

Cryptography □□

Layer	Algorithm	Purpose
Handshake	Noise IK (X25519 + ChaChaPoly + BLAKE2s)	Mutual authentication + key exchange
Transport	Noise transport state (ChaChaPoly)	All post-handshake data encryption
Additional	XChaCha20-Poly1305	Extended nonce space for data-at-rest
WireGuard	X25519 + ChaCha20-Poly1305 (via boringtun)	Standard WireGuard crypto

Binary Protocol □□

All frames use `[type:1B][length:4B][payload:NB]` with a 64KB max payload:

Type	Hex	Direction	Description
HandshakeInit	`0x01`	Client → Server	Noise IK first message
HandshakeResp	`0x02`	Server → Client	Noise IK response
IpPacket	`0x10`	Bidirectional	Encrypted tunnel data
Keepalive	`0x20`	Client → Server	App-level keepalive (not WS ping)
KeepaliveAck	`0x21`	Server → Client	Keepalive response with RTT payload
Disconnect	`0x3F`	Bidirectional	Graceful disconnect

Development □□

```
# Install dependencies
pnpm install

# Build (TypeScript + Rust cross-compile)
pnpm build

# Run all tests
pnpm test

# Run Rust tests directly
cd rust && cargo test

# Run a specific TS test
tstest test/test.flowcontrol.node.ts --verbose
```

Project Structure

```
smartvpn/
├── ts/                          # TypeScript control plane
│   ├── index.ts                 # All exports
│   ├── smartvpn.interfaces.ts   # Interfaces, types, IPC command maps
│   ├── smartvpn.plugins.ts      # Dependency imports
│   ├── smartvpn.paths.ts        # Binary path resolution
```

```
│   ├── smartvpn.classes.vpnserver.ts
│   ├── smartvpn.classes.vpnclient.ts
│   ├── smartvpn.classes.vpnbridge.ts
│   ├── smartvpn.classes.vpnconfig.ts
│   ├── smartvpn.classes.vpninstaller.ts
│   └── smartvpn.classes.wgconfig.ts
├── rust/                      # Rust data plane daemon
│   └── src/
│       ├── main.rs            # CLI entry point
│       ├── server.rs          # VPN server + hub methods
│       ├── client.rs          # VPN client
│       ├── crypto.rs          # Noise IK + XChaCha20
│       ├── client_registry.rs # Client database
│       ├── acl.rs             # ACL engine
│       ├── proxy_protocol.rs  # PROXY protocol v2 parser
│       ├── management.rs      # JSON-lines IPC
│       ├── transport.rs       # WebSocket transport
│       ├── transport_trait.rs # Transport abstraction (Sink/Stream)
│       ├── quic_transport.rs  # QUIC transport
│       ├── wireguard.rs       # WireGuard (boringtun)
│       ├── codec.rs           # Binary frame protocol
│       ├── keepalive.rs       # Adaptive keepalives
│       ├── ratelimit.rs       # Token bucket
│       ├── userspace_nat.rs   # Userspace TCP/UDP NAT proxy
│       ├── tunnel.rs          # TUN device management
│       ├── network.rs         # IP pool + networking
│       ├── telemetry.rs       # RTT/jitter/loss tracking
│       ├── qos.rs             # Priority queues + smart dropping
│       ├── mtu.rs             # MTU + ICMP too-big
│       └── reconnect.rs       # Exponential backoff + session tokens
├── test/                      # Test files
├── dist_ts/                   # Compiled TypeScript
└── dist_rust/                 # Cross-compiled binaries (linux amd64 + arm64)
```

License and Legal Information

Please note: The MIT License does not grant permission to use the trade names, trademarks, service marks, or product names of the project, except as required for reasonable and customary use in describing the origin of the work and reproducing the content of the NOTICE file.

Trademarks

This project is owned and maintained by Task Venture Capital GmbH. The names and logos associated with Task Venture Capital GmbH and any related products or services are trademarks of Task Venture Capital GmbH or third parties, and are not included within the scope of the MIT license granted herein.

Use of these trademarks must comply with Task Venture Capital GmbH's Trademark Guidelines or the guidelines of the respective third-party owners, and any usage must be approved in writing. Third-party trademarks used herein are the property of their respective owners and used only in a descriptive manner, e.g. for an implementation of an API or similar.

Company Information

Task Venture Capital GmbH
Registered at District Court Bremen HRB 35230 HB, Germany

For any legal inquiries or further information, please contact us via email at hello@task.vc.

By using this repository, you acknowledge that you have read this section, agree to comply with its terms, and understand that the licensing of the code does not imply endorsement by Task Venture Capital GmbH of any derivative works.

changelog.md for @push.rocks/smartvpn

2026-03-31 - 1.17.1 - fix(readme)

document per-transport metrics and handshake-driven WireGuard connection state

- Add README examples for getStatistics() per-transport active client and total connection counters
- Clarify that WireGuard peers are marked connected only after a successful handshake and disconnect after idle timeout
- Refresh API and project structure documentation to reflect newly documented stats fields and source files

2026-03-31 - 1.17.0 - feat(wireguard)

track per-transport server statistics and make WireGuard clients active only after handshake

- add websocket, quic, and wireguard active-client and total-connection counters to server statistics
- register WireGuard peers without marking them active until handshake/data is received, and remove them from active clients on expiration or idle timeout
- sync WireGuard byte counters into aggregate server stats independently of active client presence and expose new statistics fields in TypeScript interfaces

2026-03-31 - 1.16.5 - fix(rust-userspace-nat)

improve TCP session backpressure, buffering, and idle cleanup in userspace NAT

- apply proper bridge-channel backpressure by reserving channel capacity before consuming smoltcp TCP data
- defer bridge sender initialization until the bridge task starts and track TCP session activity timestamps
- cap per-session pending TCP send buffers at 512KB and abort stalled sessions when clients cannot keep up
- add idle TCP session cleanup and switch NAT polling to a dynamic smoltcp-driven delay

2026-03-31 - 1.16.4 - fix(server)

register preloaded WireGuard clients as peers on server startup

- Adds configured clients from the runtime registry to the WireGuard listener when the server starts.
- Ensures clients loaded from config can complete WireGuard handshakes without requiring separate peer registration.
- Logs a warning if automatic peer registration fails for an individual client.

2026-03-31 - 1.16.3 - fix(rust-nat)

defer TCP bridge startup until handshake completion and buffer partial NAT socket writes

- Start TCP bridge tasks only after the smoltcp socket becomes active to prevent server data from arriving before the client handshake completes.
- Buffer pending TCP payloads and flush partial writes so bridge-to-socket data is not silently lost under backpressure.
- Keep closing TCP sessions alive until FIN processing completes and add logging for dropped packets when bridge or route channels are full.

2026-03-31 - 1.16.2 - fix(wireguard)

sync runtime peer management with client registration and derive the correct server public key from the WireGuard private key

- Register, remove, and rotate WireGuard peers in the running listener when clients are added, deleted, or rekeyed.

- Generate client WireGuard configs with the public key derived from the configured WireGuard private key instead of reusing the generic server public key.
- Handle expired WireGuard sessions by re-initiating handshakes and mark client state as handshaking until the tunnel becomes active.
- Improve allowed IP matching and peer VPN IP extraction for runtime packet routing.

2026-03-30 - 1.16.1 - fix(rust/server)

add serde alias for clientAllowedIPs in server config

- Accepts the camelCase clientAllowedIPs field when deserializing server configuration.
- Improves compatibility with existing or external configuration formats without changing runtime behavior.

2026-03-30 - 1.16.0 - feat(server)

add configurable client endpoint and allowed IPs for generated VPN configs

- adds serverEndpoint to generated SmartVPN and WireGuard client configs so remote clients can use a public address instead of the listen address
- adds clientAllowedIPs to generated WireGuard configs to support full-tunnel or split-tunnel routing
- updates TypeScript interfaces to expose the new server configuration options

2026-03-30 - 1.15.0 - feat(vpnserver)

add nftables-backed destination policy enforcement for TUN mode

- add @push.rocks/smartnftables dependency and export it through the plugin layer
- apply destination policy rules via nftables when starting the server in TUN mode
- add periodic nftables health checks and best-effort cleanup on server stop
- update documentation for destination routing policy, socket transport mode, trusted client tags, events, and service generation

2026-03-30 - 1.14.0 - feat(nat)

add destination routing policy support for socket-mode VPN traffic

- introduce configurable destinationPolicy settings in server and TypeScript interfaces
- apply allow, block, and forceTarget routing decisions when creating TCP and UDP NAT sessions
- export ACL IP matching helper for destination policy evaluation

2026-03-30 - 1.13.0 - feat(client-registry)

separate trusted server-defined client tags from client-reported tags with legacy tag compatibility

- Adds distinct serverDefinedClientTags and clientDefinedClientTags fields to client registry and TypeScript interfaces.
- Treats legacy tags values as serverDefinedClientTags during deserialization and server-side create/update flows for backward compatibility.
- Clarifies that only server-defined tags are trusted for access control while client-defined tags are informational only.

2026-03-30 - 1.12.0 - feat(server)

add optional PROXY protocol v2 headers for socket-based userspace NAT forwarding

- introduce a socketForwardProxyProtocol server option in Rust and TypeScript interfaces
- pass the new setting into the userspace NAT engine and TCP bridge tasks
- prepend PROXY protocol v2 headers on outbound TCP connections when socket forwarding is enabled

2026-03-30 - 1.11.0 - feat(server)

unify WireGuard into the shared server transport pipeline

- add integrated WireGuard server support to VpnServer with shared startup, shutdown, status, statistics, and peer management
- introduce transportMode 'all' as the default and add server config support for wgPrivateKey, wgListenPort, and preconfigured peers
- register WireGuard peers in the shared client registry and IP pool so they use the same forwarding engine, routing, and monitoring as WebSocket and QUIC clients
- expose transportType in server client info and update TypeScript interfaces and documentation to reflect unified multi-transport forwarding

2026-03-30 - 1.10.2 - fix(client)

wait for the connection task to shut down cleanly before disconnecting and increase test timeout

- store the spawned client connection task handle and await it during disconnect with a 5 second timeout so the disconnect frame can be sent before closing
- increase the test script timeout from 60 seconds to 90 seconds to reduce flaky test runs

2026-03-29 - 1.10.1 - fix(test, docs, scripts)

correct test command verbosity, shorten load test timings, and document forwarding modes

- Fixes the test script by removing the duplicated verbose flag in package.json.
- Reduces load test delays and burst sizes to keep keepalive and connection tests faster and more stable.
- Updates the README to describe forwardingMode options, userspace NAT support, and related configuration examples.

2026-03-29 - 1.10.0 - feat(rust-server, rust-client, ts-interfaces)

add configurable packet forwarding with TUN and userspace NAT modes

- introduce forwardingMode options for client and server configuration interfaces

- add server-side forwarding engines for kernel TUN, userspace socket NAT, and testing mode
- add a smoltcp-based userspace NAT implementation for packet forwarding without root-only TUN routing
- enable client-side TUN forwarding support with route setup, packet I/O, and cleanup
- centralize raw packet destination IP extraction in tunnel utilities for shared routing logic
- update test command timeout and logging flags

2026-03-29 - 1.9.0 - feat(server)

add PROXY protocol v2 support for real client IP handling and connection ACLs

- add PROXY protocol v2 parsing for WebSocket connections, including IPv4/IPv6 support, LOCAL command handling, and header read timeout protection
- apply server-level connection IP block lists before the Noise handshake and enforce per-client source IP allow/block lists using the resolved remote address
- expose proxy protocol configuration and remote client address fields in Rust and TypeScript interfaces, and document reverse-proxy usage in the README

2026-03-29 - 1.8.0 - feat(auth,client-registry)

add Noise IK client authentication with managed client registry and per-client ACL controls

- switch the native tunnel handshake from Noise NK to Noise IK and require client keypairs in client configuration
- add server-side client registry management APIs for creating, updating, disabling, rotating, listing, and exporting client configs
- enforce client authorization from the registry during handshake and expose authenticated client metadata in server client info
- introduce per-client security policies with source/destination ACLs and per-client rate limit settings
- add Rust ACL matching support for exact IPs, CIDR ranges, wildcards, and IP ranges with test coverage

2026-03-29 - 1.7.0 - feat(rust-tests)

add end-to-end WireGuard UDP integration tests and align TypeScript build configuration

- Add userspace Rust end-to-end tests that validate WireGuard handshake, encryption, peer isolation, and preshared-key data exchange over real UDP sockets.
- Update the TypeScript build setup by removing the allowimplicitany build flag and explicitly including Node types in tsconfig.
- Refresh development toolchain versions to support the updated test and build workflow.

2026-03-29 - 1.6.0 - feat(readme)

document WireGuard transport support, configuration, and usage examples

- Expand the README from dual-transport to triple-transport support by adding WireGuard alongside WebSocket and QUIC
- Add client and server WireGuard examples, including live peer management and .conf generation with WgConfigGenerator
- Document new WireGuard-related API methods, config fields, transport modes, and security model details

2026-03-29 - 1.5.0 - feat(wireguard)

add WireGuard transport support with management APIs and config generation

- add Rust WireGuard module integration using boringtun and route management through client/server management handlers
- extend TypeScript client and server configuration schemas with WireGuard-specific options and validation
- add server-side WireGuard peer management commands including keypair generation, peer add/remove, and peer listing
- export a WireGuard config generator for producing client and server .conf files
- add WireGuard-focused test coverage for config validation and config generation

2026-03-21 - 1.4.1 - fix(readme)

preserve markdown line breaks in feature list

- Adds trailing spaces to the README feature list so each highlighted capability renders on its own line.

2026-03-19 - 1.4.0 - feat(vpn transport)

add QUIC transport support with auto fallback to WebSocket

- introduces a transport abstraction in the Rust daemon so client and server can operate over WebSocket or QUIC
- adds dual-mode server configuration with websocket, quic, and both transport modes plus QUIC idle timeout and listen address options
- adds client transport selection with auto mode that attempts QUIC first and falls back to WebSocket
- adds QUIC certificate hash pinning support and required Rust dependencies for QUIC and TLS
- updates TypeScript interfaces, config validation, tests, and documentation to cover the new transport modes

2026-03-17 - 1.3.0 - feat(tests,client)

add flow control and load test coverage and honor configured keepalive intervals

- Adds end-to-end node tests for client/server flow control, keepalive exchange, connection quality telemetry, rate limiting, concurrent clients, and disconnect tracking.
- Adds load testing with throttled proxy scenarios to validate behavior under constrained bandwidth and repeated client churn.
- Updates the Rust client to pass configured keepaliveIntervalSecs into the adaptive keepalive monitor instead of always using defaults.

2026-03-15 - 1.2.0 - feat(readme)

document QoS, telemetry, MTU, and rate limiting capabilities in the README

- Expand the architecture and feature overview to cover adaptive keepalive, telemetry, QoS, rate limiting, and MTU handling
- Update client and server examples to show new APIs such as getConnectionQuality(), getMtuInfo(), setClientRateLimit(), and getClientTelemetry()
- Add TypeScript interface documentation for connection quality, MTU info, enriched client statistics, and per-client telemetry

2026-03-15 - 1.1.0 - feat(rust-core)

add adaptive keepalive telemetry, MTU handling, and per-client rate limiting APIs

- adds adaptive keepalive monitoring with RTT, jitter, loss, and link health reporting to client statistics and management endpoints
- introduces MTU overhead calculation and oversized-packet handling support, plus client MTU info APIs
- adds token-bucket rate limiting with configurable default limits and server management commands to set, remove, and inspect per-client telemetry
- extends TypeScript client and server interfaces with connection quality, MTU, and client telemetry methods

2026-02-27 - 1.0.3 - fix(build)

add aarch64 linker configuration for cross-compilation

- Added rust/.cargo/config.toml to configure linker for target aarch64-unknown-linux-gnu
- Sets linker to 'aarch64-linux-gnu-gcc' to enable cross-compilation to ARM64

2026-02-27 - 1.0.2 - fix()

no changes detected - no code or content modifications

2026-02-27 - 1.0.1 - fix(release)

bump patch version (no code changes)

- No changes detected in the provided git diff
- Current package.json version is 1.0.0
- Recommend patch bump to 1.0.1 to create a release/trivial update

2026-02-27 - 1.0.0 - initial release

Initial commit creating the project repository and baseline files.

- Initial project scaffold and configuration
- Repository initialized with base files and metadata