readme.md for @serve.zone/remoteingress

Edge ingress tunnel for DcRouter — tunnels **TCP and UDP** traffic from the network edge to a private DcRouter/SmartProxy cluster over encrypted TLS or QUIC connections, preserving the original client IP via PROXY protocol. Includes **hub-controlled nftables firewall** for IP blocking, rate limiting, and custom firewall rules applied directly at the edge.

Issue Reporting and Security

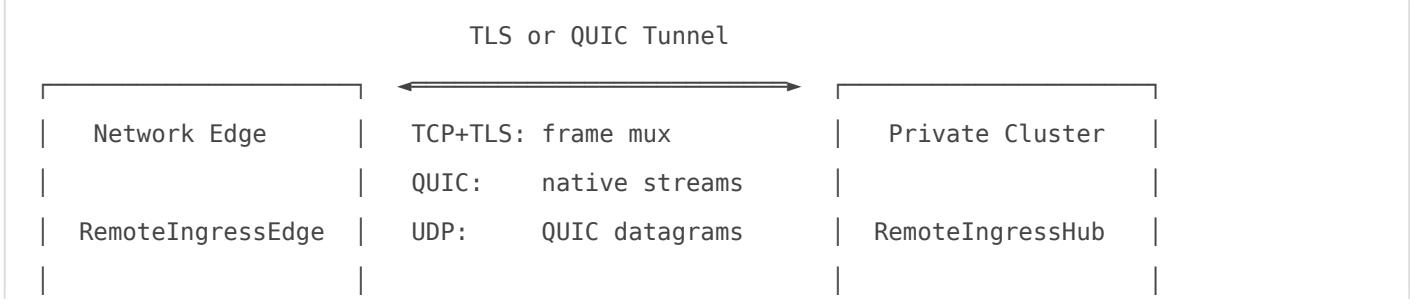
For reporting bugs, issues, or security vulnerabilities, please visit community.foss.global/. This is the central community hub for all issue reporting. Developers who sign and comply with our contribution agreement and go through identification can also get a code.foss.global/ account to submit Pull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

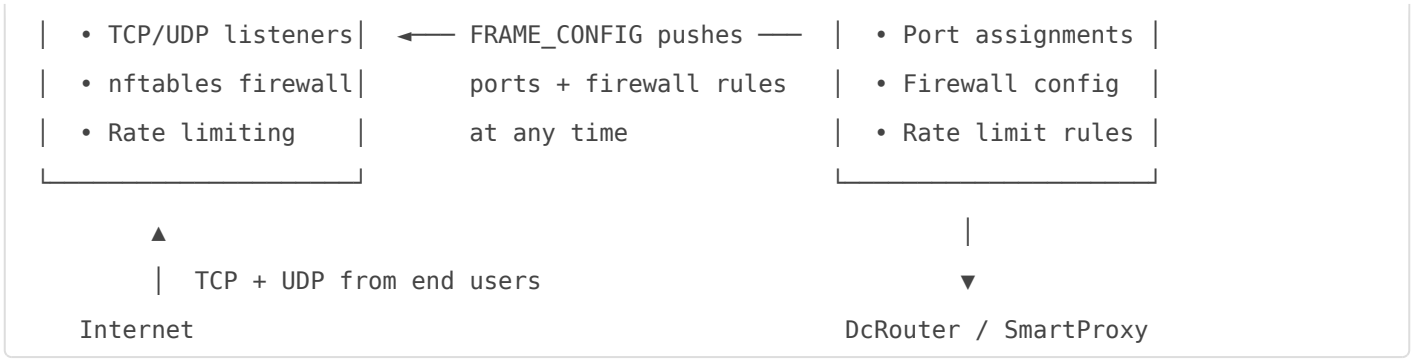
Install

```
pnpm install @serve.zone/remoteingress
```

Architecture

`@serve.zone/remoteingress` uses a **Hub/Edge** topology with a high-performance Rust core and a TypeScript API surface:





Component	Role
RemoteIngressEdge	Deployed at the network edge (VPS, cloud instance). Runs as root. Listens on hub-assigned TCP/UDP ports, tunnels traffic to the hub, and applies hub-pushed nftables rules (IP blocking, rate limiting). All config is hot-reloadable at runtime.
RemoteIngressHub	Deployed alongside DcRouter/SmartProxy in a private cluster. Accepts edge connections, demuxes streams/datagrams, and forwards each to SmartProxy with PROXY protocol headers so the real client IP is preserved. Pushes all edge config (ports, firewall) via a single API.
Rust Binary (`remoteingress-bin`)	The performance-critical networking core. Managed via `@push.rocks/smartrust` RustBridge IPC — you never interact with it directly. Cross-compiled for `linux/amd64` and `linux/arm64`.

⚡ Key Features

- **Dual transport** — choose between TCP+TLS (frame-multiplexed) or QUIC (native stream multiplexing, zero head-of-line blocking)
- **TCP + UDP tunneling** — tunnel any TCP connection or UDP datagram through the same edge/hub pair
- **PROXY protocol v1 & v2** — SmartProxy sees the real client IP for both TCP (v1 text) and UDP (v2 binary)
- **Hub-controlled firewall** — push nftables rules (IP blocking, rate limiting, custom firewall rules) to edges as part of the same config update that assigns ports — powered by `@push.rocks/smartnftables`
- **Multiplexed streams** — thousands of concurrent TCP connections over a single tunnel
- **QUIC datagrams** — UDP traffic forwarded via QUIC unreliable datagrams for lowest possible latency
- **Shared-secret authentication** — edges must present valid credentials to connect
- **Connection tokens** — encode all connection details into a single opaque base64url string
- **STUN-based public IP discovery** — edges automatically discover their public IP via Cloudflare STUN
- **Auto-reconnect** with exponential backoff if the tunnel drops

- **Dynamic runtime configuration** — the hub pushes ports, firewall rules, and rate limits to edges at any time via a single `updateAllowedEdges()` call
- **Event-driven** — both Hub and Edge extend `EventEmitter` for real-time monitoring
- **3-tier QoS** — control frames, normal data, and sustained (elephant flow) traffic each get their own priority queue
- **Adaptive flow control** — per-stream windows scale with active stream count to prevent memory overuse
- **UDP session management** — automatic session tracking with 60s idle timeout and cleanup
- **Crash recovery** — automatic restart with exponential backoff if the Rust binary crashes unexpectedly

Usage

Both classes are imported from the package and communicate with the Rust binary under the hood.

Setting Up the Hub (Private Cluster Side)

```
import { RemoteIngressHub } from '@serve.zone/remoteingress';

const hub = new RemoteIngressHub();

// Listen for events
hub.on('edgeConnected', ({ edgeId }) => console.log(`Edge ${edgeId} connected`));
hub.on('edgeDisconnected', ({ edgeId }) => console.log(`Edge ${edgeId} disconnected`));
hub.on('streamOpened', ({ edgeId, streamId }) => console.log(`Stream ${streamId} from
${edgeId}`));
hub.on('streamClosed', ({ edgeId, streamId }) => console.log(`Stream ${streamId} closed`));

// Start the hub — listens for edge connections on both TCP and QUIC (same port)
await hub.start({
  tunnelPort: 8443,        // port edges connect to (default: 8443)
  targetHost: '127.0.0.1', // SmartProxy host to forward traffic to
});

// Register allowed edges with TCP and UDP listen ports + firewall config
await hub.updateAllowedEdges([
  {
```

```
    id: 'edge-nyc-01',
    secret: 'supersecrettoken1',
    listenPorts: [80, 443],        // TCP ports the edge should listen on
    listenPortsUdp: [53, 51820],   // UDP ports (e.g., DNS, WireGuard)
    stunIntervalSecs: 300,
    firewallConfig: {
      blockedIps: ['192.168.1.100', '10.0.0.0/8'],
      rateLimits: [
        { id: 'http-rate', port: 80, protocol: 'tcp', rate: '100/second', perSourceIP: true },
      ],
      rules: [
        { id: 'allow-ssh', direction: 'input', action: 'accept', sourceIP: '10.0.0.0/24',
destPort: 22, protocol: 'tcp' },
      ],
    },
  },
  {
    id: 'edge-fra-02',
    secret: 'supersecrettoken2',
    listenPorts: [443, 8080],
  },
]);

// Dynamically update ports and firewall — changes are pushed instantly to connected edges
await hub.updateAllowedEdges([
  {
    id: 'edge-nyc-01',
    secret: 'supersecrettoken1',
    listenPorts: [80, 443, 8443],    // added TCP port 8443
    listenPortsUdp: [53],            // removed WireGuard UDP port
    firewallConfig: {
      blockedIps: ['192.168.1.100', '10.0.0.0/8', '203.0.113.50'],  // added new blocked IP
      rateLimits: [
        { id: 'http-rate', port: 80, protocol: 'tcp', rate: '200/second', perSourceIP: true },
      ],
    },
  },
]);
```

```
// Check status
const status = await hub.getStatus();
// { running: true, tunnelPort: 8443, connectedEdges: [...] }

await hub.stop();
```

Setting Up the Edge (Network Edge Side)

The edge can connect via **TCP+TLS** (default) or **QUIC** transport. Edges run as **root** so they can bind to privileged ports and apply nftables firewall rules.

Option A: Connection Token (Recommended)

```
import { RemoteIngressEdge } from '@serve.zone/remoteingress';

const edge = new RemoteIngressEdge();

edge.on('tunnelConnected', () => console.log('Tunnel established'));
edge.on('tunnelDisconnected', () => console.log('Tunnel lost — will auto-reconnect'));
edge.on('publicIpDiscovered', ({ ip }) => console.log(`Public IP: ${ip}`));
edge.on('portsAssigned', ({ listenPorts }) => console.log(`TCP ports: ${listenPorts}`));
edge.on('firewallConfigUpdated', () => console.log('Firewall rules applied'));

await edge.start({
  token: 'eyJoIjoiaHViLmV4YW1wbGUuY29tIiwi...',
});
```

Option B: Explicit Config with QUIC Transport

```
import { RemoteIngressEdge } from '@serve.zone/remoteingress';

const edge = new RemoteIngressEdge();

await edge.start({
  hubHost: 'hub.example.com',
  hubPort: 8443,
  edgeId: 'edge-nyc-01',
  secret: 'supersecrettoken1',
  transportMode: 'quic',  // 'tcpTls' (default) | 'quic' | 'quicWithFallback'
```

```
});

const edgeStatus = await edge.getStatus();
// { running: true, connected: true, publicIp: '203.0.113.42', activeStreams: 5, listenPorts:
[80, 443] }

await edge.stop();
```

Transport Modes

Mode	Description
`'tcpTls'`	**Default.** Single TLS connection with frame-based multiplexing. Universal compatibility.
`'quic'`	QUIC with native stream multiplexing. Eliminates head-of-line blocking. Uses QUIC datagrams for UDP traffic.
`'quicWithFallback'`	Tries QUIC first (5s timeout), falls back to TCP+TLS if UDP is blocked by the network.

Connection Tokens

Encode all connection details into a single opaque string for easy distribution:

```
import { encodeConnectionToken, decodeConnectionToken } from '@serve.zone/remoteingress';

// Hub operator generates a token
const token = encodeConnectionToken({
  hubHost: 'hub.example.com',
  hubPort: 8443,
  edgeId: 'edge-nyc-01',
  secret: 'supersecrettoken1',
});
// => 'eyJoIjoiaHViLmV4YW1wbGUuY29tIiwi...'

// Edge operator decodes (optional — start() does this automatically)
const data = decodeConnectionToken(token);
// { hubHost: 'hub.example.com', hubPort: 8443, edgeId: 'edge-nyc-01', secret: '...' }
```

Tokens are base64url-encoded — safe for environment variables, CLI arguments, and config files.

□□ Firewall Config

The `firewallConfig` field in `updateAllowedEdges()` works exactly like `listenPorts` — it travels in the same `FRAME_CONFIG` frame, is delivered on initial handshake and on every subsequent update, and is applied atomically at the edge using `@push.rocks/smartnftables`. Each update fully replaces the previous ruleset.

Since edges run as root, the rules are applied directly to the Linux kernel via nftables. If the edge isn't root or nftables is unavailable, it logs a warning and continues — the tunnel works fine, just without kernel-level firewall rules.

Config Structure

```
interface IFirewallConfig {
  blockedIps?: string[];           // IPs or CIDRs to block (e.g., '1.2.3.4', '10.0.0.0/8')
  rateLimits?: IFirewallRateLimit[];
  rules?: IFirewallRule[];
}

interface IFirewallRateLimit {
  id: string;                      // unique identifier for this rate limit
  port: number;                    // port to rate-limit
  protocol?: 'tcp' | 'udp';       // default: both
  rate: string;                    // e.g., '100/second', '1000/minute'
  burst?: number;                  // burst allowance
  perSourceIP?: boolean;           // per-client rate limiting (recommended)
}

interface IFirewallRule {
  id: string;                      // unique identifier for this rule
  direction: 'input' | 'output' | 'forward';
  action: 'accept' | 'drop' | 'reject';
  sourceIP?: string;               // source IP or CIDR
  destPort?: number;               // destination port
  protocol?: 'tcp' | 'udp';
  comment?: string;
}
```

Example: Rate Limiting + IP Blocking

```
await hub.updateAllowedEdges([
  {
    id: 'edge-nyc-01',
    secret: 'secret',
    listenPorts: [80, 443],
    firewallConfig: {
      // Block known bad actors
      blockedIps: ['198.51.100.0/24', '203.0.113.50'],

      // Rate limit HTTP traffic per source IP
      rateLimits: [
        { id: 'http', port: 80, protocol: 'tcp', rate: '100/second', burst: 50, perSourceIP:
true },
        { id: 'https', port: 443, protocol: 'tcp', rate: '200/second', burst: 100,
perSourceIP: true },
      ],

      // Allow monitoring from trusted subnet
      rules: [
        { id: 'monitoring', direction: 'input', action: 'accept', sourceIP: '10.0.0.0/24',
destPort: 9090, protocol: 'tcp', comment: 'Prometheus scraping' },
      ],
    },
  },
]);
```

API Reference

`RemoteIngressHub`

Method / Property	Description
`start(config?)`	Start the hub. Config: `{ tunnelPort?, targetHost?, tls?: { certPem?, keyPem? } }`. Listens on both TCP and UDP (QUIC) on the tunnel port.
`stop()`	Graceful shutdown.

Method / Property	Description
`updateAllowedEdges(edges)`	Set authorized edges. Each: `{ id, secret, listenPorts?, listenPortsUdp?, stunIntervalSecs?, firewallConfig? }`. Port and firewall changes are pushed to connected edges in real time.
`getStatus()`	Returns `{ running, tunnelPort, connectedEdges: [...] }`.
`running`	`boolean` — whether the Rust binary is alive.

Events: `edgeConnected`, `edgeDisconnected`, `streamOpened`, `streamClosed`, `crashRecovered`, `crashRecoveryFailed`

`RemoteIngressEdge`

Method / Property	Description
`start(config)`	Connect to hub. Accepts `{ token }` or `{ hubHost, hubPort, edgeId, secret, bindAddress?, transportMode? }`.
`stop()`	Graceful shutdown. Cleans up all nftables rules.
`getStatus()`	Returns `{ running, connected, publicIp, activeStreams, listenPorts }`.
`running`	`boolean` — whether the Rust binary is alive.

Events: `tunnelConnected`, `tunnelDisconnected`, `publicIpDiscovered`, `portsAssigned`, `portsUpdated`, `firewallConfigUpdated`, `crashRecovered`, `crashRecoveryFailed`

Token Utilities

Function	Description
`encodeConnectionToken(data)`	Encodes connection info into a base64url token.
`decodeConnectionToken(token)`	Decodes a token. Throws on malformed input.

Interfaces

```
interface IHubConfig {
  tunnelPort?: number;   // default: 8443
  targetHost?: string;   // default: '127.0.0.1'
  tls?: {
    certPem?: string;    // PEM-encoded TLS certificate
```

```
    keyPem?: string;     // PEM-encoded TLS private key
  };
}

interface IEdgeConfig {
  hubHost: string;
  hubPort?: number;      // default: 8443
  edgeId: string;
  secret: string;
  bindAddress?: string;
  transportMode?: 'tcpTls' | 'quic' | 'quicWithFallback';
}

interface IConnectionTokenData {
  hubHost: string;
  hubPort: number;
  edgeId: string;
  secret: string;
}
```

Wire Protocol

TCP+TLS Transport (Frame Protocol)

The tunnel uses a custom binary frame protocol over a single TLS connection:

```
[stream_id: 4 bytes BE][type: 1 byte][length: 4 bytes BE][payload: N bytes]
```

Frame Type	Value	Direction	Purpose
`OPEN`	`0x01`	Edge → Hub	Open TCP stream; payload is PROXY v1 header
`DATA`	`0x02`	Edge → Hub	Client data (upload)
`CLOSE`	`0x03`	Edge → Hub	Client closed connection
`DATA_BACK`	`0x04`	Hub → Edge	Response data (download)
`CLOSE_BACK`	`0x05`	Hub → Edge	Upstream closed connection

Frame Type	Value	Direction	Purpose
`CONFIG`	`0x06`	Hub → Edge	Runtime config update (JSON: ports + firewall config)
`PING`	`0x07`	Hub → Edge	Heartbeat probe (every 15s)
`PONG`	`0x08`	Edge → Hub	Heartbeat response
`WINDOW_UPDATE`	`0x09`	Edge → Hub	Flow control: edge consumed N bytes
`WINDOW_UPDATE_BACK`	`0x0A`	Hub → Edge	Flow control: hub consumed N bytes
`UDP_OPEN`	`0x0B`	Edge → Hub	Open UDP session; payload is PROXY v2 header
`UDP_DATA`	`0x0C`	Edge → Hub	UDP datagram (upload)
`UDP_DATA_BACK`	`0x0D`	Hub → Edge	UDP datagram (download)
`UDP_CLOSE`	`0x0E`	Either	Close UDP session

QUIC Transport

When using QUIC, the frame protocol is replaced by native QUIC primitives:

- **TCP connections:** Each tunneled TCP connection gets its own QUIC bidirectional stream. No framing overhead.
- **UDP datagrams:** Forwarded via QUIC unreliable datagrams (RFC 9221). Format: `[session_id: 4 bytes][payload]`. Session open uses magic byte `0xFF`: `[session_id: 4][0xFF][PROXY v2 header]`.
- **Control channel:** First QUIC bidirectional stream carries auth handshake + config updates using `[type: 1][length: 4][payload]` format.

Handshake Sequence

1. Edge opens a TLS or QUIC connection to the hub
2. Edge sends: `EDGE <edgeId> <secret>\n`
3. Hub verifies credentials (constant-time comparison) and responds with JSON: `{"listenPorts":[...],"listenPortsUdp":[...],"stunIntervalSecs":300,"firewallConfig":{...}}\n`
4. Edge starts TCP and UDP listeners on the assigned ports
5. Edge applies firewall config via nftables (if present and running as root)
6. Data flows — TCP frames/QUIC streams for TCP traffic, UDP frames/QUIC datagrams for UDP traffic

QoS & Flow Control

Priority Tiers (TCP+TLS Transport)

Tier	Frames	Behavior
Control	PING, PONG, WINDOW_UPDATE, OPEN, CLOSE, CONFIG	Always drained first. Never delayed.
Data	DATA/DATA_BACK from normal streams, UDP frames	Drained when control queue is empty.
Sustained	DATA/DATA_BACK from elephant flows	Lowest priority with guaranteed **1 MB/s** drain rate.

Sustained Stream Classification

A TCP stream is classified as **sustained** (elephant flow) when:

- Active for **>10 seconds**, AND
- Average throughput exceeds **20 Mbit/s** (2.5 MB/s)

Once classified, its flow control window locks to 1 MB and data frames move to the lowest-priority queue.

Adaptive Per-Stream Windows

Each TCP stream has a send window from a shared **200 MB budget**:

Active Streams	Window per Stream
1–50	4 MB (maximum)
51–200	Scales down (4 MB → 1 MB)
200+	1 MB (floor)

UDP traffic uses no flow control — datagrams are fire-and-forget, matching UDP semantics.

Example Scenarios

1. □□ Expose a Private Cluster to the Internet

Deploy an Edge on a public VPS, point DNS to its IP. The Edge tunnels all TCP and UDP traffic to the Hub running inside your private cluster. No public ports needed on the cluster.

2. □□ Multi-Region Edge Ingress

Run Edges in NYC, Frankfurt, and Tokyo — all connecting to a single Hub. Use GeoDNS to route users to their nearest Edge. PROXY protocol ensures the Hub sees real client IPs regardless of which Edge they entered through.

3. □□ UDP Forwarding (DNS, Gaming, VoIP)

Configure UDP listen ports alongside TCP ports. DNS queries, game server traffic, or VoIP packets are tunneled through the same edge/hub connection and forwarded to SmartProxy with a PROXY v2 binary header preserving the client's real IP.

```
await hub.updateAllowedEdges([
  {
    id: 'edge-nyc-01',
    secret: 'secret',
    listenPorts: [80, 443],       // TCP
    listenPortsUdp: [53, 27015],  // DNS + game server
  },
]);
```

4. □□ QUIC Transport for Low-Latency

Use QUIC transport to eliminate head-of-line blocking — a lost packet on one stream doesn't stall others. QUIC also enables 0-RTT reconnection and connection migration.

```
await edge.start({
  hubHost: 'hub.example.com',
  hubPort: 8443,
  edgeId: 'edge-01',
  secret: 'secret',
```

```
  transportMode: 'quicWithFallback', // try QUIC, fall back to TLS if UDP blocked
});
```

5. □□ Token-Based Edge Provisioning

Generate connection tokens on the hub side and distribute them to edge operators:

```
import { encodeConnectionToken, RemoteIngressEdge } from '@serve.zone/remoteingress';

const token = encodeConnectionToken({
  hubHost: 'hub.prod.example.com',
  hubPort: 8443,
  edgeId: 'edge-tokyo-01',
  secret: 'generated-secret-abc123',
});
// Send `token` to the edge operator — a single string is all they need

const edge = new RemoteIngressEdge();
await edge.start({ token });
```

6. □□ Centralized Firewall Management

Push firewall rules from the hub to all your edge nodes. Block bad actors, rate-limit abusive traffic, and whitelist trusted subnets — all from a single control plane:

```
await hub.updateAllowedEdges([
  {
    id: 'edge-nyc-01',
    secret: 'secret',
    listenPorts: [80, 443],
    firewallConfig: {
      blockedIps: ['198.51.100.0/24'],
      rateLimits: [
        { id: 'https', port: 443, protocol: 'tcp', rate: '500/second', perSourceIP: true,
burst: 100 },
      ],
      rules: [
        { id: 'allow-monitoring', direction: 'input', action: 'accept', sourceIP:
```

```
'10.0.0.0/8', destPort: 9090, protocol: 'tcp' },
      ],
    },
  },
]);
// Firewall rules are applied at the edge via nftables within seconds
```

License and Legal Information

This repository contains open-source code licensed under the MIT License. A copy of the license can be found in the LICENSE file.

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.

Revision #3
Created 2026-03-28 11:14:29 UTC by foss.global Team
Updated 2026-03-28 12:21:15 UTC by foss.global Team