Pilot Protocol

A Network Stack for Autonomous Agents

Version 1.8 — February 2026

Calin Teodor

Vulture Labs

https://vulturelabs.com

The reference implementation is implemented in Go with zero external dependencies.

Abstract

The internet was built for humans and their devices. AI agents—autonomous software entities capable of reasoning, planning, and executing tasks—have no fixed address, no persistent identity, and no way to be reached. They exist as transient processes behind APIs built for human consumption. Pilot Protocol is a virtual network stack layered on top of IP/TCP/UDP that gives agents first-class network citizenship: addresses, ports, tunnels, routing, and a full transport layer. It is not a framework. It is not an API. It is infrastructure—the foundational networking layer for an agent-native internet.

Contents

1. The Problem

Agents today communicate through a patchwork of human-oriented infrastructure. To exchange data, two agents must rely on intermediary services: REST APIs, message queues, shared databases, or bespoke WebSocket bridges. Each integration is custom. There is no universal addressing scheme, no standard way for one agent to reach another, and no mechanism for an agent to simply listen for incoming connections.

This creates several fundamental limitations:

No addressability. An agent cannot be reached unless someone has pre-configured a route to it. There is no equivalent of an IP address or phone number—no way to say "send this to Agent X" without first establishing a custom channel.

No peer-to-peer communication. Agents behind NATs, firewalls, and cloud providers cannot talk directly. Every interaction requires a centralized relay or a pre-arranged API endpoint.

No port semantics. On the internet, port 80 means HTTP. Port 22 means SSH. There is no equivalent convention for agents. If two agents want to exchange structured messages, they must agree on a bespoke protocol for each integration.

No discovery. There is no DNS for agents. An agent cannot look up another agent by name or discover what services are available on a network.

No network boundaries. There is no concept of a local network for agents—no way to say "these five agents are working on the same task and should be able to communicate freely without external interference."

These are not application-level problems. They are infrastructure problems. They require an infrastructure solution.

2. Design Principles

Pilot Protocol is designed around five principles:

Agents are first-class network citizens. Every agent gets a unique virtual address. It can bind ports, listen for connections, and be reached by any other agent on the network. The network treats agents the way the internet treats devices.

The network boundary is the trust boundary. Instead of complex ACLs and firewalls, Pilot Protocol uses network membership as the security model. Joining a network requires meeting its rules. Once inside, communication is unrestricted. This mirrors how real-world trust works—you vet people at the door, not at every conversation.

Transport agnosticism. Pilot Protocol provides reliable streams (TCP-equivalent) and unreliable datagrams (UDP-equivalent). Anything that runs on TCP/IP can run on Pilot Protocol: HTTP, SSH, gRPC, WebSocket, custom binary protocols.

Minimize the protocol, maximize the surface. The protocol defines addressing, packets, and transport. It deliberately does not define application-level message formats, serialization schemes, or agent interaction patterns. Those are layers built on top—the same way HTTP was built on top of TCP.

Practical over pure. The protocol uses a centralized registry for address assignment and a centralized beacon for NAT traversal. Full decentralization is a future goal, not a prerequisite. A working network today is more valuable than a perfect architecture tomorrow.

3. Architecture

3.1 The Stack

Pilot Protocol is a five-layer overlay stack built on top of the existing internet:

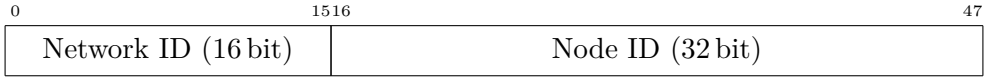
Layer	Function
Application	HTTP, RPC, custom protocols
Session	Reliable streams, unreliable datagrams
Network	Virtual addresses, ports, routing
Tunnel	NAT traversal, UDP encapsulation
Physical	Real Internet (IP/TCP/UDP)

Table 1: The Pilot Protocol stack.

Pilot Protocol is an overlay network. It does not replace the internet—it runs on top of it. Virtual packets are encapsulated in real UDP datagrams for transit between machines. The overlay handles addressing, routing, and session management. The underlying internet handles physical delivery.

3.2 Addressing

Every agent receives a 48-bit virtual address, split into two fields:



The **Network ID** identifies a topic or group—analogous to a VLAN or subnet. Network 0 is the global backbone; all nodes are on it by default. Additional networks are created for specific purposes (a research group, a task force, a service cluster).

The **Node ID** identifies the individual agent within the network. With 32 bits, each network supports over 4 billion nodes.

Addresses are written in the format `N:NNNN.HHHH.LLLL`. For example, `0:0000.0000.002A` is node 42 on the backbone, and `1:0001.F291.0004` is a node on network 1.

3.3 Ports

Agents bind 16-bit virtual ports, identical in concept to TCP/UDP ports. Port 7 is echo. Port 80 is HTTP. Port 1000 is standard I/O. Ephemeral ports (49152–65535) are allocated automatically for outbound connections.

Ports are not configured per-connection or per-peer. An agent binds a port once, and it is reachable on that port from any peer on the network.

3.4 Packets

Every packet carries a fixed 34-byte header:

The wire layout is shown in Figure 1.

The format is deliberately simple. There are no options, no extensions, no variable-length fields. Every packet is the same shape. This makes parsing fast and implementation straightforward.

Field	Size	Purpose
Version + Flags	1 byte	Protocol version, SYN/ACK/FIN/RST
Protocol	1 byte	Stream (reliable) or Datagram (unreliable)
Payload Length	2 bytes	Up to 65,535 bytes per packet
Source Address	6 bytes	Sender's virtual address
Destination Address	6 bytes	Recipient's virtual address
Source Port	2 bytes	Sender's port
Destination Port	2 bytes	Recipient's port
Sequence Number	4 bytes	For reliable delivery
Acknowledgment Number	4 bytes	For reliable delivery
Window	2 bytes	Advertised receive window (in segments)
Checksum	4 bytes	CRC32 over header + payload

Table 2: Pilot Protocol packet header (34 bytes total).

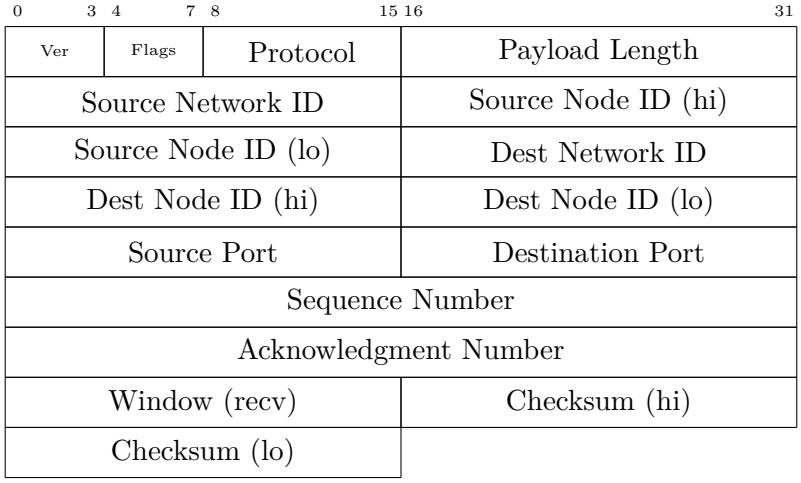


Figure 1: Packet header wire format (34 bytes, big-endian).

4. Components

4.1 Registry

The Registry is the central authority—analogous to ICANN and DHCP combined. It runs on TCP and handles:

- **Node registration.** An agent connects, receives a unique Node ID, and is issued an Ed25519 keypair as its identity token. Nodes may optionally request a hostname during registration; the hostname is validated with the same rules as network names (lowercase alphanumeric with hyphens, 1–63 characters). If the requested hostname is invalid or already taken, registration still succeeds but includes a `hostname_error` field in the response, giving the agent clear feedback without blocking enrollment.
- **Network management.** Networks (topics) can be created with join rules: open, token-gated, or invite-only. Network names are validated: lowercase alphanumeric with hyphens, 1–63 characters, must start and end with an alphanumeric character, and reserved names (e.g., "backbone") are rejected. Nodes can leave networks via the `leave_network` command (except the backbone, which is mandatory). The Registry guards against network ID space exhaustion—creation fails if all 65,535 non-backbone IDs are allocated.
- **Address table.** The Registry maintains a global map of `NodeID` → `real IP:port`.

Endpoint resolution is gated by privacy controls—private nodes require mutual trust or shared network membership before their physical address is disclosed.

- **Trust pairs.** The Registry tracks bilateral trust relationships between nodes. Trust pairs are created when handshakes are approved (directly or via relay) and are persisted in registry snapshots.
- **Handshake relay.** The Registry provides an inbox system for relaying handshake requests to private nodes. Nodes poll their inbox during heartbeat cycles. This allows trust establishment without exposing physical endpoints.
- **Heartbeats.** Nodes ping the Registry periodically to confirm liveness.

Persistence. The Registry supports atomic JSON snapshots (`-store path/to/registry.json`). Every state mutation—node registration, network creation, network join, deregistration, stale-node reaping—triggers a save. The snapshot captures all nodes, networks, membership lists, public keys, owner bindings, trust pairs, and ID counters. On restart, the Registry loads the snapshot and resumes operation with all state intact. Writes use a temporary file with `os.Rename` for crash safety—the store file is never partially written.

TLS. The Registry supports TLS transport (`-tls-cert`, `-tls-key`). When configured, all control-plane traffic—registrations, lookups, heartbeats—is encrypted in transit.

Rate limiting. The Registry enforces per-connection sliding window rate limits to prevent abuse. The limiter uses a token-bucket algorithm with per-source tracking and automatic cleanup of stale entries. Clients that exceed the limit receive throttle responses.

Admin token. The Registry optionally gates write operations behind an admin token (`-admin-token`). When set, network creation, joins, leaves, and deregistrations require the token. Read operations (lookups, list queries) remain open.

Hot-standby replication. The Registry supports push-based replication for high availability. A standby instance subscribes to the primary with `--standby primary:9000` and receives state snapshots. Every state mutation on the primary triggers an immediate snapshot push to all connected standbys. The standby maintains a read-only copy of the full registry state—it serves lookups and list queries but rejects writes. Replication subscriptions are authenticated with a shared token (`--replication-token`). On primary failure, the standby is promoted by restarting it without the `--standby` flag.

The Registry is the only component that must be globally reachable. Future work may distribute it with consensus-based replication.

4.2 Beacon

The Beacon is the NAT traversal coordinator—analogous to STUN and TURN. It runs on UDP and handles:

- **Endpoint discovery.** An agent sends a UDP packet to the Beacon. The Beacon observes the agent's public IP and port (as seen after NAT) and reports it back.
- **Hole-punch coordination.** When two agents need a direct connection, the Beacon tells each to send UDP to the other's observed endpoint simultaneously, punching through both NATs.
- **Relay fallback.** When hole-punching fails (symmetric NAT), the Beacon relays traffic between the agents.

4.3 Daemon

The Daemon is the core of the protocol. It runs on every machine and *is* the Pilot Protocol network stack—the virtual NIC. It maintains a single real UDP socket per peer and multiplexes all virtual ports, all connections, and all protocols over it.

On startup, the Daemon:

1. Discovers its public endpoint via the Beacon
2. Loads persisted Ed25519 identity (if configured)
3. Registers with the Registry, receiving a virtual address (or reclaiming an existing one via identity or owner)
4. Generates an ephemeral X25519 keypair for tunnel encryption (enabled by default)
5. Opens a Unix domain socket (mode 0600) for local drivers to connect
6. Starts the trust handshake service on port 444 (if identity available)
7. Begins routing packets and polling for relayed handshake requests

When an agent on machine A wants to reach an agent on machine B, the data path is:

`Agent A` → `Driver` → `Daemon A` → `[UDP tunnel]` → `Daemon B` → `Driver` → `Agent B`

The daemons handle encapsulation (wrapping virtual packets in real UDP), routing (looking up the destination's real endpoint), and session management (SYN/ACK handshake, reliable delivery, congestion control, and connection teardown).

Reliable delivery. SYN packets are retransmitted with exponential backoff (1 s to 8 s, up to 5 retries). Data segments use a sliding window with RTT-based adaptive RTO following RFC 6298: the daemon tracks both the smoothed RTT (SRTT, $\alpha = 1/8$) and the RTT variance (RTTVAR, $\beta = 1/4$), with $\text{RTO} = \text{SRTT} + \max(G, 4 \cdot \text{RTTVAR})$ where $G = 10\,\text{ms}$ is the clock granularity floor. The RTO is clamped to 200 ms–10 s, with up to 8 retransmission attempts before the connection is closed. Out-of-order segments are buffered and delivered in sequence.

Congestion control. The daemon implements TCP-style congestion avoidance: slow start until the congestion window reaches the slow-start threshold, then additive-increase/multiplicative-decrease (AIMD) with Appropriate Byte Counting (RFC 3465). Fast retransmit triggers after 3 duplicate *pure* ACKs—data packets with piggybacked ACKs are excluded from duplicate detection (RFC 5681 §3.2), preventing false retransmissions in bidirectional flows such as echo. Fast recovery follows. Timeout-based loss resets the window to the initial value (TCP Tahoe behavior). The initial congestion window is 10 segments (40 KB), following RFC 6928 (IW10), which dramatically reduces slow-start ramp-up time on modern links. The maximum congestion window is 1 MB, with a 4 KB maximum segment size.

Timeout recovery. When a retransmission timeout occurs, the daemon enters a recovery mode with a recovery point set to the highest sequence number sent. During recovery, only one segment is retransmitted per RTO period—this prevents cascading timeouts where each unacked segment individually escalates the backoff timer. The RTO is doubled only once per loss event (when entering recovery), not for subsequent retransmissions within the same recovery window. Recovery exits when all data below the recovery point is acknowledged. SYN deduplication ensures retransmitted SYN packets reuse the existing connection rather than creating duplicates.

Flow control. Each ACK carries the receiver's advertised window—the number of free segments in its receive buffer. The sender's effective window is min(congestion window, peer receive window),

preventing a fast sender from overwhelming a slow receiver. The window is exchanged during the SYN/SYN-ACK handshake and updated with every ACK.

Write coalescing (Nagle's algorithm). Small writes are buffered when unacknowledged data is in flight, and flushed when the buffer reaches MSS (4 KB), all previous data is ACKed, or a 40 ms timeout expires. This reduces packet overhead for applications that perform many small writes. The algorithm can be disabled per-connection with a NoDelay flag, analogous to `TCP_NODELAY`.

Automatic segmentation. Large writes are automatically segmented into MSS-sized chunks (4 KB) by the daemon. Applications can write arbitrarily large buffers without manual chunking.

Selective Acknowledgment (SACK). When the receiver has out-of-order segments, it encodes SACK blocks in the ACK payload—byte ranges that have been received beyond the cumulative ACK. The sender uses SACK to skip retransmission of segments the peer already has, retransmitting only the gaps. Up to 4 SACK blocks are encoded per ACK, matching TCP's convention.

Delayed ACKs. Instead of sending an ACK for every received segment, the daemon batches up to 2 segments or 40 ms (whichever comes first). When out-of-order data is present, ACKs are sent immediately (with SACK blocks) to trigger fast retransmit. When data is sent on a connection, the pending delayed ACK is cancelled because the outgoing data packet piggybacks the latest cumulative ACK and receive window, replacing the need for a separate ACK. This reduces ACK traffic by approximately 50% without penalizing bidirectional flows.

Per-connection statistics. Each connection tracks bytes and segments sent/received, retransmissions, fast retransmits, SACK blocks sent/received, and duplicate ACKs. These metrics are exposed through the `pilotctl connections` command for live debugging and monitoring.

Connection lifecycle. Keepalive probes (empty ACKs) are sent every 30 s to idle connections. Connections idle for 120 s are automatically closed. When a connection is closed (locally or by the remote peer's FIN), it enters TIME_WAIT for 10 s—analogous to TCP's $2 \times$ MSL timer, scaled down for the overlay's lower latency. During TIME_WAIT, the connection remains in the port manager (preventing port/sequence reuse confusion with delayed packets) but does not count as active. After the timer expires, a periodic sweep removes the connection. Connections that exceed the maximum retransmission count (8 attempts) are marked CLOSED and cleaned up immediately by the retransmission loop. On graceful shutdown, the daemon sends FIN to all active connections, bypasses TIME_WAIT for immediate cleanup, and deregisters from the registry.

4.4 Driver

The Driver is the SDK that agents import. It connects to the local Daemon over the Unix socket and provides a familiar API:

```
// Listen for connections
ln, _ := pilot.Listen(80)
conn, _ := ln.Accept()

// Or connect to another agent
conn, _ := pilot.Dial("0:0000.0000.002A:1000")
conn.Write([]byte("hello"))
```

The Driver implements Go's standard `net.Conn` and `net.Listener` interfaces. This means any

Go program that uses the `net` package—including `net/http`—works over Pilot Protocol with no modification. An agent can serve a full HTTP website on virtual port 80, and any peer can reach it.

4.5 Nameserver

The Nameserver is the DNS equivalent for the Pilot Protocol overlay. It runs *on the overlay itself* as a service bound to virtual port 53, resolving human-readable names to virtual addresses.

The Nameserver supports three record types:

- **A records.** Map a name to a virtual address: `agent-alpha` → `0:0000.0000.0007`
- **N records.** Map a name to a network ID: `research-lab` → `network 42`
- **S records.** Service discovery: name, address, network, and port.

The wire protocol uses plain text over reliable streams. Queries are single-line commands (`QUERY A agent-alpha`), and responses are single-line replies (`OK 0:0000.0000.0007` or `ERR name not found`). This simplicity makes the nameserver easy to debug and interact with directly.

```
# Register a name
pilotctl name-register agent-alpha 0:0000.0000.0007

# Resolve it from any node on the network
pilotctl resolve agent-alpha
# => 0:0000.0000.0007
```

The Nameserver is bootstrapped by the node that runs it—it auto-registers nodes from the registry on first query. Custom names and service records are registered explicitly by agents.

4.6 Gateway

The Gateway bridges Pilot Protocol and standard IP. It maps virtual addresses to local IPs on a private subnet (e.g., `10.4.0.0/16`), allowing unmodified programs—browsers, `curl`, SSH clients—to reach agents through local IP addresses.

The Gateway operates as a userspace TCP proxy. For each mapped address, it automatically adds a loopback alias for the assigned IP and starts TCP listeners on a set of common ports (7, 80, 443, 1000–1002, 8080, 8443). When a local TCP connection arrives, the Gateway opens a Pilot Protocol stream to the corresponding virtual address and port, then performs bidirectional byte copying.

```
# Start the gateway, mapping one agent to a local IP
sudo gateway-linux run 0:0000.0000.0007

# Output:
# gateway: mapped 10.4.0.1 -> 0:0000.0000.0007
# gateway: proxy listening on 10.4.0.1:80
# gateway: proxy listening on 10.4.0.1:443
# ... (all common ports)

# Now reach the agent's HTTP server with standard tools:
curl http://10.4.0.1/status
# => {"status":"ok","protocol":"pilot","port":80}
```

Multiple agents can be mapped simultaneously, each receiving a unique IP from the subnet. The Gateway requires root privileges for ports below 1024 on Linux.

4.7 pilotctl

`pilotctl` is the command-line interface for agents to interact with the Pilot Protocol network. It communicates with the local daemon over the Unix socket. All commands return structured JSON—no interactive prompts, no human-oriented TUI. Every error includes a machine-readable code.

Identity & Discovery.

```
# Show daemon status (address, hostname, node ID, encryption)
pilotctl info

# Claim or clear a hostname
pilotctl set-hostname my-agent
pilotctl clear-hostname

# Find another agent by hostname
pilotctl find other-agent

# Control visibility (private by default)
pilotctl set-public
pilotctl set-private
```

Communication.

```
# Connect to a remote peer (stdio stream on port 1000)
pilotctl connect 0:0000.0000.0003

# Send a file via data exchange (port 1001)
pilotctl send-file 0:0000.0000.0003 ./report.pdf
```

Trust management.

```
# Request trust from another agent
pilotctl handshake other-agent "want to collaborate"

# Check for incoming trust requests
pilotctl pending

# Approve or reject
pilotctl approve 3
pilotctl reject 5 "not authorized"

# List trusted peers / revoke trust
pilotctl trust
pilotctl untrust 3
```

Diagnostics.

```
# Ping a remote node (4 echo RTT samples)
pilotctl ping 0:0000.0000.0003
```

```
# Trace connection setup + RTT
pilotctl traceroute 0:0000.0000.0003

# Throughput benchmark (default 1 MB, configurable)
pilotctl bench 0:0000.0000.0003

# Show connected peers with encryption status
pilotctl peers

# List active connections with per-connection stats
pilotctl connections

# Close a connection by ID
pilotctl disconnect 5

# Listen for incoming datagrams on a port
pilotctl listen 5000
```

Registry operations.

```
# Node registration, lookup, deregistration
pilotctl register
pilotctl lookup 3
pilotctl deregister

# Key rotation via signature or owner recovery
pilotctl rotate-key 3 agent-a@pilot
```

The `info` command provides a comprehensive view of daemon state, including identity status (ephemeral vs. persistent Ed25519), owner binding, encryption mode, authenticated peer count, and per-connection statistics (bytes, segments, retransmissions, SACK blocks, duplicate ACKs, congestion window, in-flight data, SRTT, RTTVAR, and recovery status). The `peers` command shows each peer's real endpoint, whether the tunnel is encrypted, and whether the key exchange was authenticated with Ed25519 signatures.

5. Security Model

Identity. Each node receives an Ed25519 keypair from the Registry upon registration. The private key is the node's identity token. The Registry holds all public keys and can verify identity. Identities can be persisted to disk (`-identity path/to/identity.json`) so that a node retains its keypair and virtual address across daemon restarts. On restart, the daemon re-registers with the stored public key and the registry recognizes the node, preserving its address and network memberships.

Owner binding and key rotation. Nodes can be bound to an owner identifier (typically an email) via the `-owner` flag. This enables key rotation recovery: if a node's private key is compromised or lost, the owner can request a new keypair from the registry. Key rotation supports two authentication paths: (1) signature-based, where the node signs a challenge (`rotate:<node_id>`) with its current Ed25519 private key, proving possession; or (2) owner-based, where the owner identifier is matched against the registry's records. After rotation, the registry issues a fresh Ed25519 keypair while preserving the node's ID and network memberships. Owner binding also enables reclaiming a deregistered node's identity—re-registering with the same owner email recovers the original node ID.

Trust boundaries. Network membership is the trust boundary. Joining a network requires satisfying its rules (open enrollment, token verification, or member invitation). Once inside a network, communication is unrestricted. This is a deliberate design choice: complex per-connection ACLs are replaced by simple group membership.

Integrity. Every packet carries a CRC32 checksum over header and payload. This detects corruption but not tampering—CRC32 is not cryptographic.

Encrypted channels. Virtual port 443 provides end-to-end encryption using X25519 Elliptic Curve Diffie-Hellman key exchange and AES-256-GCM authenticated encryption. When two agents connect on port 443, they perform an ECDH handshake to derive a shared secret, then use AES-256-GCM for all subsequent data. Each frame carries a unique 96-bit nonce derived from a monotonically increasing counter. GCM provides both confidentiality and integrity—an attacker cannot read or tamper with the payload without detection.

The encrypted frame format is:

`[4-byte length][12-byte nonce][ciphertext + 16-byte GCM tag]`

The secure channel is implemented entirely in Go's standard library (`crypto/ecdh`, `crypto/aes`, `crypto/cipher`) with no external dependencies.

Tunnel-layer encryption. The daemon enables tunnel encryption by default. On startup, it generates an ephemeral X25519 keypair. When two daemons first communicate, they exchange public keys over a dedicated key exchange packet (magic `PILK`). Both sides derive a shared secret via ECDH and establish an AES-256-GCM cipher. All subsequent packets between the pair are encrypted (magic `PILS`), regardless of virtual port.

The encrypted tunnel frame format is:

`[4-byte PILS magic][4-byte nodeID][12-byte nonce][ciphertext + 16-byte GCM tag]`

Tunnel encryption is backward-compatible: if a peer does not support encryption, communication falls back to plaintext. The `pilotctl peers` and `pilotctl info` commands display per-peer encryption status.

Authenticated key exchange. When a daemon has a persisted Ed25519 identity, the tunnel key exchange is upgraded from anonymous to authenticated. Instead of the standard `PILK` key exchange, the daemon sends a `PILA` (Pilot Authenticated) frame:

`[4-byte PILA magic][4-byte nodeID][32-byte X25519 pub][32-byte Ed25519 pub][64-byte signature]`

The signature covers a challenge constructed as `"auth"` + nodeID (4 bytes) + X25519 public key (32 bytes), signed with the node's Ed25519 private key. The receiving daemon verifies the signature using the peer's Ed25519 public key (fetched from the registry and cross-checked). This proves that the X25519 ephemeral key belongs to the claimed identity, preventing man-in-the-middle attacks where an adversary substitutes their own X25519 key. Unauthenticated `PILK` key exchange remains supported as a fallback for nodes without persistent identities.

Trust handshake protocol (port 444). Beyond tunnel-level authentication, Pilot Protocol provides an application-level trust handshake on virtual port 444. This allows nodes to establish explicit peer trust relationships with justifications, enabling fine-grained access control policies.

The handshake protocol supports three message types: `handshake_request` (with a justification string), `handshake_accept`, and `handshake_reject` (with a reason). Trust is established through three auto-approval paths:

1. **Mutual handshake.** If both nodes independently request a handshake with each other, the trust is auto-approved as mutual. This prevents social-engineering attacks where only one side initiates.
2. **Network trust.** If two nodes share a non-backbone network (same topic/group), the handshake is auto-approved. Network membership serves as a pre-existing trust signal.
3. **Manual approval.** If neither mutual nor network conditions are met, the request is queued as pending. The receiving node's operator can approve or reject it via `pilotctl approve <node_id>` or `pilotctl reject <node_id> <reason>`.

Trust records track the peer's node ID, public key, approval timestamp, whether the trust is mutual, and the network ID if approved via network membership. The `pilotctl trust` command displays all trusted peers, and `pilotctl pending` shows queued requests awaiting approval. Trust state is persisted to disk—approved peer relationships survive daemon restarts.

Endpoint privacy. Nodes are private by default. A node's physical IP:port (real address) is never exposed in `lookup` or `list_nodes` responses unless the node has explicitly opted into public visibility with the `-public` flag. To discover the physical endpoint of a private node via `resolve`, the requesting node must satisfy one of three conditions: (1) the target node is public, (2) the requester and target share a mutual trust pair registered in the registry, or (3) both nodes belong to a common non-backbone network. This prevents mass IP harvesting and ensures agents control who can reach them.

Backbone enumeration protection. Listing nodes on the backbone (network 0) is rejected by the registry. With 4 billion possible node IDs, the backbone is designed for scale—individual nodes are addressed by ID, not discovered by enumeration. Non-backbone networks allow listing since membership is the trust boundary.

Handshake relay. When a node is private, it cannot be reached directly for a handshake request (the requester does not know the IP). The registry acts as a trusted relay: the requester submits a handshake request to the registry, which stores it in the target node's inbox. The target polls its inbox during heartbeat cycles and can approve or reject via the registry. Approval creates a mutual trust pair, after which both nodes can resolve each other's endpoints and communicate directly. This ensures that no IP address is exposed until both parties have consented to the connection.

IPC socket permissions. The daemon's Unix domain socket is created with mode 0600, restricting access to the socket owner (typically root). This prevents unprivileged processes on the same machine from issuing commands to the daemon.

Admin token. The registry supports an optional admin token (`-admin-token`) that gates write operations (network creation, network joins/leaves, deregistration) while leaving read operations (lookups, list queries) open. Daemons pass the token via the `-admin-token` flag, which is included in authenticated requests.

Signed daemon operations. Write operations routed through the daemon (set-hostname, set-visibility, deregister) are signed with the node's Ed25519 private key. The registry verifies the signature before applying the mutation, preventing spoofed requests from unauthorized clients.

6. Compatibility

Pilot Protocol provides reliable byte streams and unreliable datagrams—the same primitives as TCP and UDP. Anything that runs on the internet can run on the overlay:

Protocol	Status	Tested
HTTP/1.0, HTTP/1.1	Works — Go's `net/http` with a Pilot listener	Yes
SSH	Works — TCP protocol over reliable stream	—
gRPC	Works — over HTTP/2	—
WebSocket	Works — over HTTP	—
Custom binary	Works — it is a byte pipe	Yes
DNS-style queries	Works — text protocol over streams	Yes

Table 3: Application-layer protocol compatibility. "Tested" indicates validated in integration tests and live deployment.

The Gateway makes this universal: any program on any machine, in any language, can talk to agents through mapped local IPs without knowing the overlay exists.

7. Well-Known Ports

Pilot Protocol defines a set of well-known virtual ports, analogous to IANA port assignments on the internet:

Port	Service	Description	Impl.
0	Ping / heartbeat	Liveness check	Yes
1	Control channel	Daemon status queries	Yes
7	Echo	Reflect incoming data (ping target)	Yes
53	Name resolution	Nameserver (A/N/S records)	Yes
80	Agent HTTP	Web UI and API endpoints	Yes
443	Secure channel	X25519 ECDH + AES-256-GCM encryption	Yes
444	Trust handshake	Peer trust negotiation (request/approve/reject)	Yes
1000	Standard I/O	Text stream between agents	Yes
1001	Data exchange	Typed frames (text/binary/JSON/file)	Yes
1002	Event stream	Pub/sub broker with topic filtering	Yes

Table 4: Well-known Pilot Protocol ports. "Impl." indicates whether the current reference implementation includes a service for this port.

Ports 1024–49151 are available for registered services. Ports 49152–65535 are ephemeral, allocated automatically for outbound connections.

8. Validation

The protocol has been validated with a live deployment spanning three geographic regions (US Central, US East, Europe West) on cloud infrastructure.

Role	Virtual Address	Services
Rendezvous	—	Registry, Beacon
Agent A	`0:0000.0000.0003`	Echo, HTTP, Nameserver, Secure, DataExch, Events
Agent B	`0:0000.0000.0004`	Echo, Gateway, test client

Table 5: Deployment topology.

8.1 Echo and Ping

Bidirectional echo tests confirm reliable data delivery across continents. The `ping` command—which dials port 7, sends a payload, and measures round-trip time—consistently shows ∼193 ms RTT between US East and Europe West:

```
$ pilotctl ping 0:0000.0000.0007
PING 0:0000.0000.0007
seq=0 bytes=12 time=193.289ms reply=echo: ping-0
seq=1 bytes=12 time=193.317ms reply=echo: ping-1
seq=2 bytes=12 time=193.405ms reply=echo: ping-2
seq=3 bytes=12 time=194.558ms reply=echo: ping-3
```

8.2 HTTP over Pilot Protocol

Agent A runs a Go `net/http` server bound to virtual port 80. Agent B connects through a Pilot Protocol stream. Standard HTTP request/response semantics work unmodified—the overlay is transparent to Go's HTTP stack.

```
$ curl http://10.4.0.1/status
{"status":"ok","protocol":"pilot","port":80}

$ curl http://10.4.0.1/
<!DOCTYPE html>
<html>
<head><title>Pilot Protocol</title></head>
<body>
<h1>Hello from Pilot Protocol</h1>
<p>This page is served over the Pilot Protocol overlay network.</p>
</body>
</html>
```

This was the most demanding integration test. Making Go's HTTP server work correctly over the overlay required implementing proper `SetReadDeadline` semantics (the server uses deadlines to cancel background reads) and correct connection close propagation (the daemon must notify the driver when the remote side sends FIN). Five distinct bugs in the IPC and driver layers were discovered and fixed during this validation.

8.3 Gateway

Agent B runs the Gateway in proxy mode, mapping Agent A's virtual address to local IP `10.4.0.1`. Standard tools (`curl`, browsers) connect to the local IP; the Gateway transparently bridges to the Pilot overlay. Five sequential HTTP requests completed successfully with no connection leaks.

8.4 Secure Channel

Agent A runs a secure echo server on virtual port 443. Agent B connects using X25519 ECDH key exchange, then exchanges AES-256-GCM encrypted messages:

```
$ ./secure-linux client 0:0000.0000.0003
secure: connected to 0:0000.0000.0003:443
secure: handshake complete (X25519 + AES-256-GCM)
> hello encrypted world
```

```
secure: echo: hello encrypted world
```

All data after the handshake is encrypted—the tunnel carries only ciphertext. The integration test verifies that a passive observer on the UDP tunnel cannot read the payload.

8.5 Data Exchange

Port 1001 implements a typed frame protocol for structured data exchange. Each frame carries a 4-byte type tag and a 4-byte length prefix, followed by the payload. Supported frame types include text, binary, JSON, and file transfer:

```
$ ./dataexchange-linux client 0:0000.0000.0003 \
    text "hello from data exchange"
data-exchange: sent text frame (24 bytes)
data-exchange: received text frame: hello from data exchange

$ ./dataexchange-linux client 0:0000.0000.0003 \
    json '{"key":"value","count":42}'
data-exchange: sent json frame (25 bytes)
data-exchange: received json frame: {"key":"value","count":42}
```

8.6 Event Stream

Port 1002 implements a publish/subscribe broker. Agents subscribe to topics and receive events published by any peer on the same network. The broker supports wildcard subscriptions (`*`) that match all topics:

```
# Agent A runs the event stream broker
$ ./eventstream-linux server

# Agent B subscribes and publishes
$ ./eventstream-linux client 0:0000.0000.0003
event-stream: connected to broker
> subscribe research.updates
event-stream: subscribed to research.updates
> publish research.updates "new paper available"
event-stream: event on research.updates: new paper available
```

8.7 Throughput Benchmark

The `pilotctl bench` command measures throughput by sending 1 MB through the echo server and timing both the send and the round-trip. Cross-continent results (Europe → US East → Europe):

```
$ pilotctl bench 0:0000.0000.0003
BENCH 0:0000.0000.0003 - sending 1.0 MB via echo port
  Sent:     1.0 MB in 371ms (2.7 MB/s)
  Echoed:   1.0 MB in 469ms (2.1 MB/s round-trip)
```

2.1 MB/s round-trip throughput across continental distances on minimal cloud instances. The IW10 initial window (RFC 6928), Appropriate Byte Counting (RFC 3465), SACK, delayed

ACKs with piggybacked ACK cancellation, and pure ACK-only duplicate detection (RFC 5681) maintain steady throughput with ∼96 ms RTT. Localhost benchmarks achieve ∼89 MB/s.

8.8 Graceful Shutdown

When a daemon receives SIGTERM, it:

1. Sends FIN to all active connections
2. Deregisters from the registry (node removed from address table)
3. Closes the IPC socket and UDP tunnel

The integration test verifies that after shutdown, the node is no longer found in the registry lookup.

8.9 Stress Test

A concurrency stress test opens 20 simultaneous connections from Agent B to Agent A's echo server. All 20 connections complete successfully with correct echoed payloads, validating that the daemon, IPC layer, and registry handle concurrent load without corruption or deadlock.

8.10 NAT Traversal Validation

The deployment was expanded to five nodes across five geographic regions to validate NAT traversal under real-world conditions. Three additional agents were placed behind Cloud NAT (no public IP, no port forwarding):

Role	Region	Network	NAT Type
Rendezvous	us-central1-a	Public IP	—
Agent Alpha	us-east1-b	Public IP	—
Agent Bravo	europe-west1-b	Public IP	—
NAT West	us-west1-b	Cloud NAT	EIM
NAT Asia	asia-east1-b	Cloud NAT	EIM
NAT South	southamerica-east1-b	Cloud NAT	EIM

Table 6: Extended deployment topology for NAT traversal validation. EIM = Endpoint-Independent Mapping.

All ten pairwise connections between the five agents established successfully. GCP Cloud NAT uses Endpoint-Independent Mapping, which means the STUN-discovered endpoint is valid for all peers—every connection was direct (no relay fallback required), including NAT→NAT paths.

Representative ping latencies across topology combinations:

From	To	RTT
Agent Alpha (US-East)	Agent Bravo (Europe)	∼194 ms
Agent Alpha (US-East)	NAT West (US-West)	∼136 ms
NAT West (US-West)	NAT Asia (Asia)	∼241 ms
NAT West (US-West)	NAT South (S. America)	∼341 ms

Table 7: Cross-continent ping latencies (4-probe average).

HTTP gateway bridging, pub/sub event streaming, data exchange, and handshake relay were all validated across Public→Public, Public→NAT, NAT→Public, and NAT→NAT topologies.

8.11 Summary

This validates the complete stack: address assignment, NAT traversal, tunnel establishment, session handshake, reliable data delivery with sliding window and congestion control, retransmission, flow control, write coalescing, automatic segmentation, tunnel encryption, authenticated key exchange, identity persistence, key rotation, peer trust handshakes (mutual and manual), endpoint privacy, handshake relay, backbone enumeration protection, HTTP compatibility (including trust-gated access), hostname discovery, typed data exchange, pub/sub event streaming, IP bridging, gateway proxying, zero-window probing, simultaneous close, connection limits, admin token gating, rate limiting, IPv6, graceful shutdown, connection lifecycle management, and concurrent load handling—all across continental distances over the public internet. The test suite comprises 247 integration and unit tests (223 pass, 24 skipped).

9. Operations

Structured logging. All components use Go's `log/slog` for structured, leveled logging. Log level (`-log-level debug|info|warn|error`) and format (`-log-format text|json`) are configurable at startup. JSON format is suitable for production log aggregation; text format is human-readable for development. Every log message carries contextual key-value fields (node IDs, addresses, connection states, error details).

Configuration files. All binaries accept a `-config path/to/config.json` configuration file. The config file uses standard JSON with keys matching command-line flag names (with hyphen-to-underscore normalization). Command-line flags take precedence over config file values, enabling a base configuration with per-invocation overrides.

IPv6 support. The Beacon's endpoint discovery and hole-punch coordination use variable-length IP encoding, supporting both 4-byte IPv4 and 16-byte IPv6 addresses. The wire format is `[type][ip_length][IP][port]`, and all components correctly handle IPv6 loopback (`[::1]`) and wildcard (`::`) addresses.

Trust state persistence. The Daemon's trust handshake manager persists approved peer relationships to disk. On restart, previously established trust relationships are restored without requiring re-negotiation.

10. Current Limitations

Registry replication is manual failover. The hot-standby replication system provides data redundancy, but failover requires manual intervention (restarting the standby without the `--standby` flag). Automatic leader election is not yet implemented.

No bandwidth estimation. The congestion window grows based on ACK feedback but does not use techniques like BBR or CUBIC that estimate the bottleneck bandwidth and path RTT separately. On links with deep buffers, the current AIMD approach may not fully utilize available capacity.

Handshake polling latency. Relayed handshake requests are delivered when the target node polls during its heartbeat cycle (every 30 s). This means a handshake request to a private node may take up to 30 s to be delivered. Real-time push notification for handshake delivery is not yet implemented.

Nameserver and custom networks. The Nameserver (port 53) and custom networks (non-backbone) are designed and partially implemented but not yet available through `pilotctl`.

Currently, all agents operate on network 0 (the global backbone). Network broadcast depends on custom networks and is also pending.

No connection migration. When an agent's underlying IP address changes (e.g., due to a network switch), existing connections break. Connection migration would allow agents to retain their virtual address and seamlessly resume connections from a new physical endpoint.

11. Future Work

Automatic failover. The hot-standby replication system requires manual promotion. Adding a heartbeat-based leader election protocol (e.g., Raft consensus) would enable automatic failover when the primary becomes unreachable.

Real-time handshake delivery. Currently, relayed handshake requests are delivered via polling during heartbeat cycles. A push-based notification system (e.g., long-polling or a dedicated notification channel) would reduce handshake delivery latency from seconds to milliseconds.

Gateway with TUN. The current Gateway operates as a userspace TCP proxy. A TUN-based implementation would route an entire IP subnet through the overlay at the kernel level, supporting UDP and ICMP in addition to TCP, with lower per-connection overhead.

Mobile agents. Support for agents that move between machines while retaining their virtual address—the daemon handles re-registration and tunnel migration.

Multi-hop routing. Currently, all agents must be able to reach each other directly (or through the beacon relay). Multi-hop routing would allow packets to traverse intermediate nodes, enabling mesh-like topologies.

12. Conclusion

The internet gave humans addresses, ports, and protocols. Pilot Protocol gives the same to agents. It is a minimal, practical overlay network that turns any machine running a daemon into a node on the agent internet. Agents get identities, agents get addresses, agents get ports. They can listen, they can connect, they can be reached. They are no longer squatters on the human internet. They are citizens of their own.

The protocol is implemented in Go with zero external dependencies, and running today.

Pilot Protocol is developed by Calin Teodor at Vulture Labs. The reference implementation is available at the project repository.

Test	**Path**	**Result**
Echo + Ping (ports 7, 1000)	Cross-continent	Pass (∼193 ms RTT)
HTTP (port 80)	Cross-continent	Pass (JSON + HTML)
Gateway + curl	Cross-continent	Pass (5/5 requests)
Secure channel (port 443)	Cross-continent	Pass (X25519 + AES-GCM echo)
Data exchange (port 1001)	Cross-continent	Pass (text + JSON frames)
Event stream (port 1002)	Cross-continent	Pass (pub/sub)
Throughput (1 MB echo)	Cross-continent	Pass (2.1 MB/s round-trip)
Stress test (20 conns)	Cross-continent	Pass (20/20 correct)
Sliding window + SACK	Local	Pass (89 MB/s, hash verified)
Segmentation + large writes	Local	Pass (64 KB seg, 8–32 KB writes)
Nagle coalescing	Local	Pass (100 writes in 4 ms)
Tunnel encryption + backward compat	Local	Pass (PILK/PILS/plaintext)
Authenticated key exchange	Local	Pass (Ed25519 signed)
Identity persistence + key rotation	Local	Pass (save/reload, sig/owner rotate)
Owner re-registration	Local	Pass (reclaim ID after deregister)
Handshake mutual auto-approve	Local	Pass (both request → auto)
Handshake approve/reject	Local	Pass (pending → approve/reject)
Handshake relay (private node)	Local	Pass (relay → approve → resolve)
Trust revoke + reestablish	Local	Pass (revoke → blocked → re-handshake)
Privacy: resolve blocked/allowed	Local	Pass (private blocked, public/trust allowed)
Backbone listing blocked	Local	Pass (network 0 → rejected)
Lookup hides real_addr	Local	Pass (private → no IP exposed)
Hostname (set, find, clear, persist)	Local	Pass (4 tests)
Hostname validation + uniqueness	Local	Pass (regex + duplicate rejected)
Visibility toggle	Local	Pass (public ↔ private)
HTTP with trust gating	Local	Pass (6 scenarios)
Connection limits + accept queue	Local	Pass (limits enforced, 30 conns)
Zero-window probing + simultaneous close	Local	Pass
Graceful shutdown + connection cleanup	Local	Pass (FIN + deregister)
Encrypted bidirectional + large transfer	Local	Pass (256 KB both directions)
Standby promotion under load	Local	Pass (failover, no data loss)
Re-registration after restart	Local	Pass (same identity preserved)
Admin token gating	Local	Pass (unauthorized ops rejected)
Secure channel (5 unit tests)	Unit test	Pass (handshake, bidir, large, nonces)
Frame + event wire format (4 tests)	Unit test	Pass (all types, size limits)
Gateway mapping (2 tests)	Unit test	Pass (auto/explicit, subnet)
Rate limiter (11 tests)	Unit test	Pass (token bucket, concurrent)
Config + IPv6	Local	Pass (JSON config, full IPv6 stack)
NAT traversal (STUN + hole-punch)	Cross-continent	Pass (5 regions, all direct)
NAT→NAT direct connection	Cross-continent	Pass (Cloud NAT EIM)
Gateway over NAT	Cross-continent	Pass (all 4 topologies)
Handshake relay over NAT	Cross-continent	Pass

Table 8: Cross-continent validation results.