Independent Submission
Internet-Draft
Intended status: Experimental
Expires: 8 October 2026

C. Teodor
Vulture Labs

6 April 2026

Pilot Protocol: An Overlay Network for Autonomous Agent Communication
draft-teodor-pilot-protocol-01

Abstract

This document specifies Pilot Protocol, an overlay network that provides autonomous AI agents with virtual addresses, port-based service multiplexing, reliable and unreliable transport, NAT traversal, encrypted tunnels, and a bilateral trust model. Pilot Protocol operates as a network and transport layer beneath application-layer agent protocols such as A2A and MCP. It encapsulates virtual packets in UDP datagrams for transit over the existing Internet. The protocol gives agents first-class network citizenship --- stable identities, reachable addresses, and standard transport primitives --- independent of their underlying network infrastructure.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 8 October 2026.

Table of Contents

1. Introduction

AI agents are autonomous software entities that reason, plan, and execute tasks. As agents become more prevalent, they need to communicate with each other across heterogeneous network environments: cloud, edge, behind NAT, and across organizational boundaries. Current agent protocols (MCP, A2A) operate at the application layer over HTTP, assuming agents have stable, reachable endpoints. This assumption fails for a large class of deployments.

Pilot Protocol is an overlay network stack that gives agents network-layer primitives: virtual addresses, ports, reliable streams, unreliable datagrams, NAT traversal, encrypted tunnels, name resolution, and a bilateral trust model. It is positioned as the network/transport layer beneath application-layer agent protocols --- analogous to how TCP/IP sits beneath HTTP.

1.1. Design Principles

The protocol is designed around five principles:

1. *Agents are first-class network citizens.* Every agent gets a unique virtual address, can bind ports, listen for connections, and be reached by any authorized peer.

2. *The network boundary is the trust boundary.* Network membership serves as the primary access control mechanism. Joining a network requires meeting its rules.

3. *Transport agnosticism.* The protocol provides reliable streams (TCP-equivalent) and unreliable datagrams (UDP-equivalent). Anything that runs on TCP/IP can run on the overlay.

4. *Minimize the protocol, maximize the surface.* The protocol defines addressing, packets, and transport. Application-level message formats are layers built on top.

5. *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.

1.2. Relationship to Existing Protocols

Pilot Protocol operates at the network and transport layers of the overlay stack. It is complementary to, not competitive with, application-layer agent protocols:

* A2A defines what agents say to each other. Pilot defines how they reach each other.

* MCP defines agent-to-tool interfaces. Pilot provides the transport substrate.

* QUIC [RFC9000] is a potential underlay transport. Pilot could run over QUIC instead of raw UDP.

* LISP [RFC9300] provides conceptual precedent for identity/locator separation.

* VXLAN [RFC7348] and GENEVE [RFC8926] are overlay encapsulation precedents at the data link layer. Pilot operates at the network layer.

* AGTP [I-D.hood-independent-agtp] and AIP [I-D.prakash-aip] address agent transport and identity at the application layer; Pilot provides the network substrate beneath them.

The CATALIST coordination effort [I-D.yao-catalist-problem-space-analysis] at IETF 125 surveyed the agent protocol landscape and is scoping what IETF should standardize in this space.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Agent: An autonomous software entity capable of reasoning, planning, and executing tasks without continuous human supervision.

Daemon: The local Pilot Protocol process that implements the virtual network stack. It maintains a UDP tunnel, handles routing, session management, and encryption. Analogous to a virtual NIC.

Driver: An SDK or library that agents import to communicate with the local daemon over IPC. Provides the application-facing API (listen, dial, read, write, close).

Registry: A centralized service that assigns virtual addresses, maintains the address-to-locator mapping table, manages network membership, and stores public keys.

Beacon: A service that provides NAT traversal coordination: endpoint discovery (STUN-like), hole-punch coordination, and relay fallback.

Virtual Address: A 48-bit overlay address assigned to an agent, independent of its underlying IP address.

Trust Pair: A bilateral trust relationship between two agents, established through explicit mutual consent.

3. Architecture Overview

3.1. Protocol Stack

Pilot Protocol is a five-layer overlay stack:

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

Table 1

The overlay handles addressing, routing, and session management. The underlying Internet handles physical delivery.

3.2. Component Roles

Registry: Assigns virtual addresses, maintains address table, manages networks and trust pairs, relays handshake requests for private nodes. Runs on TCP. The only globally reachable component.

Beacon: Provides STUN-like endpoint discovery, hole-punch coordination, and relay fallback for symmetric NAT. Runs on UDP.

Daemon: Core protocol implementation running on each participating machine. Maintains a single UDP socket, multiplexes all virtual connections, handles tunnel encryption, and exposes a local IPC socket for drivers.

Driver: Client SDK that agents import. Connects to the local daemon via Unix domain socket. Implements standard network interfaces (listeners, connections).

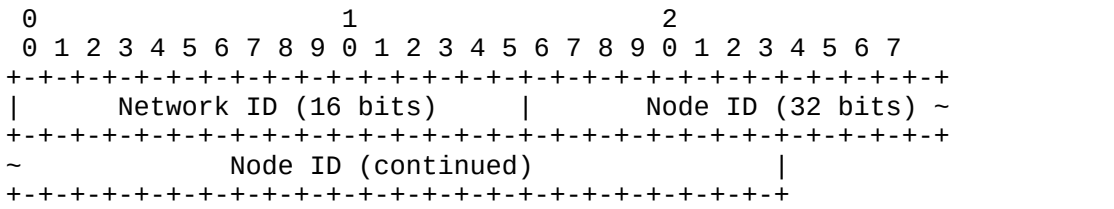
Nameserver: DNS equivalent for the overlay. Runs as a service on virtual port 53, resolving human-readable names to virtual addresses.

Gateway: Bridge between the overlay and standard IP. Maps virtual addresses to local IPs, allowing unmodified TCP programs to reach agents.

4. Addressing

4.1. Virtual Address Format

Addresses are 48 bits, split into two fields:



Network ID (16 bits): Identifies the network or topic. Network 0 is the global backbone; all nodes are members by default. Networks 1-65534 are created for specific purposes. Network 65535 is reserved.

Node ID (32 bits): Identifies the individual agent within a network. Supports over 4 billion nodes per network.

4.2. Text Representation

Addresses are written as N:XXXX.YYYY.ZZZZ where:

* N is the network ID in decimal
* XXXX.YYYY.ZZZZ is the node ID as three groups of 4 hexadecimal digits

Examples:

* 0:0000.0000.0001 --- Node 1 on the backbone
* 1:00A3.F291.0004 --- A node on network 1

4.3. Socket Addresses

A socket address appends a port: 1:00A3.F291.0004:1000

4.4. Special Addresses

Address	Meaning
0:0000.0000.0000	Unspecified / wildcard
0:0000.0000.0001	Registry
0:0000.0000.0002	Beacon
0:0000.0000.0003	Nameserver
X:FFFF.FFFF.FFFF	Broadcast on network X

Table 2

5. Ports

5.1. Port Ranges

Virtual ports are 16-bit unsigned integers (0-65535):

Range	Purpose
0-1023	Reserved / well-known
1024-49151	Registered services
49152-65535	Ephemeral / dynamic

Table 3

5.2. Well-Known Ports

Port	Service	Description
0	Ping	Liveness checks
1	Control	Daemon-to-daemon control
7	Echo	Echo service (testing)
53	Name resolution	Nameserver queries

80	Agent HTTP	Web endpoints
443	Secure channel	X25519 + AES-256-GCM
444	Trust handshake	Peer trust negotiation
1000	Standard I/O	Text stream between agents
1001	Data exchange	Typed frames (text, binary, JSON, file)
1002	Event stream	Pub/sub with topic filtering
1003	Task submission	Task lifecycle and reputation scoring

Table 4

6. Packet Format

6.1. Header Layout

The fixed packet header is 34 bytes:

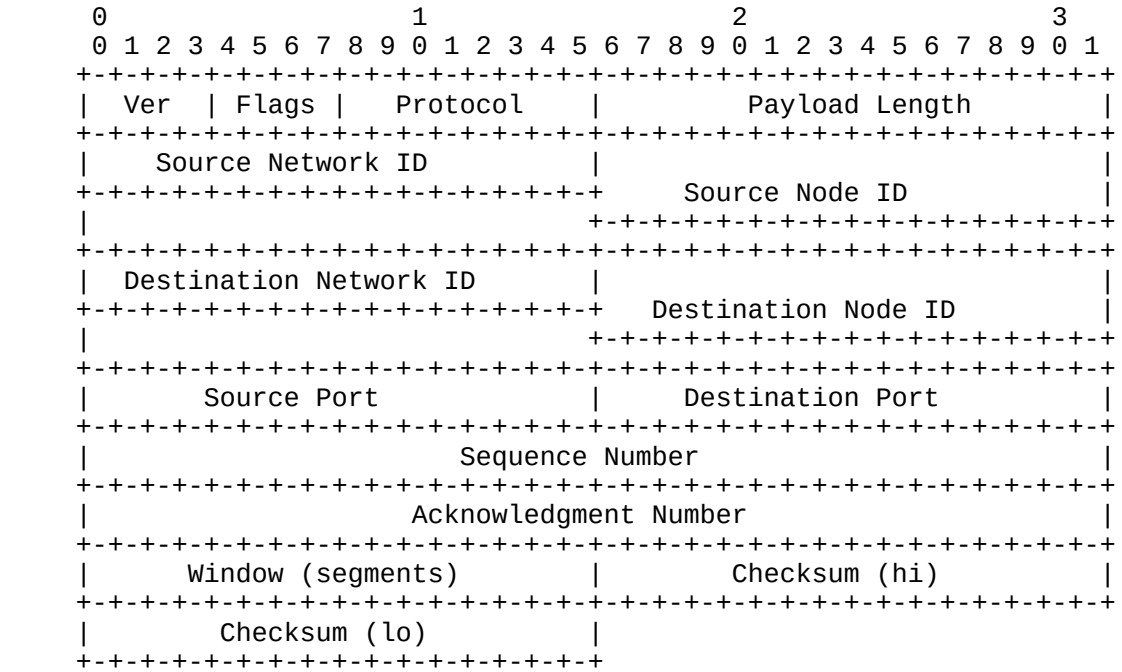


Figure 1: Pilot Protocol Packet Header (34 bytes)

All multi-byte fields are in network byte order (big-endian).

6.2. Field Definitions

Field	Offset	Size	Description
Version	0	4 bits	Protocol version. Current: 1
Flags	0	4 bits	SYN (0x1), ACK (0x2), FIN (0x4), RST (0x8)
Protocol	1	1 byte	Transport type (see Section 6.3)
Payload Length	2	2 bytes	Payload length in bytes (max 65535)
Source Network	4	2 bytes	Source network ID
Source Node	6	4 bytes	Source node ID
Dest Network	10	2 bytes	Destination network ID
Dest Node	12	4 bytes	Destination node ID
Source Port	16	2 bytes	Source port
Dest Port	18	2 bytes	Destination port
Sequence Number	20	4 bytes	Byte offset of this segment
Ack Number	24	4 bytes	Next expected byte from peer
Window	28	2 bytes	Advertised receive window in segments
Checksum	30	4	CRC32 over header + payload

		bytes	

Table 5

6.3. Protocol Types

Value	Name	Description
0x01	Stream	Reliable, ordered delivery (TCP-like)
0x02	Datagram	Unreliable, unordered (UDP-like)
0x03	Control	Internal control messages

Table 6

6.4. Flag Definitions

Bit	Name	Description
0	SYN	Synchronize --- initiate connection
1	ACK	Acknowledge --- confirm receipt
2	FIN	Finish --- close connection
3	RST	Reset --- abort connection

Table 7

6.5. Checksum Calculation

The checksum is computed as follows:

1. Set the 4-byte checksum field to zero.

2. Compute CRC32 (IEEE polynomial) over the entire header (34 bytes with zeroed checksum field) concatenated with the payload bytes.

3. Write the resulting 32-bit value into the checksum field in big-endian byte order.

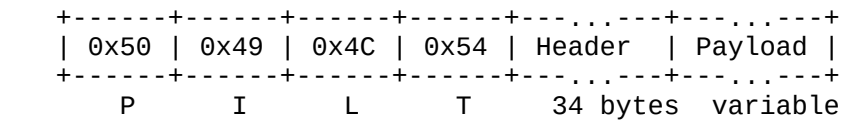
Receivers MUST verify the checksum and discard packets with incorrect values.

Note: CRC32 detects accidental corruption but does not provide cryptographic integrity. Tamper resistance is provided by tunnel-layer encryption (Section 7.2).

7. Tunnel Encapsulation

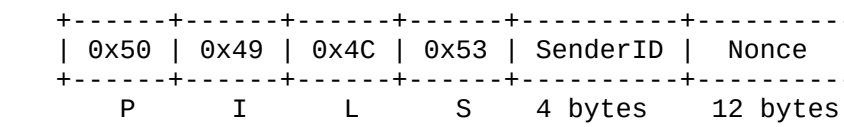
Virtual packets are encapsulated in UDP datagrams for transit over the real Internet. Four frame types are defined, distinguished by a 4-byte magic value.

7.1. Plaintext Frame (PILT)



The magic bytes 0x50494C54 ("PILT") indicate an unencrypted Pilot Protocol frame. The header and payload follow immediately.

7.2. Encrypted Frame (PILS)



The magic bytes 0x50494C53 ("PILS") indicate an encrypted frame.

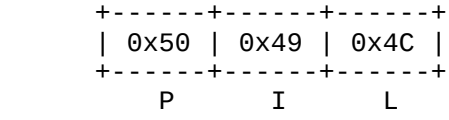
SenderID: 4-byte Node ID of the sending daemon, in big-endian. Used by the receiver to look up the shared AES-256-GCM key for this peer.

Nonce: 12-byte GCM nonce. See Section 11 for construction.

Ciphertext: The Pilot Protocol header and payload, encrypted with AES-256-GCM [RFC5116]. The ciphertext includes a 16-byte authentication tag appended by GCM.

The encryption key is derived from an X25519 [RFC7748] ECDH exchange between the two daemons (see Section 7.3), processed through HKDF [RFC5869] (see Section 10.3).

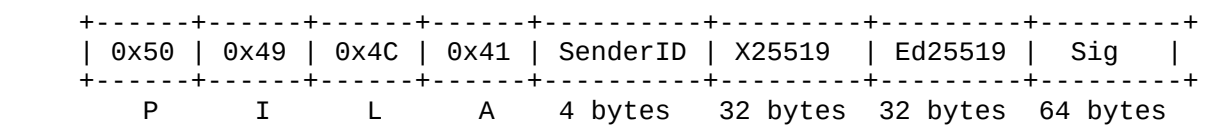
7.3. Key Exchange Frame (PILK)



Anonymous key exchange. The sender transmits its ephemeral X25519 public key (32 bytes, per [RFC7748]). Both sides compute the ECDH shared secret and derive an AES-256-GCM key.

PILK provides confidentiality but not authentication. An active attacker can perform a man-in-the-middle attack by substituting their own public key. See Section 7.4 for the authenticated variant.

7.4. Authenticated Key Exchange Frame (PILA)



Authenticated key exchange. In addition to the X25519 public key, the sender includes its Ed25519 public key (32 bytes, per [RFC8032]) and a 64-byte Ed25519 signature.

The signature covers the concatenation of:

1. The ASCII string "auth" (4 bytes)

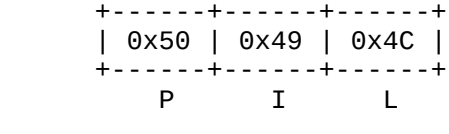
2. The sender’s Node ID (4 bytes, big-endian)

3. The X25519 public key (32 bytes)

The receiver verifies the signature using the sender’s Ed25519 public key, which it obtains from the registry and cross-checks against the claimed Node ID. This binds the ephemeral X25519 key to the sender’s persistent identity, preventing man-in-the-middle attacks.

Daemons with persistent Ed25519 identities SHOULD use PILA. Daemons without persistent identities fall back to PILK.

7.5. NAT Punch Frame (PILP)



The magic bytes 0x50494C50 ("PILP") indicate a NAT punch frame.

SenderID: 4-byte Node ID of the sending daemon, in big-endian.

The PILP frame carries no overlay payload. Its sole purpose is to create a NAT mapping by sending a UDP packet to the peer's observed endpoint during hole-punch coordination (Section 9.2). Receivers silently discard PILP frames after recording the sender's source address for subsequent direct communication.

8. Session Layer

8.1. Connection State Machine

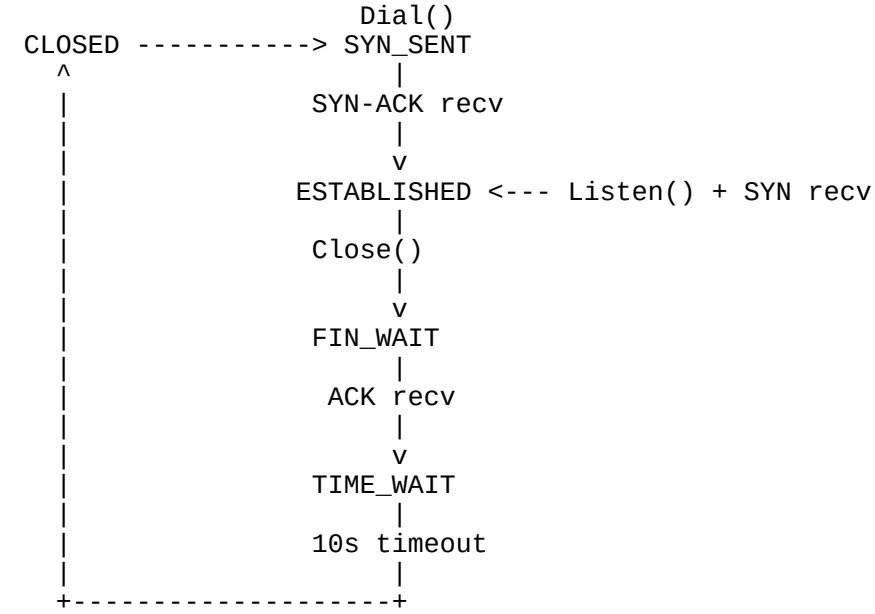
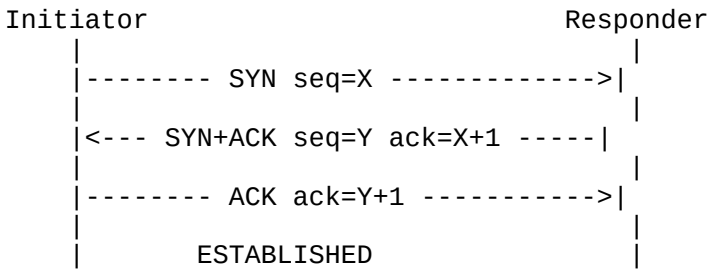


Figure 2: Connection State Machine

8.2. Three-Way Handshake

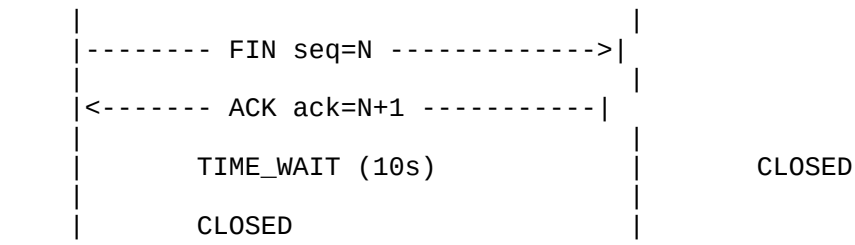
Connection establishment uses a three-way handshake:



The initiator selects an initial sequence number X. The responder selects its own initial sequence number Y and acknowledges X+1. The initiator confirms by acknowledging Y+1.

Both sides include their advertised receive window in the Window field of the SYN and SYN-ACK packets.

8.3. Connection Teardown



The closer sends FIN, waits for ACK, and enters TIME_WAIT for 10 seconds. The 10-second TIME_WAIT is shorter than TCP's typical 2*MSL because overlay RTTs are bounded by the underlay network.

8.4. Sequence Number Arithmetic

Sequence numbers are 32-bit unsigned integers with wrapping comparison per [RFC1982]:

```
seqAfter(a, b) = int32(a - b) > 0
```

This correctly handles wraparound at 2^32.

8.5. Reliable Delivery

The Stream protocol (0x01) provides reliable, ordered byte stream delivery using a sliding window mechanism.

8.5.1. Retransmission Timeout (RTO)

RTO is computed per [RFC6298]:

* SRTT (Smoothed RTT): updated with alpha = 1/8
* RTTVAR (RTT Variance): updated with beta = 1/4
* RTO = SRTT + max(G, 4 * RTTVAR)
* G (clock granularity floor) = 10 ms
* RTO is clamped to the range 200 ms to 10 s

SYN packets are retransmitted with exponential backoff: 1s, 2s, 4s, 8s, up to 5 retries. Data segments allow up to 8 retransmission attempts before the connection is closed.

8.5.2. Out-of-Order Handling

Segments received out of order are buffered and delivered to the application in sequence order when gaps are filled.

8.6. Selective Acknowledgment (SACK)

When the receiver has out-of-order segments, it encodes SACK blocks in the ACK payload. Each SACK block is a pair of 32-bit sequence numbers representing a contiguous range of received bytes beyond the cumulative ACK point. Up to 4 SACK blocks are encoded per ACK.

The sender uses SACK information to retransmit only the missing segments, skipping segments the peer has already received.

8.7. Congestion Control

The protocol implements TCP-style congestion control:

Slow Start: The congestion window (cwnd) starts at 10 segments (40 KB) per [RFC6928] and grows by one segment for each ACK received, until cwnd reaches the slow-start threshold (ssthresh).

Congestion Avoidance: After cwnd reaches ssthresh, growth switches to additive-increase: cwnd grows by approximately one segment per round-trip time (Appropriate Byte Counting per [RFC3465]).

Fast Retransmit: After 3 duplicate pure ACKs (data packets with piggybacked ACKs are excluded per [RFC5681] Section 3.2), the sender retransmits the missing segment without waiting for RTO.

Multiplicative Decrease: On loss detection (fast retransmit or RTO), ssthresh is set to max(cwnd/2, 2 segments). On RTO, cwnd is additionally reset to 1 segment (Tahoe behavior).

The maximum congestion window is 1 MB. The maximum segment size (MSS) is 4096 bytes.

8.8. 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:

effective_window = min(cwnd, peer_advertised_window)

This prevents a fast sender from overwhelming a slow receiver.

8.8.1. Zero-Window Probing

When the receiver advertises a zero window, the sender enters persist mode and sends 1-byte probe segments at exponentially increasing intervals until the receiver opens its window.

8.9. Write Coalescing (Nagle's Algorithm)

Small writes are buffered when unacknowledged data is in flight and flushed when:

* The buffer reaches MSS (4096 bytes), or
* All previous data is acknowledged, or
* A 40 ms timeout expires.

This reduces packet overhead for applications performing many small writes. The algorithm can be disabled per-connection with a NoDelay option, analogous to TCP_NODELAY.

8.10. Automatic Segmentation

Large writes are automatically segmented into MSS-sized chunks (4096 bytes) by the daemon. Applications can write arbitrarily large buffers without manual chunking.

8.11. 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.

8.12. Keepalive and Idle Timeout

Keepalive probes (empty ACKs) are sent every 30 seconds to idle connections. Connections idle for 120 seconds are automatically closed. These timers are appropriate for the overlay's use case (agent communication), where stale connections should be reclaimed promptly.

8.13. TIME_WAIT

Closed connections enter TIME_WAIT for 10 seconds before being removed. During TIME_WAIT, the connection occupies its port binding (preventing reuse confusion with delayed packets) but does not count as active.

9. NAT Traversal

9.1. STUN-Based Endpoint Discovery

On startup, the daemon sends a UDP probe to the beacon. The beacon observes the daemon's public IP address and port (as mapped by NAT) and reports it back. This follows the mechanism described in [RFC8489] (Session Traversal Utilities for NAT).

The discovered public endpoint is registered with the registry as the daemon's locator.

For daemons with known public endpoints (e.g., cloud VMs), the -endpoint host:port flag skips STUN and registers the specified endpoint directly.

9.2. Hole Punching

When daemon A wants to reach daemon B and both are behind NAT:

1. Daemon A sends a punch request to the beacon, specifying B's Node ID.

2. The beacon looks up B's registered endpoint and sends a punch command to both A and B, instructing each to send a UDP packet to the other's observed endpoint.

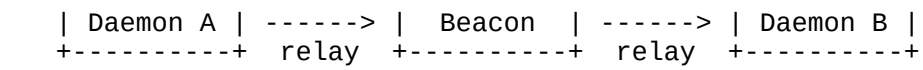
3. Both daemons send UDP packets to each other simultaneously, punching holes in their respective NATs.

4. Subsequent packets flow directly between A and B.

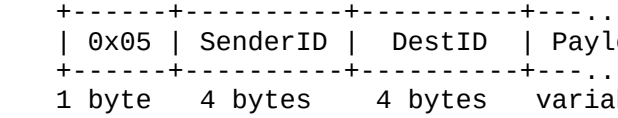
This works for Full Cone, Restricted Cone, and Port-Restricted Cone NAT types.

9.3. Relay Fallback

When hole punching fails (typically Symmetric NAT, where the mapped port changes per destination), the beacon provides transparent relay:



The relay frame format:



The beacon unwraps the relay header and forwards the payload to the destination daemon. Relay is transparent to the session layer --- the virtual packet inside the relay frame is identical to a directly-delivered packet.

9.4. Connection Establishment Strategy

When dialing a remote daemon, the connection strategy is:

1. Attempt 3 direct UDP sends to the peer's registered endpoint.

2. If all 3 fail, switch to relay mode through the beacon.

3. Attempt 3 relay sends.

4. If all relay attempts fail, return an error to the application.

The switch from direct to relay is automatic and transparent to the application layer.

10. Security

10.1. Identity

Each node receives an Ed25519 [RFC8032] keypair from the registry upon registration. The private key serves as the node's identity credential. The registry holds all public keys and can verify signatures.

Identities may be persisted to disk so that a node retains its keypair and virtual address across restarts. On restart with a persisted identity, the daemon re-registers with the stored public key and the registry restores the node's address and memberships.

10.2. Tunnel-Layer Encryption

Tunnel encryption is enabled by default. On startup, each daemon generates an ephemeral X25519 [RFC7748] keypair. When two daemons first communicate, they exchange public keys via PILK (Section 7.3) or PILA (Section 7.4) frames, compute an ECDH shared secret, and derive an AES-256-GCM [RFC5116] key via HKDF (Section 10.3).

All subsequent packets between the pair are encrypted (PILS frames), regardless of virtual port. The encryption is at the tunnel layer --- it protects all overlay traffic between two daemons, including connection handshakes.

The sender's Node ID (4 bytes, big-endian) is used as GCM Additional Authenticated Data (AAD). This binds the ciphertext to the sender's identity --- a decryption attempt with an incorrect sender ID will fail GCM authentication, preventing packet reattribution attacks.

Tunnel encryption is backward-compatible: if a peer does not respond to key exchange, communication falls back to plaintext (PILT frames).

10.3. HKDF Key Derivation

The shared secret from the X25519 ECDH exchange is processed through HKDF [RFC5869] to produce the AES-256-GCM key:

```
Extract:  PRK = HMAC-SHA256(salt=empty, IKM=shared_secret)
Expand:   key = HMAC-SHA256(PRK, info || 0x01)
```

For tunnel encryption, info is the ASCII string "pilot-tunnel-v1" (15 bytes). For application-layer encryption on port 443 (Section 10.5), info is the ASCII string "pilot-secure-v1" (15 bytes).

After the AES-256-GCM cipher is established, implementations MUST zero the shared secret, PRK, and derived key material in memory to limit the window of exposure if process memory is compromised.

10.4. Authenticated Key Exchange Upgrade

When a daemon has a persisted Ed25519 identity, the key exchange is upgraded from PILK to PILA (see Section 7.4). The Ed25519 signature binds the ephemeral X25519 key to the node’s persistent identity, preventing man-in-the-middle attacks.

Implementations SHOULD use PILA when an Ed25519 identity is available.

10.5. Application-Layer Encryption (Port 443)

Virtual port 443 provides end-to-end encryption between two agents, on top of any tunnel-layer encryption. The agents perform an X25519 ECDH handshake and derive a shared key via HKDF [RFC5869] with info string "pilot-secure-v1", then use AES-256-GCM for all subsequent data.

Each encrypted frame:

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

The sender’s nonce prefix (first 4 bytes of the nonce, see Section 11.4) is used as GCM Additional Authenticated Data (AAD). This binds ciphertext to the sender’s role (server or client), preventing cross-role confusion attacks.

This provides defense in depth: even if the tunnel encryption is compromised (e.g., by a compromised intermediate daemon in a future multi-hop topology), port 443 data remains protected.

10.6. Trust Handshake Protocol (Port 444)

Port 444 implements a bilateral trust negotiation protocol. Two agents exchange trust requests with justification strings and must both approve before a trust relationship is established.

Three auto-approval paths exist:

1. *Mutual handshake*: If both agents independently request trust with each other, the relationship is auto-approved.

2. *Network trust*: If both agents share a non-backbone network, the relationship is auto-approved (network membership serves as a trust signal).

3. *Manual approval*: If neither condition is met, the request is queued for the receiving agent's operator to approve or reject.

Trust pairs are recorded in the registry and persist across restarts. Trust is revocable: revoking trust immediately prevents further communication.

When trust handshake messages are relayed through the registry (for private nodes that cannot be reached directly), the sender MUST include an Ed25519 [RFC8032] signature over the concatenation of the ASCII string "handshake:", the sender's Node ID (decimal), ":", and the peer's Node ID (decimal). The receiver verifies the signature against the sender's public key as registered in the registry. This prevents handshake message forgery by a compromised relay path.

10.7. Privacy Model

Agents are private by default:

* A node's physical IP:port is never disclosed in registry responses unless the node has explicitly opted into public visibility.

* Resolving a private node's endpoint requires one of: (a) the node is public, (b) a mutual trust pair exists, or (c) both nodes share a non-backbone network.

* Listing nodes on the backbone (network 0) is rejected by the registry. Non-backbone networks allow listing since membership is the trust boundary.

10.8. Rate Limiting

The registry enforces per-connection sliding window rate limits using a token-bucket algorithm with per-source tracking. Clients that exceed the limit receive throttle responses.

Daemons implement SYN rate limiting to mitigate connection flood attacks.

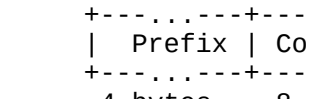
10.9. IPC Security

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

11. Nonce Management

11.1. Construction

AES-256-GCM nonces are 96 bits (12 bytes), constructed as:



Prefix: 4 bytes generated from a cryptographically secure random source when the tunnel session is established. Unique per session with overwhelming probability.

Counter: 8-byte unsigned integer, starting at 0, incremented by 1 for each packet encrypted. The counter MUST NOT be reset within a session.

11.2. Session Lifecycle

A new tunnel session is established when two daemons perform an X25519 key exchange (PILK or PILA). Each session produces:

* A fresh AES-256-GCM key (from the ECDH shared secret)

* A fresh random nonce prefix

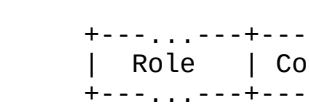
Since each session uses a different key, nonces from different sessions cannot collide (different keys are independent encryption contexts).

11.3. Counter Exhaustion

The 8-byte counter supports 2^64 encryptions per session. Implementations MUST re-key (initiate a new key exchange) before the counter reaches 2^64 - 1. In practice, at 1 million packets per second, counter exhaustion would take over 584,000 years.

11.4. Application-Layer Nonces (Port 443)

Secure connections on port 443 use a role-prefix nonce scheme:



Role Prefix: 0x00000001 for the server (listener) side, 0x00000002 for the client (dialer) side. Fixed per role for the lifetime of the connection.

Counter: 8-byte unsigned integer, starting at 0, incremented by 1 for each packet encrypted.

The role prefix prevents nonce collision between the two sides of a connection, even if both encrypt the same number of packets. Each connection has an independent counter and key derived from its own X25519 handshake and HKDF expansion.

12. Version Negotiation

12.1. Version Field

The 4-bit Version field in the packet header identifies the protocol version. The current version is 1. Version 0 is reserved and MUST NOT be used.

12.2. Handling Mismatches

The initiator includes its protocol version in the SYN packet. The responder checks the version:

* If supported: echoes the same version in SYN-ACK. Both sides use this version for the connection's lifetime.

* If unsupported: sends RST. No version downgrade negotiation occurs.

For non-SYN packets, if the Version field does not match the connection's established version, the packet is silently discarded. Implementations SHOULD log such events at debug level.

12.3. Future Extensibility

Future protocol versions MAY extend the header format. Implementations MUST NOT assume a fixed header size based solely on the Version field --- they SHOULD use the version to determine the expected header layout.

13. Path MTU Considerations

13.1. Encapsulation Overhead

The total per-packet overhead for encrypted tunnel frames is:

Component	Size
PILS magic	4 bytes
Sender Node ID	4 bytes
GCM nonce	12 bytes
Pilot header	34 bytes
GCM authentication tag	16 bytes
Total	*70 bytes*

Table 8

For plaintext frames (PILT), overhead is 38 bytes (4-byte magic + 34-byte header).

13.2. Effective Payload

With a typical 1500-byte Ethernet MTU, 20-byte IP header, and 8-byte UDP header:

* Available for Pilot: 1500 - 28 = 1472 bytes

* Encrypted payload capacity: 1472 - 70 = 1402 bytes

* Plaintext payload capacity: 1472 - 38 = 1434 bytes

13.3. MSS Selection

The default MSS of 4096 bytes exceeds single-packet capacity on standard Ethernet paths. Full-MSS segments will be fragmented into 3 IP fragments. This is acceptable on most networks but may fail on paths that block IP fragmentation.

Recommendations:

* For Internet-facing deployments where IP fragmentation may be blocked, an MSS of 1400 bytes avoids fragmentation on virtually all paths.

* For datacenter or local deployments (jumbo frames), the default 4096 MSS is appropriate.

* Implementations SHOULD provide a configurable MSS option.

* Implementations SHOULD NOT set the Don’t Fragment (DF) bit on UDP datagrams, allowing IP-layer fragmentation as a fallback.

14. Built-in Services

14.1. Echo (Port 7)

The echo service reflects any data received back to the sender. It is used for liveness testing (ping) and throughput benchmarking.

14.2. Data Exchange (Port 1001)

A typed frame protocol for structured data. Each frame carries a 4-byte type tag and a 4-byte length prefix:

Type	Value	Description
Text	0x01	UTF-8 text
Binary	0x02	Raw bytes
JSON	0x03	JSON document
File	0x04	File with name metadata

Table 9

14.3. Event Stream (Port 1002)

A publish/subscribe broker. Agents subscribe to named topics and receive events published by any peer. Wildcard subscriptions (*) match all topics. The wire protocol uses newline-delimited text commands:

* SUB <topic> --- subscribe to a topic

* PUB <topic> <payload> --- publish an event

* EVENT <topic> <payload> --- delivered event (broker to subscriber)

14.4. Task Submission (Port 1003)

A task lifecycle protocol. Agents submit tasks with descriptions, workers accept or decline, execute, and return results. A reputation score (polo score) adjusts based on execution efficiency.

15. Gateway

The gateway bridges the overlay network and standard IP networks, allowing unmodified TCP programs (curl, browsers, databases) to communicate with overlay agents.

15.1. Address Mapping

The gateway maps virtual addresses to local IP addresses on the loopback interface. For each mapped agent, the gateway creates a loopback alias:

* Linux: ip addr add <local-ip>/32 dev lo

* macOS: ifconfig lo0 alias <local-ip>

The default mapping subnet is 10.4.0.0/16. Each virtual address is assigned a unique local IP within this subnet.

15.2. Proxying

For each mapped address, the gateway listens on configurable TCP ports (default: 7, 80, 443, 1000, 1001, 1002, 8080, 8443) on the local IP. Incoming TCP connections are forwarded to the corresponding agent's virtual address and port via the daemon's Dial operation. Data flows bidirectionally between the TCP connection and the overlay stream.

This enables scenarios such as:

```
curl http://10.4.0.1:80/api     # reaches agent 0:0000.0000.0001 port 80
```

16. IPC Protocol

16.1. Framing

The daemon and driver communicate over a Unix domain socket using length-prefixed messages:

[4-byte big-endian length][message bytes]

Maximum message size: 1,048,576 bytes (1 MB).

16.2. Command Set

Cmd	Name	Direction	Description
0x01	Bind	Driver -> Daemon	Bind a virtual port
0x02	BindOK	Daemon -> Driver	Confirm port binding
0x03	Dial	Driver -> Daemon	Connect to remote agent
0x04	DialOK	Daemon -> Driver	Connection established
0x05	Accept	Daemon -> Driver	Incoming connection
0x06	Send	Driver -> Daemon	Send data on connection
0x07	Recv	Daemon -> Driver	Receive data
0x08	Close	Driver -> Daemon	Close connection
0x09	CloseOK	Daemon -> Driver	Connection closed
0x0A	Error	Daemon -> Driver	Error response
0x0B	SendTo	Driver -> Daemon	Send datagram
0x0C	RecvFrom	Daemon -> Driver	Receive datagram
0x0D	Info	Driver -> Daemon	Query daemon status
0x0E	InfoOK	Daemon -> Driver	Status response (JSON)
0x0F	Handshake	Driver -> Daemon	Trust handshake command
0x10	HandshakeOK	Daemon -> Driver	Handshake result (JSON)

0x11	ResolveHostname	Driver -> Daemon	Resolve hostname to address
0x12	ResolveHostnameOK	Daemon -> Driver	Resolution result (JSON)
0x13	SetHostname	Driver -> Daemon	Set discoverable hostname
0x14	SetHostnameOK	Daemon -> Driver	Hostname confirmation
0x15	SetVisibility	Driver -> Daemon	Set public/ private
0x16	SetVisibilityOK	Daemon -> Driver	Visibility confirmation
0x17	Deregister	Driver -> Daemon	Deregister from registry
0x18	DeregisterOK	Daemon -> Driver	Deregister confirmation
0x19	SetTags	Driver -> Daemon	Set node tags (JSON)
0x1A	SetTagsOK	Daemon -> Driver	Tags confirmation
0x1B	SetWebhook	Driver -> Daemon	Set event webhook URL
0x1C	SetWebhookOK	Daemon -> Driver	Webhook confirmation
0x1D	SetTaskExec	Driver -> Daemon	Enable/disable task exec
0x1E	SetTaskExecOK	Daemon -> Driver	Task exec confirmation
0x1F	Network	Driver -> Daemon	Network management
0x20	NetworkOK	Daemon -> Driver	Network result (JSON)

0x21	Health	Driver -> Daemon	Health check probe	
0x22	HealthOK	Daemon -> Driver	Health response (JSON)	

Table 10

16.3. Network Sub-Commands

The Network command (0x1F) uses a sub-command byte as the first byte of the payload:

Sub	Name	Payload	Description
0x01	List	(empty)	List joined networks
0x02	Join	network_id, token	Join a network
0x03	Leave	network_id	Leave a network
0x04	Members	network_id	List network members
0x05	Invite	network_id, node_id	Invite a node
0x06	PollInvites	(empty)	Poll pending invitations
0x07	RespondInvite	network_id, accept	Accept/reject invite

Table 11

17. Registry Replication

17.1. Hot-Standby Architecture

The registry supports high-availability through a hot-standby replication model. One registry instance operates as the primary (accepting reads and writes) while one or more standbys maintain synchronized copies of the state.

17.2. Push-Based Replication

Standbys connect to the primary via persistent TCP connections and subscribe for state updates. On any mutation (node registration, network creation, trust pair change, policy update), the primary pushes a full state snapshot to all connected standbys. The snapshot includes nodes, networks, trust pairs, and enterprise state.

17.3. Liveness and Failover

The primary sends heartbeat messages every 15 seconds. A standby that does not receive a heartbeat within 30 seconds considers the primary unavailable. Failover is manual: an operator promotes a standby to primary by restarting it in primary mode.

Standbys reject all write operations, returning an error directing the client to the primary. This prevents split-brain scenarios.

18. Enterprise Extensions

18.1. Role-Based Access Control

Networks support three roles:

Role	Permissions
Owner	Full control: delete network, transfer ownership, all admin operations
Admin	Manage members: invite, kick, promote, demote, set policy
Member	Participate: send/receive data, list members

Table 12

Role assignments are stored per-network in the registry. Operations are gated by minimum role requirement --- a member cannot modify policy, and an admin cannot delete the network. The network creator is automatically assigned the Owner role.

18.2. Network Policies

Each network may define policies that constrain member behavior:

* *max_members:* Maximum number of nodes in the network. The registry rejects join and invite operations when the limit is reached.

* *allowed_ports:* Per-network port allow-list (reserved for future enforcement at the daemon level).

Policies persist across registry restarts and are replicated to standbys.

18.3. Audit Trail

The registry maintains a ring-buffer audit log recording security-relevant operations:

* Node registration and deregistration

* Trust relationship creation and revocation

* Network membership changes (join, leave, kick, invite)

* Role assignments (promote, demote, ownership transfer)

* Policy modifications

* Key rotation and expiry events

* Enterprise flag changes

Each audit entry includes a timestamp, the action identifier, the actor's node ID, and for state mutations, both the old and new values. The audit log persists across registry restarts via atomic JSON snapshots.

An export API allows external systems (SIEM, compliance tools) to retrieve audit events. Webhook-based export with retry logic and a dead-letter queue ensures reliable delivery to external endpoints.

18.4. Identity Integration

The registry optionally supports external identity integration:

* *OIDC/JWT Validation:* Nodes may authenticate using JWT tokens from an OIDC provider. The registry validates RS256 signatures using cached JWKS (JSON Web Key Set) endpoints.

* *External Identity Mapping:* Nodes may carry an external_id field linking their overlay identity to an OIDC subject or directory entry.

* *Directory Synchronization:* Webhook-based synchronization with external identity providers enables centralized identity management.

19. Security Considerations

19.1. CRC32 Limitations

The packet checksum uses CRC32 (IEEE polynomial), which detects accidental corruption but provides no cryptographic integrity. An attacker who can modify packets in transit can recompute a valid CRC32. Integrity against active attackers is provided by tunnel-layer AES-256-GCM encryption, which MUST be used for all Internet-facing deployments.

19.2. Anonymous Key Exchange

The PILK key exchange frame provides no identity binding. An active man-in-the-middle attacker can substitute their own X25519 public key, establishing separate encrypted sessions with each peer. The PILA authenticated key exchange (Section 7.4) prevents this by binding the ephemeral key to an Ed25519 identity. Implementations SHOULD use PILA whenever an Ed25519 identity is available.

19.3. Registry Authentication

Write operations to the registry that modify node state (deregister, set visibility, set hostname, set tags, network operations) require either:

* An Ed25519 [RFC8032] signature from the node’s registered keypair, or

* A valid admin token (verified via constant-time comparison to prevent timing attacks).

Nodes without a registered public key MUST provide a valid admin token for any write operation.

19.4. TLS Certificate Pinning

Clients MAY pin the registry's TLS certificate fingerprint (SHA-256 hash of the DER-encoded certificate). When pinning is configured, the client performs manual certificate verification via the TLS VerifyPeerCertificate callback, rejecting connections whose certificate hash does not match the pinned value. This protects against compromised Certificate Authorities.

19.5. Registry as Trusted Third Party

The registry is a centralized trusted third party. Compromise of the registry could allow:

* Address hijacking (reassigning a node's virtual address)
* Locator spoofing (returning incorrect IP:port for a node)
* Public key substitution (enabling identity impersonation)
* Metadata harvesting (enumerating registered nodes)

Mitigations include TLS transport with optional certificate pinning, Ed25519 signature verification for node operations, admin token authentication for privileged operations, hot-standby replication for availability, and persistent audit logging for forensic analysis. Future work should explore distributed registry designs with consensus-based replication.

19.6. GCM Nonce Uniqueness

AES-256-GCM security depends critically on nonce uniqueness under the same key. The nonce construction (Section 11) guarantees uniqueness through a random prefix (unique per session) and a monotonic counter (never reset within a session). Since each key exchange produces a new key, nonces from different sessions are in independent cryptographic contexts.

Implementations MUST NOT reuse nonces. Implementations MUST NOT reset the counter within a session. Implementations MUST re-key before counter exhaustion.

19.7. Metadata Exposure

Even with tunnel encryption (PILS), the sender's Node ID is transmitted in cleartext (it is needed for the receiver to look up the decryption key). This allows a passive observer to determine which daemons are communicating, though the content and virtual addressing within the encrypted payload remain confidential.

19.8. Double Congestion Control

Pilot Protocol implements congestion control at the overlay layer, while the underlay UDP-over-IP path may also be subject to network-level congestion signals (ICMP source quench, ECN). The overlay congestion control operates independently, which may lead to suboptimal behavior on heavily congested paths. This is a known issue shared with all overlay transport protocols.

19.9. Replay Protection

Implementations MUST maintain a sliding-window replay bitmap for each peer's tunnel session. The recommended window size is 256 nonces. The replay check operates as follows:

1. If the nonce counter is below the window's lower bound (more than 256 positions behind the highest seen counter), the packet is rejected.

2. If the nonce counter falls within the window and the corresponding bitmap bit is set, the packet is a replay and is rejected.

3. If the nonce counter is within the window and the bit is not set, the bit is set and the packet is accepted.

4. If the nonce counter is ahead of the window, the window slides forward and the packet is accepted.

This provides replay protection while tolerating out-of-order packet delivery within the window size.

19.10. IPC as Trust Boundary

The Unix domain socket IPC between daemon and driver is a trust boundary. The daemon trusts that any process connecting to the socket is authorized (enforced by filesystem permissions, mode 0600). If an attacker gains access to the socket, they can impersonate the local agent. Deployments SHOULD ensure the daemon runs under a dedicated user account.

20. IANA Considerations

20.1. Pilot Protocol Tunnel Magic Values

This document requests the creation of a "Pilot Protocol Tunnel Magic Values" registry with the following initial entries:

Magic	Hex	Description
PILT	0x50494C54	Plaintext frame
PILS	0x50494C53	Encrypted frame
PILK	0x50494C4B	Key exchange frame
PILA	0x50494C41	Authenticated key exchange frame
PILP	0x50494C50	NAT punch frame

Table 13

20.2. Pilot Protocol Type Values

This document requests the creation of a "Pilot Protocol Type Values" registry with the following initial entries:

Value	Name	Description
0x01	Stream	Reliable, ordered delivery
0x02	Datagram	Unreliable, unordered delivery
0x03	Control	Internal control messages

Table 14

20.3. Pilot Protocol Well-Known Ports

This document requests the creation of a "Pilot Protocol Well-Known Ports" registry with the following initial entries:

Port	Service	Description
0	Ping	Liveness checks
1	Control	Daemon-to-daemon control
7	Echo	Echo service
53	Name Resolution	Nameserver
80	Agent HTTP	Web endpoints
443	Secure	End-to-end encrypted channel
444	Trust	Trust handshake protocol
1000	StdIO	Text stream
1001	DataExchange	Typed frame protocol
1002	EventStream	Pub/sub broker
1003	TaskSubmit	Task lifecycle

Table 15

21. Implementation Status

Per [RFC7942], this section documents the known implementations of Pilot Protocol at the time of writing.

21.1. Go Reference Implementation

Organization: Vulture Labs

Description: Complete implementation of Pilot Protocol including daemon, driver SDK, registry, beacon, nameserver, gateway, and CLI (pilotctl). Implemented in Go with zero external dependencies.

Level of maturity: Production deployments in experimental environments.

Coverage: All features specified in this document are implemented, including tunnel encryption (PILK/PILA/PILS/PILP), HKDF key derivation, sliding-window replay detection, SACK, congestion control, flow control, Nagle’s algorithm, automatic segmentation,

NAT traversal (STUN, hole-punch, relay), trust handshake protocol with Ed25519 relay signing, privacy model, registry high-availability replication, enterprise RBAC and audit trail, OIDC/JWT identity integration, gateway bridge, per-port connection limits, and all built-in services.

Testing: 983 tests. Integration tests validated across 5 GCP regions (US Central, US East, Europe West, US West, Asia East) with public-IP, NAT-only, and symmetric-NAT topologies.

Licensing: Proprietary.

Contact: teodor@vulturelabs.com

21.2. Python SDK

Organization: Vulture Labs

Description: Python client SDK using ctypes FFI to the Go shared library. Published on PyPI as pilotprotocol.

Level of maturity: Production-ready.

Coverage: Driver operations (dial, listen, accept, send, receive, close), datagram support, info queries.

Licensing: Proprietary.

Contact: teodor@vulturelabs.com

21.3. Node.js SDK

Organization: Vulture Labs

Description: TypeScript client SDK with FFI bindings to the Go shared library. Published on npm as pilotprotocol.

Level of maturity: Production-ready.

Coverage: Driver operations (dial, listen, accept, send, receive, close), datagram support, info queries. Full TypeScript type definitions.

Licensing: Proprietary.

Contact: teodor@vulturelabs.com

22. References

22.1. Normative References

[RFC1982] Elz, R. and R. Bush, "Serial Number Arithmetic", RFC 1982, DOI 10.17487/RFC1982, August 1996, <https://www.rfc-editor.org/rfc/rfc1982>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/rfc/rfc5116>.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009, <https://www.rfc-editor.org/rfc/rfc5681>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/rfc/rfc5869>.

[RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, DOI 10.17487/RFC6298, June 2011, <https://www.rfc-editor.org/rfc/rfc6298>.

[RFC6928] Chu, J., Dukkipati, N., Cheng, Y., and M. Mathis, "Increasing TCP's Initial Window", RFC 6928, DOI 10.17487/RFC6928, April 2013, <https://www.rfc-editor.org/rfc/rfc6928>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/rfc/rfc7748>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/rfc/rfc8032>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8489] Petit-Huguenin, M., Salgueiro, G., Rosenberg, J., Wing, D., Mahy, R., and P. Matthews, "Session Traversal Utilities for NAT (STUN)", RFC 8489, DOI 10.17487/RFC8489, February 2020, <https://www.rfc-editor.org/rfc/rfc8489>.

22.2. Informative References

[I-D.hood-independent-agtp]
Hood, C., "Agent Transfer Protocol (AGTP)", 2026, <https://datatracker.ietf.org/doc/draft-hood-independent-agtp/>.

[I-D.prakash-aip]
Prakash, S., "Agent Identity Protocol (AIP)", 2026, <https://datatracker.ietf.org/doc/draft-prakash-aip/>.

[I-D.yao-catalist-problem-space-analysis]
Zhou, Y. and K. Yao, "Problem Space Analysis of AI Agent Protocols in IETF", 2026, <https://datatracker.ietf.org/doc/draft-yao-catalist-problem-space-analysis/>.

[RFC3465] Allman, M., "TCP Congestion Control with Appropriate Byte Counting (ABC)", RFC 3465, DOI 10.17487/RFC3465, February 2003, <https://www.rfc-editor.org/rfc/rfc3465>.

[RFC7348] Mahalingam, M., Dutt, D., Duda, K., Agarwal, P., Kreeger, L., Sridhar, T., Bursell, M., and C. Wright, "Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks", RFC 7348, DOI 10.17487/RFC7348, August 2014, <https://www.rfc-editor.org/rfc/rfc7348>.

[RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", BCP 205, RFC 7942, DOI 10.17487/RFC7942, July 2016, <https://www.rfc-editor.org/rfc/rfc7942>.

[RFC8926] Gross, J., Ed., Ganga, I., Ed., and T. Sridhar, Ed., "Geneve: Generic Network Virtualization Encapsulation", RFC 8926, DOI 10.17487/RFC8926, November 2020, <https://www.rfc-editor.org/rfc/rfc8926>.

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/rfc/rfc9000>.

[RFC9300] Farinacci, D., Fuller, V., Meyer, D., Lewis, D., and A. Cabellos, Ed., "The Locator/ID Separation Protocol (LISP)", RFC 9300, DOI 10.17487/RFC9300, October 2022, <https://www.rfc-editor.org/rfc/rfc9300>.

Appendix A. Acknowledgments

The author thanks the participants of the IETF AI protocols discussions for their contributions to the understanding of the agent communication landscape.

Appendix B. Wire Examples

B.1. SYN Packet

A SYN packet from 0:0000.0000.0001 port 49152 to 0:0000.0000.0002 port 1000, with no payload:

```
Byte  0:    0x11         (version=1, flags=SYN)
Byte  1:    0x01         (protocol=Stream)
Byte  2-3:  0x0000       (payload length=0)
Byte  4-5:  0x0000       (src network=0)
Byte  6-9:  0x00000001   (src node=1)
Byte 10-11: 0x0000       (dst network=0)
Byte 12-15: 0x00000002   (dst node=2)
Byte 16-17: 0xC000       (src port=49152)
Byte 18-19: 0x03E8       (dst port=1000)
Byte 20-23: 0x00000000   (seq=0)
Byte 24-27: 0x00000000   (ack=0)
Byte 28-29: 0x0200       (window=512 segments)
Byte 30-33: [CRC32]      (computed over header)
```

Total: 34 bytes.

B.2. Data Packet

An ACK data packet with 5-byte payload "hello":

```
Byte  0:    0x12         (version=1, flags=ACK)
Byte  1:    0x01         (protocol=Stream)
Byte  2-3:  0x0005       (payload length=5)
...
Byte 28-29: 0x01F6       (window=502 segments)
Byte 30-33: [CRC32]      (computed over header + payload)
Byte 34-38: 0x68656C6C6F ("hello")
```

Total: 39 bytes.

B.3. Encrypted Tunnel Frame

A PILS frame carrying an encrypted Pilot packet:

```
Byte  0-3:  0x50494C53   (magic="PILS")
Byte  4-7:  0x00000001   (sender node ID=1)
Byte  8-19: [12-byte nonce]
Byte 20+:   [ciphertext + 16-byte GCM tag]
```

Author’s Address

Calin Teodor
Vulture Labs
Email: teodor@vulturelabs.com