Directed Messaging

Ted Blackman ~rovnys-ricfer,*
Joe Bryan ~master-morzod,†
Luke Champine ~watter-parter,*
Pyry Kovanen ~dinleb-rambep,†
Jose Cisneros ~norsyr-torryn,†
Liam Fitzgerald ~hastuc-dibtux‡
* Martian Engineering,
† Tlon Corporation,
‡ Axiomatic Systems

Abstract

Directed Messaging is a redesign of Urbit's networking protocol achieving over 100× throughput improvements while implementing content-centric networking in a production peer-to-peer system. Unlike address-based routing, all network operations (queries and commands) are expressed as remote namespace reads. Urbit's immutable scry namespace enables efficient caching, deterministic encryption, and stateless publishing, while a pre-distributed PKI eliminates handshake overhead for single-roundtrip transactions. Lockstep Streaming, a novel scale-invariant packet authentication scheme using binary numeral trees, maintains authentication integrity across variable MTU sizes at relay hops. The Lackman traversal pattern enables constant-space streaming authentication. Directed routing simplifies peer discovery and NAT traversal compared to previous approaches, while source-independent routing minimizes relay state. Begun in 2023 under the auspices of the Urbit Foundation

Manuscript submitted for review.

Address author correspondence to ~rovnys-ricfer.

and deployed in early 2025, Directed Messaging represents the first large-scale deployment of content-centric networking.

Contents

1 Introduction

The Directed Messaging project was a fundamental overhaul of Urbit's networking stack. It rewrote the protocol definition and the protocol implementation, split between Hoon code inside Arvo (the Urbit overlay OS), and C code in Vere (the Urbit runtime). Directed Messaging addressed several major limitations of Urbit's previous networking stack: it increased throughput by over 100×, improved peer discovery reliability, enabled the scalability of content delivery, and introduced a modular internal architecture that reduced implementation complexity. Directed Messaging is an encrypted, authenticated, peer-to-peer, packet-switched, message-oriented, connectionless, content-centric, transactional network protocol with its own congestion control, transmission control, and packet-level authentication. It was deployed to the Urbit network in early 2025.

These improvements were driven by Directed Messaging's total adherence to a request-response discipline throughout the stack, bundled with heavy use of Urbit's immutable referentially-transparent global namespace, called the "scry namespace". Every network message is either a request for data at a "scry path" (Urbit's equivalent of a URL), or an authenticated response that includes the data at that path. This is true at the message layer and the packet layer, and for both reads (queries) and writes (commands).

Before Directed Messaging, the bandwidth Urbit was able to utilize was extremely limited, maxing out in the hundreds of kilobytes per second. It lacked orders of magnitude of performance in order to make effective use of commodity networking hardware, such as on a laptop or cloud instance. With Directed Messaging, Urbit's networking speed was able to reach over 1 gigabit/s on commodity hardware, sufficient for the vast majority of contemporary personal use cases.

Directed Messaging managed to improve throughput while preserving Urbit's already-good latency performance. For small messages, a network transaction is accomplished in one roundtrip: a command sent one way, followed by an acknowledgment the other way. Due to Urbit's public key infrastructure (PKI) being disseminated to all nodes *a priori* from a Byzantine fault-tolerant source (the Ethereum blockchain and a bespoke L2 for economic efficiency), no cryptographic handshake is needed for two Urbit nodes (called "ship"s in Urbit lingo) to begin communication – even their first exchange is a single roundtrip.

In addition to performance improvements, Directed Messaging scales better by leveraging runtime caching to disseminate data to many clients simultaneously. This strategy is particularly effective within Urbit's immutable scry namespace. Directed Messaging also introduces a new procedure for peer discovery and routing that is much more reliable and performant than the previous setups. It deals better with NAT traversal and with using supernodes as relays more efficiently and reliably.

All of this is done by simplifying the basic networking structure to enforce a rigid request-response discipline throughout the entire system. In addition to that, it also places all network messages, including acknowledgments and commands, in addition to responses for queries asking for data, into Urbit's immutable namespace, called the scry namespace. This makes Urbit's entire networking stack a named data networking system, which is also called content-centric networking. As far as the authors know, Directed Messaging is the very first production deployment of a content-centric networking protocol, as well as the deployment with the largest number of nodes. Not only that, its content-centricity preserves the immutability of Urbit's namespace throughout the stack. The immutability is key to the scalability improvements and reliability improvements, and it also helps with single-threaded performance.

Along the way, we designed and implemented a novel scheme for streaming packet authentication. This helps prevent denial-of-service attacks that could forge individual packets and spoof them in order to invalidate a large download. This attack is prevented by authenticating every packet, but unlike the previous version of Urbit's networking stack, which authenticated each packet with a signature (meaning one signature for every kilobyte of data, which was extremely inefficient), there's only one signature per message. A Merkelization scheme using binary numeral trees (TODO: cite) is used to authenticate the packets within that message.

This achieves a property that, to our knowledge, has not been demonstrated in prior packet authentication schemes: the ability to handle a different maximum transmission unit (MTU) at each hop in a relay chain without losing packet-level authentication. It is a scale-invariant packet-authentication scheme, and it also has good memory use characteristics, due to a novel algorithm based on the "Lackman" scale-invariant tree-traversal pattern developed by the authors. A relay receiving packets of one size and emitting packets of another size only needs to hold one large packet in memory while streaming (e. g. if receiving 1 kiB packets and sending 16kiB packets, it only needs to store 16 kiB of packet data at a time). It can seamlessly handle any MTU that is one kilobyte, two kilobytes, or any power-of-two number of kilobytes above that, up to 128 MiB.

Another advantage of the Directed Messaging protocol is that much more of the logic can be offloaded from the Urbit operating system, Arvo, into the runtime. This enables decoupling between the formal specification, which is written in Nock, and implementation, which is written in C. This is in keeping with the spirit of Urbit's "jet" system that separates mechanism and policy for code execution. The most straightforward advantage of this decoupling is that each packet can be processed ephemerally, without incurring a disk write as in previous versions of Urbit's network protocol – that was a severe bottleneck on the maximum throughput. The implementation in the runtime could be swapped out with another implementation written in another language, the congestion control algorithm could be swapped out, and parallelism strategies could readily be employed to increase multicore CPU utilization. The implementation that was deployed contains a major optimization: specialized arena allocators to reduce memory management overhead.

2 High-Level Protocol Design

2.1 Request/Response, Namespace Reads

The protocol design is based off the idea of a remote namespace read, wherein a "subscriber" ship requests data from a "publisher" ship, and the publisher sends that data as the response. The publisher ship makes the data available over the network by assigning it a path in Urbit's "scry" namespace and setting permissions appropriately (permissioning will be described fully in a later section). The subscriber ship, then, can download data from another ship by sending it a request for the data at a path and waiting for the response.

A network roundtrip is conceived of as a request, followed by a response. The request consists of a "scry request", i. e. a request for data at a "scry path". An example scry path is `/~zod/1/2/c/x/~2025.9.22/sys/kelvin`. A scry path immutably names a datum, in this case the `sys.kelvin` file published by `~zod` (at `rift=1` and `life=2`), within the `%c` kernel module (Clay, the revision control system), with request type `%x` (request file contents), and with timestamp at the start of the day on September 22, 2025. When a ship receives a scry request over the network, it can respond by sending a "scry response" containing the datum bound to that path.

Over the network, a network request is a single UDP packet that encodes a scry path, limited to 384 characters so the request packet always fits into the internet-standard 1500-byte MTU (maximum transmission unit). A scry response may consist of a single packet, or multiple packets, depending on the size of the datum bound to that path. If the datum is 1kiB or less, the response is encoded into a single UDP packet. Otherwise, the first scry response packet contains only the first 1kiB of the datum, along with a fixed-width field containing the number of bits in the whole datum.

If the subscriber ship receives a response indicating the datum is multi-packet, it switches modes and begins requesting the remainder of the datum, one kiB at a time. Each request for a kiB of data is itself a fully-fledged namespace read request, where the path in the request contains the path of the whole datum as well as the chunk size being requested (configured to 1 kiB over the public internet, but this could be increased to any power of 2 kiB's up to 128 MiB, for other environments, such as intra-datacenter). The protocol definition does not require those fragment requests to be sent in any particular order or according to any particular congestion control algorithm. The current implementation uses a packet-switched variant of the TCP NewReno congestion control algorithm, written in C in Urbit's runtime, to manage multi-packet reads. The message-level authentication is sent in the first response packet. Each subsequent packet is authenticated as being part of that message by using the LockStep streaming authentication scheme, described in a later section.

This remote read flow is a "pure read": handling a read request does not require the publisher ship to change any persistent state. But a general-purpose network protocol needs to be able to express commands, not just reads. Directed Messaging builds commands out of reads. A command is conceived of as two reads, one in each direction:

1. The ship sending the command makes the command datum available within its own scry namespace, so the receiving ship has the ability to read the command by sending a remote read request.

2. The ship that receives the command, after attempting

to execute the command, makes the command's result available within its namespace, so the sending ship has the ability to read the result (hereafter called the "ack") by sending a remote read request. A result can be success ("ack") or an error datum ("naxplanation", named after "nack" for negative acknowledgment).

This approach is conceptually clean but immediately presents two practical challenges. The first is triggering: how does the receiving ship *know* to request the command datum from the sending ship? There are many ships on the network; it would be absurdly impractical to send requests to all of them on the off-chance that one or two of them have an outstanding command for us. The second challenge is latency: a naive implementation would imply every command requires two network roundtrips, one to remote-read the command and one to remote-read the command's result (ack or naxplanation). If so, that would be unfortunate, since Urbit's previous networking required only one roundtrip for a command and ack, in the common case of a small ($\leq$ 1 kiB) command datum, and unneeded roundtrips are anathema to a good user experience (Cheshire, 1996).

Fortunately, we can solve both problems with one weird trick. We add a 'request-type' bit to each network request packet indicating whether it is a read or a command, and if it is a command, it includes not only the scry request path, but also a scry response containing the first 1 kiB of the command datum. When the receiving ship's runtime receives the packet, it looks at the 'request-type' bit to determine how to handle the packet.

If the incoming request packet is a read, the runtime performs a read request on the Arvo OS by firing its `+peek` arm, a Nock function that reads from Arvo's namespace. This read request does not trigger any disk writes and could be run in parallel with other reads and with an Arvo event. The runtime then encodes the result of this read as a scry response packet and sends it back to the IP and port that sent the request.

If the packet is a command, the runtime injects the packet as a stateful Arvo "event" by firing its `+poke` arm (a Nock function that sends an event or command for Arvo to process, producing effects and a new Arvo OS with a modified state). When this event completes, one of the effects it produces can be a scry response packet containing the ack, which the runtime will send back to the IP and port that sent the request.

If the command datum fits within 1 kiB, the entire command is sent in the first packet, recapturing the single-roundtrip flow for a command and an ack. Multi-packet commands are downloaded by the commanded ship using the same congestion control as downloading any other potentially large datum – and, importantly, those incremental downloads do not necessarily trigger unnecessarily frequent disk writes.

3 Routing

3.1 Directed Routing

The Directed Messaging protocol gets its name from its routing scheme, which treats each bidirectional communication between two Urbit ships as directed, like a directed edge in a graph. For each request/response roundtrip, one ship is the requester, and the other is the responder, and that distinction is known at every layer of the system. Making this directionality known to routing enables a routing paradigm where a response packet traces the exact same relay path through the network as the request path, in the reverse order. Previous Urbit networking protocols used the opposite paradigm: “criss-cross routing”, so-called because both request and response could be routed through the destination ship’s “sponsor”, i. e. the supernode ship (“galaxy” root node or “star” infrastructure node) responsible for relaying packets to that ship. In contrast, in directed routing, the request and the response both use the same relay: the responder ship’s sponsor.

3.1.1 Request (Same for both Directed and Criss-Cross Routing)

```
requester's sponsor       -------- responder's sponsor
                         /             \
```

```
requester  ------------                 ---> responder
```

3.1.2 Response (Directed Routing)

```
requester's sponsor       -------- responder's sponsor
                         /             \
requester  <-----------                 ---- responder
```

3.1.3 Response (Criss-Cross Routing)

```
requester's sponsor -----          responder's sponsor
                /          \
requester <---              --------------   responder
```

3.2 Relaying

Making the directedness of communication legible to relays and Urbit runtimes allows the protocol to be a faithful, if Urbit-specific, Named Data Networking (NDN) protocol, which had been Urbit's stated goal since 2010. A Directed Messaging request packet acts as an NDN "interest" packet, and a response packet acts as an NDN "data" packet.

The NDN family of protocols, created by Van Jacobson et al. in the mid-2000s, differs from traditional networking protocols in that an interest packet has no sender address field – the identity of the sender of a request is unknown to the network. Instead, a receiver (relay or server) remembers "where" it heard the request from, and sends the response back to that. The notion of "where" varies depending on the layer of the stack the system is operating at: for an IP replacement, a relay would remember the physical interface (e. g. Ethernet port) on which it heard the incoming request. When building on top of IP and UDP, as Directed Messaging does, a receiver remembers the IP and port of the sender.

Previous Urbit network protocols still included the sender's Urbit identity in the request packets, failing to achieve the "source independence" property that defines NDN. Directed

Messaging is completely source-independent for reads, and for commands, it piggybacks a source-independent response onto a source-independent request in such a way that the receiver does know which Urbit address sent the request, but not in a way that requires packet routing to know or use the source Urbit address.

Source independence lets the routing operate fully locally, i. e. without reference to or knowledge of anything beyond a single hop away in either direction. This minimizes the data storage and synchronization requirements for nodes, which could become significant at scale.

Source independence entails stateful routing. Directed Messaging adopts NDN's Pending Interest Table, which stores a mapping from each outstanding request to the IP and port that sent that request. If the request times out (after roughly 30 seconds), or the request is satisfied by a response, the request is deleted from the table.

Since responses are authenticated down to the packet level, and immutable (meaning no cache invalidation is ever needed, only eviction), it should be straightforward for relays to cache responses, enabling efficient content distribution through the network. At present, each Urbit ship's runtime has a cache for its own responses, but supernodes (galaxies and stars) do not yet cache responses from their sponsored ships.

3.3 Peer Discovery

Urbit is a peer-to-peer network. Only root nodes (galaxies) list their own IP addresses publicly (on the Ethereum blockchain); all other ships must be discovered on demand. When one ship first sends a request to another ship, it generally doesn't know at what IP and port that ship could be reached, and it also doesn't know if the ship is reachable directly, or behind a firewall and only reachable through its sponsor.

The criss-cross routing used in previous Urbit protocols was hard to work with in practice and suffered from inefficiencies and bugs. Directed Messaging has a simpler approach to peer discovery that is easier to reason about.

The main difficulties with criss-cross routing stem from the structure of the internet. Most residential internet connections live behind a firewall that blocks all unsolicited incoming UDP packets. A laptop at home can send an outgoing packet to some destination IP and port, and the router will relay response packets back to the laptop for some time afterward, usually 30 seconds, as long as those response packets come from that same destination IP and port.

A UDP-based peer-to-peer network, then, needs to include not only residential nodes but also nodes on the public internet, not behind firewalls. These nodes must be discoverable so that residential nodes can ping them every 25 seconds, to ensure the residential nodes can receive messages. In Urbit, these public nodes are the galaxies (root nodes) and stars (infrastructure nodes). For now, only galaxies perform routing, due to edge cases with two levels of supernodes in peer discovery using criss-cross routing – we expect Directed Messaging will unblock stars from participating in routing.

When communicating with another ship for the first time, a ship first sends the packet to the other ship's sponsoring galaxy, which has an open connection to its sponsored ship if it's online. The galaxy receives a ping every 25 seconds from its sponsored ship. Whenever the source IP and port change, the galaxy's runtime injects an Arvo event and its Arvo OS saves the sponsored ship's new location to disk.

In Directed Messaging, when the galaxy receives a packet intended for one of its sponsored ships, it relays the packet. This uses the pending interest table described above to track the fact that there is an outstanding request that the galaxy is expecting to be honored by a response from the sponsored ship. The fundamental invariant of directed routing is that the response packet must trace the exact same path through the network as the request packet had, just reversed. Since this request had gone through this galaxy, the response must also route through the galaxy.

In Urbit's previous protocols, in contrast, the response would go directly from the sponsored ship back to the requesting ship, and also potentially through the requesting ship's sponsoring galaxy. This works in principle, but one drawback has to do with "route tightening" (described below): the response route only tightens to a direct route (without relays) if there are requests flowing in both directions; otherwise every response will flow through the requester's sponsor, even if the requester is not behind a firewall.

3.4 Route Tightening

It is better for performance, scaling, and individual sovereignty to obtain a direct route to another ship, rather than communicating through relays. In order to facilitate this, the system must automatically "tighten" a route over time to reduce the number of hops. Directed Messaging accomplishes this in the relay. When a relay receives a response packet (originally sent by a transivitely sponsored ship, but possibly through a relay once stars begin relaying packets), it appends the IP and port from which it heard that packet to the end of the packet before forwarding it to the requesting ship. Once the requesting ship receives this augmented packet, it knows the address appended to that packet is next hop in the relay chain. Once it knows that, when it sends packets to the responding ship, it can send them through the first hop in the relay chain (the receiving ship's galaxy), or to the next hop, which could be another relay or the ship itself.

In the current implementation, the requesting ship uses a simple procedure to decide which routes to send the packet on. It tracks the date of the last received packet from both the direct route and the route through the galaxy. A route is considered active if a packet has been received on it within the last five seconds. When the requesting ship goes to send a packet, if the direct route is active, it sends the packet on that route. Otherwise, it sends the packet through the galaxy and also sends a copy of the packet on the direct route as a probe.

This ensures continuity of communication when switching to or from a direct route, and it automatically tightens to the direct route if the direct route is responsive and loosens to the galaxy route if the direct route becomes unresponsive.

4 Authentication and Encryption

Directed Messaging is always authenticated and it supports encryption, in a number of different modes: - **Unencrypted reads:** A ship can publish data into its namespace without encryption. Response messages are signed using the ship's private key. The signature attests to the scry binding: the pair of the scry path and the datum at that path. - **One-to-One Encrypted Reads:** A ship can make data available to a single peer ship. Response messages are authenticated via external HMAC and internal signature. They are encrypted using a symmetric key derived from the Diffie-Hellman key exchange of the two ships' keys. - **Commands:** Commands and their acks are both handled as one-to-one encrypted reads. - **One-to-Many Encryption:** A ship can make data available to many ships by sharing an encryption key for that data to each ship using a one-to-one encrypted read. The requesting ship then uses that key to encrypt the request's scry path and decrypt the scry response.

All encryption is deterministic. This solves multiple problems: - It (along with other principles of Directed Messaging) prevents replay attacks by construciton. Every Directed Messaging packet is idempotent at the application level (a duplicate packet can trigger a duplicate ack and minor state changes in the runtime and Arvo kernel related to routing state, but it cannot modify anything visible to an application). - It allows encrypted values to be assigned paths in the scry namespace without maintaining nonce state for each encrypted value. Not tracking nonces reduces the system's security attack surface area considerably, since nonce state mismanagement is a common source of vulnerabilities. - Supporting encrypted values in the namespace allows the system to implement encryption using overlay namespaces (described in more detail below), which provide a clean layering that separates encryption from other concerns.

Directed Messaging's encryption scheme uses a variant of ChaCha20 with reduced rounds for performance. The process begins with XChaCha, an extended-nonce variant of ChaCha that accepts a 192-bit nonce instead of the standard 96-bit nonce. However, rather than using the caller-provided initialization vector directly as the nonce, Directed Messaging first applies a Blake3 key derivation function to derive a deterministic 24-byte nonce from the IV using the context string “mesa-crypt-iv”. This XChaCha operation with 8 rounds then produces a derived key and extended nonce, which are subsequently used for the actual ChaCha encryption (also with 8 rounds) of the message payload. The use of 8 rounds instead of the standard 20 is a performance optimization—ChaCha’s security margin allows for this reduction in cryptographic applications where the extreme paranoia of 20 rounds may be unnecessary, and the deterministic nonce derivation via Blake3 adds an additional layer of domain separation. The encrypted output is then appended with a single 0x1 trailer byte that serves as a format marker, which the decryption process verifies before removing to recover the plaintext. This construction provides authenticated encryption properties through the deterministic relationship between the IV and nonce, ensuring that any tampering with the ciphertext or IV will result in decryption failure.

The trailer byte solves a representation problem specific to Urbit’s atom system. In Urbit, data is ultimately stored as atoms (arbitrary-precision integers), and atoms don’t distinguish between 0x1000 and 0x1000000000—trailing zero bytes are simply not part of the numeric value. If ciphertext happens to end with one or more zero bytes, those would be stripped when the ciphertext is represented as an atom, corrupting the data. By appending a single 0x1 byte to the end, Directed Messaging ensures that the atom representation always preserves the full length of the ciphertext including any trailing zeros. During decryption, the code verifies that the final byte is indeed 0x1 (which catches truncation or corruption) and then strips it before decrypting. It’s purely a data representation concern, not a cryptographic feature—without it, you’d silently lose any ciphertext bytes that happen to be zero at the end.

4.1 Message Authentication and Encryption Details

4.1.1 Overlay Namespaces

Each of the three privacy modes has its own "overlay namespace", i. e. a part of the scry namespace that somehow transforms values bound to a different part of the namespace. A path in an overlay namespace often consists of a path prefix containing the name of the overlay and any parameters needed for the transformation it performs on the datum at the overlaid path, followed by the overlaid path.

A handler function that deals with scry requests to the overlay namespace is free to parse transformation parameters out of the overlay prefix, inspect the overlaid path, make a scry request for the data at that path, make scry requests related to that path (such as existence checks for file paths), and run arbitrary code to transform the results. This is all possible because scry requests are purely functional by construction – they are deterministic functions of their inputs, with no hidden inputs, and there is no mechanism by which they could change Arvo state.

Directed Messaging uses overlay namespaces not only for privacy modes, but also to publish individual packets within a message. The packet's size (expressed as the log base 2 kiB, e. g. 3 for 8kiB or 4 for 15kiB) and fragment number (index within the message) are parameters to this overlay, which overlays the message's scry path.

Directed Messaging is the first Urbit kernel module to make heavy use of overlay namespaces in its design. They play an important role in maintaining boundaries between layers within the system: privacy is separated, by construction, from other concerns due to the isolation imposed by overlay namespaces. For example, an application can declare what privacy mode it wants to use for a piece of data it publishes, and the kernel enforces that by exposing it over the network using the appropriate overlay namespace.

4.1.2 %publ namespace (unencrypted public)

Authentication: Ed25519 signature only (no encryption)

1. No encryption: The message data is jammed but not encrypted. The serialized response is sent in plaintext.
2. Signature authentication: A 64-byte Ed25519 signature is computed over:

- The encoded beam path
- The LSS root of the unencrypted jammed data

3. Publisher's signing key: The signature uses the publisher's Ed25519 private key (extracted from their networking key).
4. Verification: The receiver verifies the signature using the publisher's Ed25519 public key retrieved from Azimuth.

The `%publ` namespace uses unencrypted paths with the structure:

`/publ/[life]/[path]`

Components:

1. life: The life (key revision number) of the publisher's networking keys, used to identify which public key to use for signature verification.
2. path: The actual user path in plaintext. This is not encrypted and is visible to all network observers.

Privacy properties:

- Everything is public: both the publisher's identity/key version and the content path are visible to anyone
- No encryption is applied to either the path or the message payload
- Authentication comes solely from the Ed25519 signature, which proves the publisher created this content
- Key rotation is supported through the life counter

Use cases:

- Public data that should be readable by anyone
- Content where authenticity matters but confidentiality doesn't
- Simpler than encrypted namespaces since no key exchange or group key distribution is required

4.1.3 %shut namespace (group encrypted)

Authentication: Ed25519 signature over encrypted data

1. Message encryption: The message is encrypted with XChaCha20-8 using a group symmetric key. The key is provided via the %keen task (not derived from ECDH). The encrypted path indicates which group key to use.
2. Signature authentication: A 64-byte Ed25519 signature is computed over:

- The encoded beam path
- The LSS root of the encrypted data

3. Publisher's signing key: The signature uses the publisher's Ed25519 private key, proving the publisher created this encrypted payload.
4. Verification: The receiver verifies the signature using the publisher's Ed25519 public key from Azimuth, then decrypts with the group key.
5. Security model: Signature proves authenticity from the publisher; encryption provides confidentiality to group members. Anyone with the group key can decrypt, but only the publisher can create valid signatures.

The %shut namespace uses encrypted paths with the structure:

`/shut/[key-id]/[encrypted-path]`

Components:

1. key-id: A numeric identifier indicating which group symmetric key to use for decryption. This allows multiple groups to be supported, each with their own key.
2. encrypted-path: The actual user path, encrypted using the group key. This is sealed with the same symmetric key used to encrypt the message payload, making the entire scry path opaque to anyone without the group key.

Privacy properties:

- The actual content path is hidden from network observers and unauthorized parties

- Only the key ID is visible in plaintext
- The group key must be obtained separately (typically via a %keen task) to decrypt both the path and the message payload
- Different groups using different key IDs can coexist without revealing which content is being accessed

4.1.4 %chum namespace (1-to-1 encrypted)

Authentication: HMAC only (no signatures)

1. Message encryption: The message is encrypted with XChaCha20-8 using a symmetric key derived from Curve25519 ECDH key exchange between the two ships' networking keys.
2. HMAC authentication: A 16-byte HMAC (Blake3 keyed hash) is computed over:

- The encoded beam path
- The LSS root of the encrypted data

3. Shared symmetric key: Both HMAC computation and encryption use the same ECDH-derived symmetric key. Both parties can independently derive this key from their own private key and the other party's public key.
4. Verification: The receiver verifies the HMAC using the shared symmetric key, then decrypts with the same key.
5. Security model: The HMAC proves the sender possesses the shared symmetric key (implicitly authenticating them as the expected peer). No signatures are needed since only two parties share this key. This applies to both pokes and acks.

The %chum namespace uses encrypted paths with the structure:

```
/chum/[server-life]/[client-ship]/[client-life]/[encrypted-path]
```

Components:

1. server-life: The life (key revision number) of the server ship's networking keys, used to identify which version of their keys to use for ECDH key derivation.

2. client-ship: The @p address of the client ship in the communication pair.
3. client-life: The life of the client ship's networking keys, used to identify which version of their keys to use for ECDH key derivation.
4. encrypted-path: The actual user path, encrypted using the symmetric ECDH key derived from both ships' networking keys. This makes the scry path opaque to network observers.

Privacy properties:

- The actual content path is hidden from network observers
- The identities of both parties and their key versions are visible in plaintext
- Only the two ships involved can derive the symmetric key to decrypt the path and payload
- Key rotation is supported through the life counters

4.1.5 Other Cryptographic Properties

Directed Messaging relies on the ability to rotate keys on chain for its forward secrecy. Future versions of the protocol might add a ratchet to minimize the damage if a private key is compromised.

4.2 Packet Authentication

TODO: more closely match the tone of the rest of this document TODO: more diagrams, clearer explanation of Lackman traversal

One of the goals of Directed Messaging was to improve upon the safe-but-dumb "sign every 1 KiB packet" design of old `%ames`. The standard approach is to use asymmetric crypto to establish a shared AEAD key, and use it to authenticate each packet. This is just as safe as signing every packet, and orders of magnitude faster. However, we still can't verify that a peer is sending us *correct* data until we've received the entire message. They could send us 999 good packets and 1 bad one, and we'd have no way of knowing which was which. This is especially annoying if we want to download in parallel from multiple peers: if the final result is invalid, which peer is to blame? If we want to solve this problem, we need to get a little more bespoke.

Verifying that a packet belongs to a particular message is a job for a Merkle tree. So our protocol needs to split message data into the leaves of a tree, and send both leaf data and tree hashes. Early on, we debated whether to make these distinct request types; we settled on interleaving them. Now the question becomes: which tree hashes do you need to send, and when do you send them?

Our relevant design constraints were as follows:

- Sufficiently-small messages should not require more than one packet.
- It should be possible to download in parallel.
- The protocol should be flexible with respect to the size of a leaf.

An obvious first place to look for inspiration was Bao. However, Bao is not very amenable to being split into fixed-size packets: it intermingles leaf data and tree hashes into one stream, and the number of consecutive hashes varies based on the offset. You could modify it such that each packet consists of a leaf followed by at most one hash; indeed, this was the initial plan. Visually:

```
                       ┌──────────────┴─────────────a
           ┌───────────┴───────────b              ┌─┴─q
     ┌─────┴─────c           ┌─────┴─────j        │   │
  ┌──┴──d     ┌──┴──g     ┌──┴──k     ┌──┴──n     │   │
┌─┴─e ┌─┴─f ┌─┴─h ┌─┴─i ┌─┴─l ┌─┴─m ┌─┴─o ┌─┴─p ┌─┴─r │
0   1 2   3 4   5 6   7 8   9 10 11 12 13 14 15 16 17 18
```

This is a binary numeral tree: a structure composed of perfect binary trees, imposed upon a flat sequence of bytes. The numbers 0-18 represent leaf data (typically 1 KiB per leaf), while letters *a*-*r* represents tree hashes that are used to verify the leaves. So packet 3 would contain bytes 3072–4096 and leaf hash *d*.

The main problem with this approach is that it requires buffering. In order to verify leaf 0, we need hashes *a* through *e* – five packets! Worse, once we've received five packets, we have the whole `[0,4)` subtree, making hashes *d* and *e* redundant. (In BNTS, we use the notation `[n,m)` to refer to the perfect subtree containing leaves `n` through `m-1`.)

Buffering a few packets is not the end of the world, but the whole thing had kind of a bad smell to it. We asked: what would happen if we added another constraint?

- It should be possible to validate each packet as soon as it arrives, with no buffering.

For starters, an inescapable consequence of this constraint is that we must send the *full* Merkle proof for the first leaf before we can send the leaf data itself. Also, we can no longer send hashes that can't be immediately verified. For example, to verify *g*, we first need to have verified *c*; we can then combine *g* with its sibling hash and confirm that the result matches *c*.

While adding another constraint seems like it would make our life harder, in reality the opposite happened: the protocol was greatly simplified. It turns out that by front-loading the initial proof hashes, we ensure that the received leaf data and hashes will never "run ahead" of what can be immediately verified. Here's what it looks like in practice:

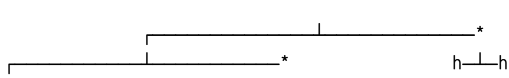
```
                       ┌──────────────┴─────────────*
           ┌───────────┴───────────*              l─┴─l
     ┌─────┴─────*           e─────┴─────e        │   │
  ┌──┴──*     b──┴──b     f──┴──f     i──┴──i     │   │
*─┴─* a─┴─a c─┴─c d─┴─d g─┴─g h─┴─h j─┴─j k─┴─k m─┴─m │
0   1 2   3 4   5 6   7 8   9 10 11 12 13 14 15 16 17 18
```

In the initial packet, we send the Merkle proof for leaf 0, i. e. all of the hashes marked with `*`. Each subsequent packet contains leaf data (leaf 0, 1, 2, etc.), and possibly a "pair" (*a*, *b*, *c*, etc.), which comprises *both* child hashes under a particular node. Packet-by-packet, the verifier sees:

```
Packet 0: Signed Merkle root + Merkle proof for leaf 0.
          Verify proof against signed root.
          We now have [0,1), [1,2), [2,4), [4,8), [8,16),
     and [16,19).
Packet 1: Leaf 0 + Pair a.
          Verify leaf 0 against [0,1).
          Verify pair a against [2,4).
          We now have [1,2), [2,3), [3,4), [4,8), [8,16),
     and [16,19).
Packet 2: Leaf 1 + Pair b.
          Verify leaf 1 against [1,2).
          Verify pair b against [4,8).
          We now have [2,3), [3,4), [4,6), [6,8), [8,16),
     and [16,19).
Packet 3: Leaf 2 + Pair c.
          Verify leaf 2 against [2,3).
          Verify pair c against [4,6).
          We now have [3,4), [4,5), [5,6), [6,8), [8,16),
     and [16,19).
... and so on.
```

At each step, we "consume" two hashes (one to verify a leaf, one to verify a pair), and add the pair to our verified set; thus, the number of verified-but-unused hashes stays constant until we get to packet 13. At this point, packets still contain leaf data (so we'll consume one hash), but there are no more pairs left to send; thus, our stockpile of hashes is steadily exhausted, until we consume the final hash to verify the final packet.

This is a solid improvement! We dubbed it "Lockstep Streaming," after the fact that verification proceeds in lockstep with packet receipt. But when we sat down to write the code for matching verified-but-unused hashes to incoming leaves and pairs, things got hairy. It was clearly *possible*, but the ugliness of the logic suggested that there was a better way. And after filling plenty of notebook pages with hand-drawn Merkle trees, ~rovnys-ricfer found it: a mapping of packet numbers to hash pairs that was not only much cleaner, but also *scale invariant*. It's called Blackman ordering, and it looks like this:



```
     ┌─────┴─────*           d─────┴─────d        │   │
  ┌──┴──*     b──┴──b     f──┴──f     j──┴──j     │   │
*─┴─* a─┴─a c─┴─c e─┴─e g─┴─g i─┴─i k─┴─k m─┴─m o─┴─o │
0   1 2   3 4   5 6   7 8   9 10 11 12 13 14 15 16 17 18
```

See the difference? Instead of ordering the pairs on a first-needed basis, we jump around a bit. Specifically, we send a pair whose "height" corresponds to the number of trailing zeroes in the binary representation of the packet number. For example, packet 4 is 100 in binary, with two trailing zeroes, so we send pair d, which sits two levels above the leaves. (As you might expect, the logic gets slightly less clean when the number of leaves is not a power of two, but it's not the end of the world.) an Packet-by-packet verification for Blackman ordering:

The `[x,y)` notation indicates a half-open set, i. e. it includes `x, x+1, x+2, ..., y-1`. `[2, 4)` contains elements `2` and `3`. `[0,1)` contains the single element `0`.

```
Packet 0: Signed Merkle root + Merkle proof for leaf 0.
          Verify proof against signed root.
          We now have [0,1), [1,2), [2,4), [4,8), [8,16),
     and [16,19).
Packet 1: Leaf 0 + Pair a.
          Verify leaf 0 against [0,1).
          Verify pair a against [2,4).
          We now have [1,2), [2,3), [3,4), [4,8), [8,16),
     and [16,19).
Packet 2: Leaf 1 + Pair b.
          Verify leaf 1 against [1,2).
          Verify pair b against [4,8).
          We now have [2,3), [3,4), [4,6), [6,8), [8,16),
     and [16,19).
Packet 3: Leaf 2 + Pair c.
          Verify leaf 2 against [2,3).
          Verify pair c against [4,6).
          We now have [3,4), [4,5), [5,6), [6,8), [8,16),
     and [16,19).
... and so on.
```

Shortly after, ~watter-parter tweaked the ordering slightly, offsetting it by one; this further simplified the low-level bithacking. We called this variant "Lackman ordering," and it's what we used in the final version of Lockstep Streaming.

Packet-by-packet verification for Lackman ordering:

```
Packet 0: Signed Merkle root + Merkle proof for leaf 0.
          Verify proof against signed root.
          We now have [0,1), [1,2), [2,4), [4,8), [8,16),
     and [16,19).
Packet 1: Leaf 0 (no pair).
          Verify leaf 0 against [0,1).
          We now have [1,2), [2,4), [4,8), [8,16), and
     [16,19).
Packet 2: Leaf 1 + Pair a.
          Verify leaf 1 against [1,2).
          Verify pair a against [2,4).
          We now have [2,3), [3,4), [4,8), [8,16), and
     [16,19).
Packet 3: Leaf 2 + Pair b.
          Verify leaf 2 against [2,3).
          Verify pair b against [4,8).
          We now have [3,4), [4,6), [6,8), [8,16), and
     [16,19).
... and so on.
```

Compared to Blackman ordering, the pairs appear offset by one position, which simplifies the bit-manipulation logic for computing which pair to include in each packet.

There's one more optimization worth mentioning: If the message is small enough, we can skip the initial step of sending a packet containing only a Merkle proof (with no leaf data). Obviously, for a one-leaf message, we can simply send that leaf; the hash of that leaf is the Merkle root. For a two-leaf message, we can send the leaf plus its sibling hash (`[1,2)`); the verifier can hash the first leaf and combine it with the sibling hash to recover the root. And for a three- or four-leaf message, we can send `[1,2)` and `[2,3)` (or `[2,4)`, respectively). That's the limit, though; if a message has five leaves, we would need to send at least three sibling hashes for the verifier to recompute the root, but our packet framing only allows up to two hashes.

4.2.1 Arena Allocator

Directed Messaging uses a simple bump allocator arena for memory management. Each arena is a contiguous block of memory with three pointers: the start of the allocation (`dat`), the current allocation position (`beg`), and the end of the block (`end`). The `new()` macro allocates objects by advancing the `beg` pointer with proper alignment.

The arena allocator provides **no individual deallocation**—once memory is allocated from an arena, it can't be freed separately. Instead, the entire arena is freed at once when the data structure that owns it is destroyed.

Allocation Patterns Arenas are created with sizes tailored to their use case:

Pending Interest Table entries use **16 KiB arenas**. These store lane addresses for pending requests, with the arena holding the entry itself plus a linked list of address records.

Pending requests allocate arenas at **5× the expected message size**. A request receiving a 1 MiB message gets a 5 MiB arena. This single arena holds the request state, fragment data buffer, LSS authentication pairs, packet statistics, bitset tracking received fragments, and pre-serialized request packets.

Jumbo frame cache entries allocate based on proof size, data size, hash pairs, plus a 2KB buffer. For a 1 MiB message, this might be around 1-2 MiB. The arena stores the cached response data, Merkle proof spine, and authentication hashes.

Temporary arenas for packet sending use message size plus 16KB to hold serialized packets plus overhead.

Scry callbacks get small arenas for asynchronous Arvo interactions.

Deallocation Triggers Arenas are freed only when their parent data structure is destroyed:

Request completion: When all fragments arrive, the request is deleted and its arena freed. This happens asynchronously through libuv's handle cleanup to ensure proper timer shutdown before freeing memory.

Authentication failure: If LSS verification fails while processing fragments, the entire request is immediately deleted.

Timeout expiration: When retry timers exhaust their attempts, the request is deleted.

PIT expiration: After **20 seconds**, entries are cleaned from the Pending Interest Table.

Cache eviction: When the jumbo cache exceeds **200 MB**, it's entirely cleared and all cached arenas are freed.

Lifecycle A typical request lifecycle:

1. **Allocation**: Receive initial packet, create arena with 5× data size, allocate all request state from arena
2. **Growth**: As fragments arrive, write into pre-allocated buffers within the arena
3. **Completion**: All fragments received, construct final message, send to Arvo
4. **Cleanup**: Delete request from map, stop timer, close handle asynchronously
5. **Deallocation**: In UV callback, free entire arena with single call

This design trades memory efficiency for speed. Arenas may hold unused space, but allocation is extremely fast (just pointer arithmetic), and the single-free design eliminates per-object deallocation overhead and fragmentation issues.

4.2.2 Download Checkpointing

This has not been deployed to the network, but this design allows the requesting ship's runtime to inject jumbo frames of arbitrary size into its Arvo as each one finishes downloading, with real authentication by using Lockstep. Arvo will seamlessly store those jumbo frames and accumulate them until it has the whole message, at which time it will deserialize the message into an Urbit 'noun' data structure and deliver it to the application or kernel module that had triggered the request.

This allows the system to make effective use of Arvo as a download checkpointing system. After a process crash, machine restart, or any other transient failure, the download can be resumed with minimal loss of information.

Injecting an authenticated jumbo frame into Arvo maintains a security boundary. Urbit's main runtime, Vere, has two Unix processes: one runs the Arvo kernel, and the other handles input and output. Arvo maintains ultimate responsibility for cryptographic operations. This lets the private key remain solely in the Arvo process, leaving the I/O process without the ability to encrypt, decrypt, authenticate, or verify authentication.

Instead, the runtime delegates any operation requiring a private key to Arvo, including validating the message-level authentication in the first packet of a scry response. To add a layer of defense in depth in case the I/O process is compromised, Arvo performs its own packet validation, including the Lockstep packet authentication. This remains efficient because each packet can be a large jumbo frame.

Checkpointing has another benefit. No matter how large a message is, the downloader can keep a fixed upper bound on the memory footprint while download that message, proportional to one jumbo frame.

4.2.3 Download Resumption

After a transient failure – most commonly a process crash or machine restart – a requesting ship can resume a download. In order to pick up from where it left off, the runtime first asks its local Arvo for the leaf-packet hashes it needs, which Arvo generates on demand from the jumbo frames that have already been downloaded and stored in Arvo. This is $O(\log(n))$ hashes, where n is the message length, and no message data needs to be sent over inter-process communication in order to resume a download, preventing restarts from becoming slow and memory-intensive.

Once the runtime has the hashes it needs, it resumes the Lockstep streaming verification that it had been doing, beginning with the next jumbo frame after the last one that had been downloaded and saved in Arvo.

Download checkpointing and resumption together provide a good set of tools for download management. This is beyond what TCP provides, or even HTTP. HTTP has resumption headers, but both client and server have to opt into using them, so in practice many HTTP-based downloads cannot be resumed.

5 Congestion Control

In Directed Messaging, congestion control is pluggable. The requesting ship decides how many request packets to send and at what time. The publisher ship is only responsible for responding to requests and does not participate in congestion control.

It is possible for an Urbit implementation to have functioning, if not performant, networking, without any runtime implementation of congestion control. The formal specification in the networking module of the Arvo kernel for how a ship sends request packets is a simple one-at-a-time indefinite repeating timer. The ship sends the first request packet, repeating it every thirty seconds until a response packet is heard, at which point it begins requesting the next packet.

Performant congestion control, then, is an extension of Urbit's idea of a "jet", i. e. a peephole optimization that replaces a built-in function, defined formally but is likely slow, with an optimized low-level implementation.

In practice, this slow packet re-send is used for retrying dead connections, where the publishing ship has been unresponsive for a long time. This is important because Urbit network requests generally do not time out at the application level; they are considered persistent, and they must be retried indefinitely. Fast, runtime-based congestion control only kicks in when the runtime receives a response packet, indicating the publishing ship has become responsive.

The current implementation of Directed Messaging employs a modified TCP Tahoe-style congestion control algorithm adapted to its request/response architecture and packet-oriented nature. The protocol's congestion control differs from traditional TCP in several fundamental ways due to its pull-based communication model and implicit acknowledgment scheme.

5.1 Architectural Foundation

Unlike TCP's push-based model where senders transmit data and await separate acknowledgment packets, Directed Messaging operates on a request/response paradigm. The requesting ship sends PEEK packets to solicit specific fragments, and the responding ship sends PAGE packets containing the requested data. The arrival of each PAGE packet serves as an implicit acknowledgment—no separate ACK packets exist in the protocol. This inversion places congestion control responsibility on the requester rather than the sender, allowing the party pulling data to directly regulate network load.

The protocol operates on fixed-size fragments rather than byte streams. Each fragment contains up to 1024 bytes of payload data (at the default `$bloq` parameter of 13). The congestion window (`cwnd`) measures capacity in fragment count rather than bytes, providing coarser but simpler granularity than TCP's byte-oriented approach.

5.2 State Variables

Congestion control state is maintained per peer and includes:

- `cwnd` (congestion window): Number of fragments allowed in flight simultaneously
- `ssthresh` (slow start threshold): Boundary between exponential and linear growth phases
- `rtt` (round-trip time): Smoothed estimate of network latency
- `rttvar` (RTT variance): Smoothed variance in round-trip measurements
- `rto` (retransmission timeout): Calculated timeout for loss detection

Per-request state tracks which fragments have been sent, when they were sent, how many retransmission attempts have occurred, and which fragments have been received using an efficient bitset representation.

5.3 Slow Start and Congestion Avoidance

The protocol implements two growth phases analogous to TCP:

- **Slow Start Phase (`cwnd < ssthresh`)**: Upon initiating a request or recovering from congestion, `cwnd` begins at 1 fragment. For each fragment acknowledgment received (implicitly, by receiving the corresponding PAGE packet), `cwnd` increments by 1. This produces exponential growth: 1→2→4→8→16, allowing rapid probing of available bandwidth.

- **Congestion Avoidance Phase (`cwnd >= ssthresh`)**: Once `cwnd` reaches `ssthresh`, growth becomes linear. The implementation uses a fractional accumulation strategy: for each acknowledgment, a fractional counter accumulates 1/`cwnd` of a window increment. When the accumulated value reaches `cwnd`, the actual `cwnd` increments by 1, yielding approximately one window size increase per round-trip time.

The default `ssthresh` is initialized to 10,000 fragments (approximately 10 MB), effectively allowing slow start to dominate for typical transfer sizes.

5.4 Loss Detection and Recovery

Directed Messaging currently implements timeout-based loss detection only, without fast retransmit or fast recovery mechanisms. This places it closest to TCP Tahoe's behavior, though with an important modification to timeout handling.

- **Timeout Detection**: Each in-flight fragment's transmission time is recorded. A retransmission timer fires when the oldest unacknowledged fragment exceeds the calculated RTO. The protocol scans all in-flight fragments upon timeout and retransmits any that have been outstanding beyond the RTO interval.

- **Timeout Response**: Upon detecting packet loss via timeout, the protocol reduces network load by:

1. Setting `ssthresh = max(1, cwnd / 2)`.
2. Setting `cwnd = ssthresh`.
3. Doubling `rto` (up to a maximum bound).

This differs from TCP Tahoe, which sets `cwnd = 1` and restarts slow start from the beginning. Directed Messaging's approach is less conservative, immediately resuming transmission at the reduced threshold rather than slowly ramping up from a single packet. This assumes that while congestion occurred, the network can still sustain traffic at half the previous rate without requiring a full slow start restart.

The lack of fast retransmit (triggering on three duplicate acknowledgments) represents a significant difference from modern TCP variants. Fast retransmit requires detecting duplicate acks, which in Directed Messaging would mean detecting requests for the same fragment. However, the current implementation treats each arriving PAGE packet independently without tracking the ordering implications that would enable fast retransmit. This is a known simplification intended for future enhancement.

5.5 Round-Trip Time Estimation

The protocol employs Jacobson/Karels RTT estimation, the same algorithm used in TCP. When a fragment acknowledgment arrives (excluding retransmissions), the round-trip time measurement (`rtt_datum`) is calculated as the difference between current time and transmission time.

RTT smoothing uses exponential weighted moving averages with traditional TCP parameters:

- `rtt = (rtt_datum + 7×rtt) / 8  ( α = 1/8)`
- `rttvar = (|rtt_datum - rtt| + 7×rttvar) / 8  ( β = 1/4)`

The retransmission timeout is calculated as `rto = rtt + 4×rttvar`, clamped to a minimum of 200 milliseconds and a maximum that varies by context (typically 2 minutes for most traffic, 25 seconds for keepalive probes to sponsors).

Retransmitted fragments do not contribute to RTT estimation, following Karn's algorithm to avoid ambiguity about which transmission is being acknowledged.

5.6 Selective Request Architecture

The implicit acknowledgment scheme combines naturally with selective fragment requesting. The protocol maintains a bitset tracking which fragments have been received. When requesting additional fragments during congestion-controlled transmission, the requester consults both the congestion window (how many new requests can be sent) and the bitset (which fragments are needed). This provides the benefits of TCP SACK without requiring additional protocol machinery – selectivity is inherent to the request/response model.

When fragments arrive out of order, the LSS (Lockstep Signature Scheme) authentication requires buffering misordered packets until their Merkle proof predecessors arrive. Once authenticated, these fragments are marked received in the bitset, and the congestion control state updates accordingly.

5.7 Request Rate Limiting

The congestion window limits the number of PEEK requests in flight. Before sending additional requests, the protocol calculates

```
available_window = cwnd - outstanding_requests
```

where `outstanding_requests` counts fragments that have been requested but not yet received. This naturally throttles the request rate according to observed network capacity. As PAGE packets arrive (serving as acknowledgments), `outstanding_requests` decreases, allowing new PEEK packets to be sent.

This pull-based flow control provides inherent advantages: the requester cannot be overwhelmed by data it didn't request, and the congestion control directly limits the rate at which the requester pulls data from the network.

5.8 Per-Peer State Management

Congestion control state is maintained per peer rather than per connection or per flow. All concurrent requests to the same peer share a single congestion window and RTT estimate. This design choice reflects the architectural principle that network capacity constraints exist between pairs of ships rather than between individual conversations.

Sharing state across requests to the same peer provides several benefits:

1. RTT measurements from any request improve estimates for all requests.
2. Congestion signals from one request protect other concurrent requests.
3. State initialization costs are amortized across multiple requests.
4. The aggregate transmission rate to each peer is controlled.

However, this also means that multiple concurrent large transfers to the same peer must share available bandwidth, which could reduce throughput compared to per-flow windows in some scenarios.

5.9 Initial Window and Probing

New peer connections begin with conservative initial values: `cwnd = 1`, `rtt = 1000 ms`, `rttvar = 1000 ms`, `rto = 200 ms`. The first fragment request initiates RTT measurement and slow start growth. This cautious initialization ensures the protocol probes network capacity gradually rather than assuming high bandwidth is available.

For peers with no recent traffic, the congestion state persists but becomes stale. Future enhancements may include state expiration and re-initialization after prolonged idle periods, though the current implementation maintains state indefinitely once a peer is known.

5.10 Comparison with TCP Variants

The congestion control algorithm most closely resembles TCP Tahoe but with notable differences:

Similarities to Tahoe:

- Slow start with exponential growth
- Congestion avoidance with linear growth
- Loss detection via timeout only
- Conservative initial probing

Differences from Tahoe:

- Modified timeout recovery (`cwnd = ssthresh` rather than `cwnd = 1`)
- Packet-oriented rather than byte-oriented windows
- Implicit acknowledgment via data receipt
- Pull-based rather than push-based architecture
- Per-peer rather than per-connection state

Compared to NewReno/SACK, the protocol lacks fast retransmit and fast recovery, making it less responsive to isolated packet loss. However, the selective request architecture provides the functional benefits of SACK naturally. The implicit acknowledgment scheme eliminates issues with ACK loss and compression that affect TCP.

5.11 Design Trade-offs

The congestion control design reflects several architectural trade-offs:

Advantages:

- Simpler than modern TCP variants (no fast recovery complexity).

- Natural selective acknowledgment through request/response model.
- Requester controls rate, preventing receiver overwhelm.
- No separate ACK channel to fail.
- Precise retransmission control with bitset tracking.

Limitations:

- Lack of fast retransmit increases latency for isolated losses.
- Packet-oriented windows provide coarser bandwidth control.
- Per-peer state sharing may reduce throughput for concurrent flows.
- Modified timeout behavior is less studied than standard algorithms.

5.12 Future Enhancements

The protocol architecture supports several potential improvements without fundamental redesign. Fast retransmit could be implemented by tracking fragment request patterns and detecting when requests skip over missing fragments. Fast recovery could leverage the existing `ssthresh` calculation while avoiding the full slow start restart. Additional sophistication in RTT measurement could distinguish network delay from application processing time.

The current implementation represents a pragmatic balance between simplicity and effectiveness, providing reasonable congestion control while keeping the protocol accessible to implementation and formal verification.

6 Integration

In order to deploy Directed Messaging to Urbit's live network, the previous version of the protocol needed to remain operational, since there is no central authority that can force Urbit ships to update to a particular version. The authors decided further that each ship should be able to upgrade connections to peer ships one by one and be able to downgrade it without data loss.

This was possible due to the persistent, transactional nature of both the previous and new versions of the protocol.

6.1 Ames Flows

The Arvo kernel has a concept of an Ames "flow", a directed connection between ships where the subscriber ship can send commands and the publisher ship can send responses, both as "commands" at the level of Directed Messaging.

The implementation of Directed Messaging maintained the interface to the rest of the system, without modification. Applications do not need to modify their code at all to make use of Directed Messaging.

7 Future Work

TODO not sure what to include here

- star relaying
- star scry caching
- download checkpointing and resumption
- add fast retransmit to congestion control

speculative: - add %pine - add sticky scry Those together would flesh out a full pub-sub system with stateless publishers

References

Cheshire, Stuart (1996) “It’s the Latency, Stupid”. URL: `https://www.stuartcheshire.org/rants/latency.html` (visited on ~2024.6.5).