Urbit Networking: Ad Fontes

Edward Amsden ~ritpub-sipsyl,
Ted Blackman ~rovyns-ricfer
Zorp Corp, Urbit Foundation

Abstract

The Ad Fontes proposal was an influential document outlining a vision for how Urbit networking could be made more reliable and solid-state. Its ideas have remained relevant as remote scry, directed messaging, and other developments have advanced the state of the Ames protocol. This presentation of ~ritpub-sipsyl's original document and ~rovnys-ricfer's response provides a historical perspective on the development of the Urbit networking stack, and is lightly annotated to connect its ideas to current developments. Some technical details as implemented deviate from the proposal and response.

Contents

Urbit Systems Technical Journal I:2 (2024): 67–87.

Address author correspondence to ~ritpub-sipsyl.

1 Ad Fontes

The original Ad Fontes proposal was produced by ~ritpub-sipsyl *on* ~2022.12.2 *and revised through* ~2022.12.10.

1.1 In Principio

> On Mars, SIN is taken to an extreme. Logically, Urbit is a single broadcast network – a single big Ethernet wire. Everyone sees everyone else's packets. You don't send a packet. You "release" it. As a matter of practical optimization, of course, routing is necessary, but it is opaque to the receiver. If there is a routing envelope, it is stripped before processing.

> An easy way to see this is to compare an Urbit packet, or card, to an IP packet. An IP packet has a header, which is visible both to the routing infrastructure and the application stack. The header, of course, contains the infamous source address. Urbit removes not just the source address, but also the rest of the header. An Urbit card is all payload – it has no region seen both by receiver and router. (~sorreg-namtyv, 2010)

> In the classical stack, a basic result of protocol design is that you can't have exactly-once message delivery . To put it in the terms ... [of] subsection 3.2: you can't build a bus on a network. With permanent networking between solid-state interpreters, this feature is straightforward. Why? Because two uniformly persistent nodes can maintain a permanent session
>
> Arvo defines a typed, global, referentially transparent namespace with the Ames network identity at the root of the path. User-level code has an extended Nock operator that dereferences this namespace and blocks until results are available. So the Hoon programmer can use any data in the Urbit universe as a typed constant. (~sorreg-namtyv et al., 2016)

1.2 Nunc

Today there is Ames. Ames provides a command protocol with exactly-once semantics over the wire. Urbit ships can poke other ships and expect eventually once, and only once, action on the poke provided that at some point in the future both hosts are able to communicate. Urbit ships can also watch a `path` (`wire`) on another ship and receive updates, provided the subscription is not kicked. Both commands and updates must be acknowledged as a core requirement of the protocol.

What does not exist is a viable implementation of the typed, global, referentially transparent namespace. Neither the in-

terface for application writing, nor the implementation of the stack beneath it, support programming by binding paths in a namespace and apprising other ships of such bindings or permitting them to request such bindings.

This document is in aid of the effort to implement such an interface and networking stack.

1.3 Observations

> As a matter of practical optimization, of course, routing is necessary, but it is opaque to the receiver.

This is a massively important observation. Arvo should not "do routing." It should not be aware of sources or destinations. The runtime may route. Hoon code kept up to date by Arvo for the runtime's use may route. Agents may route. Non-Urbit or quasi-Urbit entities on the network may route. Arvo, a kelvin-versioned artifact which must not be subject to the uncertainty of external technical development, must not route.

1.4 The Stack

From `~rovnys-ricfer`:

- PKI
- routing and peer discovery
- transport layer (including packetization)
- scry resolution layer
- frontier discovery layer
- command layer

The PKI can be treated for now as a solved problem. Routing and peer discovery must have presently viable implementations, but nothing which is being kelvined should constrain future, more Urbit-maximalist implementations from creation.

Likewise the transport layer. Urbit needs to function both now, over various Ethernets, and millenia from now, over the Third and Most Glorious Quasiluminal Marterran Subspace Transponsive. Packetization is not something that should be kelvin versioned. None of these problems cry out for an immediate solution, though packetization ought not to churn Arvo's event loop.

1.4.1 Scry resolution

The scry resolution layer is where the rubber meets the road. How are paths bound by remote ships resolved to data? The only possible way is for some ship (initially the binding ship, and later possibly other ships and even non-urbit transponders) to communicate the binding. Since bindings are irrevocable, we call a particular binding an "oath" and we say that a ship will "avow" the oath by taking action to inform others of it.

A ship has no possible way to force its runtime to do anything. Everything depends on the runtime's operations on a given and current state of Arvo. (More of the runtime should be written in Hoon, it should be said.) But a ship can advise or insist that the runtime take some action. Oaths are immutable but may be avowed many times.

1.4.2 Frontier discovery

The frontier is, at this point in time, the limit of what is knowable. In general we do not know all that is knowable. Many ships have bound and even avowed oaths which we have not yet heard of, and may never hear of. But we would like to be able to reliably discover what oaths our peers will avow to us.

Unfortunately, there is no way to do this which is optimal for all applications. The search for such a mechanism is at the root of much of the conceptual and design difficulty for Urbit's networking layer.

We want low-latency, push-based subscriptions. We want low load on large publishers. There is a fundamental tension here.

The correct approach is to provide an interface for applications to implement different strategies in this regard. No networked application can demand absolute upper bounds on delivery latency and absolute reliability.

For example: a chat application might work by eagerly broadcasting avowals of new chat messages, and then beaconing out reavowals in the interim between new chat messages on some configurable but acceptable latency interval. Ten, fifteen, or even thirty seconds of time is not a steep penalty to pay for a dropped packet in a chat application. Of course, this penalty is only paid if the entire message is dropped. If we become aware of a message but we are missing some packets for it, we can always request (see below) an immediate re-send.

A media streaming application, whether for one-to-many broadcast or two-way or many-way communication, may never reavow a binding, since a binding which is known late is no longer relevant.

However, there should of course be a way to request a path from a ship. Thinking of this in terms of "which path exactly must I request?" is incorrect. The information being communicated is "I am interested in a (the latest?) path of this form", which information ought to be avowed, as an oath. This permits replication and caching of interest information, and application-specific but still not onerous handling of the problem of a dead or silent interest.

1.4.3 Command layer

The Ames command layer is, from the view of this system, an application, if a very general and ubiquitous one. A poke takes the form of an oath that you wish to poke another ship with some path and data. The command-layer application insists on re-avowing pokes on a decaying schedule until they are acknowledged. Acknowledgement is, yet again, an oath.

1.5 Routing and discovery, revisited

Alright then. Urbit is conceptually a broadcast network, but we necessarily optimize it by routing. So on what basis to we

route, and to where?

Routing must be a decision taken on registration of interest. By what policy we determine interest may vary, but it should be informed primarily by avowals of interest by the destination. An exception is commands which should be routed unless disinterest is avowed.

The public Internet is not a broadcast network, and we cannot yet communicate over a global network where Urbit has its own ethertype and its own routers. Thus we must be able to select a set of public IPs to send avowals (our own or others) to.

For this reason, galaxies already live in DNS. Commands and registration of interest should by default be routed to sponsoring galaxies and be shared with other galaxies. Galaxies learn IPs of their sponsored stars by receiving such communication. Perhaps the registeration of interest is re-avowed on a heartbeat. Similarly, stars learn IPs of their planets by receiving registrations of interest. Any retransmitted avowal can have several lanes attached in the packet header (stripped before offering to Arvo) by which the avowing ship can be reached. Commands can be routed to the IP of the commanded ship, if known, its sponsor's IP, if known, or its sponsor's sponsor's IP. More general oaths are checked against registered interest (which may be expired if not re-avowed) and sent along known lanes to interested ships.

(In an Urbit maximalist future, BGP-type protocols between Urbit-aware routers will likely permit link-shaped routing of such avowals as well as sponsorship-shaped routing, which can reasonably exist now.)

Local networks, by contrast, generally permit and even depend on broadcast behavior. The Address Resolution Protocol is fundamental to the operation of local IP subnets over lower-layer networks. A packet is broadcast which requests a MAC address for an IP, and provides a MAC address to reply to. At higher layers this is replicated. Peer discovery for local networks at the OS layer (DHCP, zeroconf networking), shared-hardware layer (printers and screens) and application layer (media) rely on on broadcast announcements which receive addressed replies. Commands and interest registrations

could be broadcast on local networks to discover peers either directly, or via Urbit-aware routers.

2 Response

This appendix is a response by ~rovyns-ricfer *to the original document on* ~2022.12.10 *and revised through* ~2023.1.10.

~ritpub-sipsyl wrote up the philosophy for an Urbit networking paradigm that could be used to realize several different dreams:

- universal referential transparency
- transport agnosticism
- global broadcast
- a "narrow waist" that expresses networking in terms of a small number of primitives that compose well
- layering, to reduce the dimensionality of the problem

2.1 Layers

The main layers are:

- PKI
- Transport
- Routing
- Messaging
- Queries
- Frontier
- Commands

2.2 Layer 1: PKI

The PKI establishes identities, keys, and sponsorship information. The PKI is already factored out well into Azimuth. This document does not propose any changes to that.

2.3 Layer 2: Transport

The transport layer is concerned with packetization, transmission control, congestion control, and packet-level authentication. The messaging layer defines the set of messages, i. e. which nouns can be emitted and received by Arvo as semantically meaningful units of information.

All higher layers should be agnostic to the transport layer. This means they can't know anything about the transport layer – things like how a message is broken into packets, how those packets are authenticated, and how congestion control and transmission control are defined.

A user or an application should be able to configure a ship to use multiple transports. A ship could even use different transports for different subtrees of the scry namespace. This configuration can be thought of as a "scry resolution layer" – someone should be able to mirror any part of the Urbit namespace on IPFS, torrents, i2p, Tor, an S3 server, other Urbit ships, or a USB drive, and as long as the requesting ship has the corresponding plugin for that transport protocol, it should be able to access the data.

The default transport layer should be a UDP-based protocol that looks a lot like the current Ames and Fine protocols: 1kB of data per packet, with some kind of packet forgery prevention to protect against denial of service attacks.

One difference between the proposed default transport and Urbit's current packet-level protocol is that the proposd transport will include "advisory" packets, i. e. packets whose emission does not imply its sender Arvo knows the packet was sent. Advisory acks could be used to indicate that the receiving Vere has heard the packet, without implying that its Arvo has heard it – this will enable incoming message fragment packets to be acked immediately without waiting for a disk write. In con-

trast, message-level acks do imply the receiving Arvo ingested and processed the message.

2.4 Layer 3: Routing

The basic idea of the routing layer is that each ship maintains a set of other ships with whom it's currently trying to interact, and the routing layer attempts to find the most direct connection possible with each of those ships.

Routing can be transport-dependent. The scry configuration system that allows for different transports to be used for different subtrees applies to routing-related requests too.

The default transport will use symmetric routing with keepalives. For each ship that my ship is trying to communicate with, my ship's routing system will send a keepalive packet every 25 seconds. This keepalive request packet will go down the receiving ship's sponsorship hierarchy, and the response packet will retrace the same path through the network in the opposite direction.

If my ship does not receive the keepalive response within a short window of time (perhaps a few seconds), it will send another keepalive, this time through the ship's sponsor's last known location. If that does not yield a prompt response, my ship sends a keepalive to the sponsor's sponsor's last known location, with this chain terminating in the sponsoring galaxy.

Keepalive responses contain a list of transport addresses that can be used to reach the responder ship. This list will generally include addresses of the sponsorship hierarchy, appended by relays later in the response chain.

(Aside: [Do we want] push-based reconnect? or do we not need that explicit query for sponsee address? Implicit is more efficient and maybe simpler.)

Arvo itself will not maintain an up-to-date list of transport-level addresses for each of its peers. Instead, off-loop code (i. e. part of Urbit whose state is not persisted as part of the Arvo core and whose execution does not occur within Arvo's event loop) will be responsible for maintaining those addresses. For sovereignty reasons, it's good for these systems to occasionally inform Arvo of the known direct addresses for other ships, but

that can be done every few minuteswithout hitting the event loop on every state update.

2.5 Layer 4: Messaging

The message layer is the "narrow waist" between the different parts of the system. Every message is simply an `$oath`: a signed pair of `[path noun]` representing a scry binding – the path is permanently bound to that noun, and the signature attests to this permanence.

An oath is quite general – it could mean many different things. The higher layers of the system build conventions into the namespace for interpreting oaths whose paths fit certain patterns.

2.6 Layer 5: Queries

The query layer establishes a convention for how one scry binding can be interpreted as a request to download the value bound to a different scry path on another ship. This is a "remote scry protocol". The fundamental interaction consists of a subscriber ship emitting an oath representing a scry request for the value at a fully qualified path, then receiving a response oath from another ship that contains the requested scry binding.

2.7 Layer 6: Frontier

The frontier layer extends the query layer to express a request for the latest value at a path – this can't be a fully qualified scry path, since that implies a concrete `$case` (revision number or date). Instead, the path in such a request has a wildcard character in place of a `$case`, indicating the request is a `%pine`, which can be resolved by any binding whose path matches the path in the request.

The ship receiving the `%pine` will respond with the latest binding it has that matches the request. Note that for ephemerally bound oaths, such as requesting a piece of data from a Gall

agent at the latest date, this operation must be atomic. Breaking it up into two requests (the first to retrieve the case, the second to grab the value at the path with that case) will fail unless both requests are performed on the same Arvo state.

A `%pine` should only be used when the last case is unknown. Otherwise, a request with a concrete case should be used. This increases the degree of effective referential transparency in the networking – since everything is an oath in this system, everything is at least trivially referentially transparent, but it's good for the referential transparency to be pervasive at every layer, rather than merely trivial. Making more of the requests referentially transparent yields a more stable, cacheable, analyzable system.

2.7.1 Layer 6b: Publications

A further layer on top of queries and `%pine` is publications. A publication is solid-state, meaning it traces a sequence of scry bindings across the namespace, with an incrementing numeric case for each update.

On initial subscription, a subscriber emits a `%pine` to request the latest rock. Once it has the rock at revision n, it scries for wave $n + 1$ – not as a `%pine`, but as a scry request.

The publisher responds in one of three ways to this request: with a scry response if it has the data, with a `%nigh` response if it doesn't have the data yet, or with a `%yore` response if it deleted the data. A `%yore` or `%nigh` response would bind a path including the original path, but the publisher's current dateas the official `$case`.

If the subscriber hears a scry response, it requests the next wave, n+2. If it hears a `%nigh` response, it re-sends the request 25 seconds later. This request should be "sticky", so that if the publisher binds the requested path within 25 seconds of hearing the request, it will immediately route the new binding to the subscriber.

If the subscriber hears a `%yore` response, that means it has fallen behind the publisher, which has deleted the wave it asked for. The subscriber falls back to a `%pine` for the latest rock, just like in initial subscription.

2.8 Layer 7: Commands

A command consists of an oath broadcasting a command from the emitting ship to another ship, followed by an oath from the other ship broadcasting the ack, which is either empty, indicating the receiver performed the command, or a `$tang` error message describing why the command was not performed.

Each command is sent on a "flow", identified by a number called a `$bone`. Within a flow, commands are delivered to the receiving application in order, with exactly-once delivery at the application level unless one of the two ships breaches.

The transport and routing layers need to know this is a command, so that it can be routed to the receiver despite the receiver not having first emitted an interest oath – every ship is considered to have implicitly registered interest in commands to it.

Note that this design would allow commands to use the same encryption scheme as other broadcasts. This would likely mean the kernel would use a cryptographic ratchet to maintain a secure channel between publisher and subscriber, which the publisher would use to send the subscriber the symmetric keys used to encrypt and decrypt various subpaths within the namespace it's publishing. Since commands will just be one kind of scry binding, they could reuse this key distribution system for forward secrecy.

(Aside: Which ratchet makes the most sense – double ratchet?)

This design also suggests that in the default transport, the receiver of the command would be the ship performing congestion control, which is more polite than the present situation, which is the reverse. All large messages will be pulled, rather than pushed, including commands, giving the ship ingesting the data control over how it obtains the data and at what rate. Only the first packet of a command will be pushed, to notify the receiver the command can be pulled.

2.9 Message Protocol – Interest Paths

Paths of a certain format are recognized by Vere as "interest paths", which register interest (or disinterest) in the value bound to another path. These paths all use `%a` as their `$vane`, representing the Ames vane.

2.9.1 Request Types (`$care`s)

- `%a`: fetch a value at a fully qualified path (remote scry)
- `%b`: fetch the latest value (`%pine`)
- `%c`: send command (`%poke`)

2.9.2 Response Types

- `%d`: poke ack
- `%z`: metadata about whether a path is bound and available

2.9.3 Remote Scry Motif

The remote scry motif implements a remote query at a fully qualified path. It is primarily a request/response system where a request for the value at a path yields a single formal response containing the requested value. This motif also includes advisory responses that the publisher can emit as a courtesy if it is not sending the formal response at this time.

Request

```
path:  /~rovnys/1/2/a/a/~2023.1.10/~zod/2/3/c/x/4
    /base/sys/hoon/hoon
value: &
```

The value bound to a scry request path is a flag indicating interest if yes, or disinterest if no. Disinterest can be used to cancel a previous request.

Positive Response (Permanent, Value is Present)

```
path:  /~zod/2/3/c/x/4/base/sys/hoon/hoon
value: [~ %hoon '::  hoon 141k\0a...']
```

The value in any permanent response is a `(unit [mark noun])`.

Negative Response (Permanent, Value Is Absent)

```
path:  /~zod/2/3/c/x/4/base/nonexistent/hoon
value: ~
```

A permanent negative response binds the requested path directly to a null value.

Blocked Responses

A blocked response represents a refusal to answer the scry request. Any blocked response binds a temporary path to a value indicating the kind of refusal. The path is bound at the current date with a `%z` care, and it includes the original scry request path as the spur.

Deleted

This response is emitted if the publisher has deleted the value at this path.

```
path:  /~zod/2/3/a/z/~2023.1.10/~zod/2/3/c/x/4/base
    /sys/hoon/hoon
value: %yore
```

Not Yet Bound

This response is emitted if the publisher knows it might bind the path later, but has not yet bound the path to any value.

This can happen if the subscriber asks for a Clay file or publication data at a future `$case`.

```
path:  /~zod/2/3/a/z/~2023.1.10/~zod/2/3/c/x/4/base
    /sys/hoon/hoon
value: %nigh
```

General Refusal (Other)

There are other reasons the publisher might not want to respond: for example, the request could be malformed, causing a crash in the scry handler; or the value might not have permissions to be read over the network. This response is a catch-all that does not provide any information about why the request was denied.

```
path:  /~zod/2/3/a/z/~2023.1.10/~zod/2/3/c/x/4/base
    /sys/hoon/hoon
value: %deny
```

2.9.4 Fetch at Latest (`%pine`) Motif

The `%pine` motif implements a "query at latest" request/response flow. The requester asks for the latest value at a scry path whose `$case` (revision number or date) is a wildcard (?). The request has a different `$care` to indicate it's a `%pine` rather than a scry request. The publisher responds with a path that matches the request predicate, with the latest response it can, as a direct binding, or a generic blocked response.

Request

```
path:  /~rovnys/1/2/a/b/~2023.1.10/~zod/2/3/c/x/?
    /base/sys/hoon/hoon
value: %ud
```

The value of a `%pine` request indicates which kind of `$case` it wants in the response path. A case can be either `%ud` for

numeric revision, `%da` for date, `%tas` for label, or `%uw` for hash. If the publisher can't bind a path at the requested `$case` type, it will emit a general refusal block response.

Response

The `%pine` response message is the same as a scry response message. Its path has a concrete case, not a wildcard.

```
path:  /~zod/2/3/c/x/4/base/sys/hoon/hoon
value: [~ %hoon '::  hoon 141k\0a...']
```

2.9.5 Command (`%poke`) Motif

Request

```
path:  /~rovnys/1/2/a/c/1.234/~zod/2/3/a/d/1.234
    /~rovnys/36
value: [/g/landscape/chat/my-channel %add-post
    'first!!1']
```

Here 36 is the `$bone`, i. e. the number of this "flow" between the two ships. `1.234` is the message sequence number, which is also the case of the request path. The poke request can be thought of as expressing interest in the ack message bound by the receiving ship.

Positive Response (Ack)

```
path:  /~zod/2/3/a/d/1.234/~rovnys/36
value: &
```

If the receiver performed the command, it emits a response on the requested ack path with a value of `&` ("yes").

Negative Response (Nack)

```
path:  /~zod/2/3/a/d/1.234/~rovnys/36
value: [| 'chat %add-post failure:' 'no permissions
    for this channel' ~]
```

If the receiver declined to perform the command, it emits a response on the requested ack path containing a `|` ("no") and a `$tang` datum containing either a stack trace or an application-level error message. Note that this is not intended as a general-purpose response mechanism, i. e. it is an anti-pattern to dispatch on the stack trace programmatically.

(Aside: Should there be a way to dispatch on the kind of error? Userspace devs keep asking for this. We could easily make the poke motif a special case of a more general request/response motif. That arguably violates CQRS, but it's a very useful pattern, both for errors and for things like responding to a command with a ticket number that the requester could use later to check in on the status of an ongoing async operation that this command kicked off.)

(Aside: Should there be an explicit disinterest message that can be sent in response to a poke, as an advisory scry binding? This could be useful for mitigating DoS attacks, and it could even specify an explicit backoff interval.)

2.10 Handshake Motif

Any two ships need to able to negotiate a symmetric key to distribute other symmetric keys that will be used to encrypt scry `$spurs` (the last part of the scry path, after all the standardized path elements) and bound values.

An encrypted spur looks like `/14/0w2.CESjp.GdASq.zpdVc.b4Y1T.JXcY1`, where `14` is the number of the key used to encrypt the second element of the path, which is the result of `jamming` (serializing) and then encrypting the original spur. Any of the spurs used as examples of remote scry and `%pine` in this document could be replaced by a spur of this form without loss of generality. Poke paths are not encrypted, only their values.

(Aside: Should poke values always be `[key-num encrypted-value]` just like encrypted spurs? That might be nice, to avoid race conditions between the handshake protocol and other pokes.)

Since cryptography evolves, it cannot be kelvin versioned. Urbit's handshake protocol should instead have an incrementing version number. As of this writing in 2023, modern protocols use various kinds of "cryptographic ratchet" to cycle keys frequently, providing forward secrecy and sometimes other desirable privacy properties. These ratchets often involve "ephemeral Diffie-Hellman" key exchange and "key derivation functions (KDFs)" to limit how far back or forward an attacker could see if they crack any single key.

The network messages to request a symmetric key used to encrypt scry `$spurs` can use the normal `%pine` and remote scry motifs. The `$spurs` involved in the request itself will be encrypted using the latest key negotiated by the handshake protocol.

Any handshake protocol can be expressed as pokes back and forth between the two ships, encrypted using the latest known key negotiated by the handshake protocol. So that both ships know that a poke relates to a handshake, we could reserve bone `0` for the handshake protocol.

The first implementation of this networking design will likely triage the handshake and use the Diffie–Hellman of both ships' static keys as a permanent symmetric key. This does not provide forward secrecy, but keeping the handshake protocol orthogonal from the rest of this suite of protocols allows a nontrivial handshake protocol to be added later without changing the rest of the design.

2.11 Routing

Note that since routing is transport-dependent, there are no message-level semantics, other than Arvo emitting oaths to its local Vere to indicate interest or disinterest in routing messages to a ship.

```
path:  /~rovnys/1/2/a/a/~2023.1.10/lane/~wicdev/5/6
value: &
```

When Vere hears this oath, it is responsible for finding wicdev using whatever transport protocol is indicated for that part of the namespace. The rest of this description applies only to the default UDP transport.

Note: in a certain sense, we don't strictly need this, since other requests could be used to infer interest or disinterest in the locations of other ships. The reasons to include it anyway are to:

1. ensure orthogonality of routing and other layers; and
2. to allow a ship to deliberately ask its Vere to find a peer for it, which is useful for sovereignty and network diagnostics.

2.11.1 Keepalive Request

A keepalive request looks like a `%pine`, generated by Vere, with a `%e` synthetic `$care` indicating interest in a keepalive response:

```
path:  /~rovnys/1/2/a/e/~2023.1.10/~wicdev/5/6/a/f
    /?/keep
value: &
```

2.11.2 Keepalive Response (Direct)

The "direct" (i. e. originated by the target of the keepalive request) response to a keepalive request has a path containing a `%f` synthetic `$care`, at the publisher's latest date. Its value contains a list of IPv4 addresses and UDP ports, whose sole element is the IP and port of the ship itself, as reported to it by its sponsor.

```
path:  /~wicdev/5/6/a/f/~2023.1.11/keep
value: [~ %udp-ipv4 [%& .192.168.1.1 1.337] ~]
```

2.11.3 Keepalive Response (Relayed)

If a relay hears a keepalive response, it wraps the result in its own address, prepending its own address to the list of lanes and including the signature of the wrapped response in its own response. The relay address can then be used by the original requester to "tighten" the route it uses in later keepalive requests.

```
path:  /~dev/7/8/a/g/~2023.1.12/fore/~wicdev/5/6
    /a/f/~2023.1.11/keep
value: :+  ~  %udp-ipv4
       :~  :-  %&  .123.456.789.0
           :+  %|
5            0x<signature>
           :^    ~
              %udp-ipv4
            [%& .192.168.1.1 1.337]
           ~
10     ==
```

Ad Fontes has proven influential on the implementation of the remote scry protocol, `%fine`*, and the directed messaging project of Urbit 410 K.*

References

~sorreg-namtyv, Curtis Yarvin (2010) "Urbit: functional programming from scratch". URL: `http://moronlab.blogspot.com/2010/01/urbit-functional-programming-from.html` (visited on ~2024.1.25).

~sorreg-namtyv, Curtis Yarvin et al. (2016). *Urbit: A Solid-State Interpreter*. Whitepaper. Tlon Corporation. URL: `https://media.urbit.org/whitepaper.pdf` (visited on ~2024.1.25).