The State of Urbit: Eight Years After the Whitepaper

Ted Blackman ~rovnys-ricfer
Urbit Foundation

Abstract

The Urbit whitepaper was released in 2016. In the intervening eight years, the project has become much more fleshed out: the sketch has started to take on color and detail. This article explores the major developments since the whitepaper, including changes to the runtime, the network, the kernel, and the userspace. It also discusses the current state of the project and intended refinements.

Contents

Urbit Systems Technical Journal I:1 (2024): 1–46.

Address author correspondence to ~rovnys-ricfer.

1 Introduction

The Urbit whitepaper (~sorreg-namtyv et al., 2016) publicly elaborated the technical prospectus for a solid-state operating function, an operating system built as a simple state machine defined as a pure function of its events. While much of the whitepaper was concerned with establishing the philosophical objectives of the project, there was also substantial exposition about key parts of the Arvo event handler, the language, and

the network. Given that the whitepaper was a snapshot of the project and its ambitions as it stood in 2016, how has that vision held up in practice, and how has the project fared?

While the whitepaper by its own admission presented itself as a "work-in-progress presentation" of a "semi-closed alpha test", the basic claims of the Urbit whitepaper are still true today. Most of the intervening work over the past eight years has gone into improving robustness and better support for applications. A capsule history of "late early" Urbit includes the following milestones:

- 2015–2017: Early vane explosion
- 2017–2019: Rewrites of Hoon, Ford, Eyre, and Vere refine the system
- 2018–2019: Rewrites of Azimuth and Ames refine the network stack
- 2018–2019: Nock bytecode interpreter
- 2020: Ford Fusion enables network continuity
- 2020: Security audit on Ames
- 2021: Azimuth Layer2
- 2021: Software distribution introduced
- 2022–2024: Software distribution maturation
- 2023–2024: Network scaling

This article will discuss the changes to Urbit since the whitepaper, focusing on the runtime, the network, the kernel, and the userspace. It will also discuss the current state of the project and intended refinements.

2 Changes to Urbit

Briefly, the lower layers of the system have deviated very little from their descriptions in the whitepaper. Nock (the machine

code), Hoon (the programming language), and Arvo (the operating system kernel) – which implies atomic event processing and orthogonal persistence – are very much the same system today as they were even a decade ago.

Since 2016, Nock has undergone a revision from kelvin version 5K down to 4K. This was a refinement rather than a deviation, mainly adding an opcode to manage tree mutation (see appendix for details). Hoon has undergone a number of changes but is certainly recognizable as the same language.

At the next layer up, the Arvo kernel, despite having been rewritten at least once, serves the same purpose and has a similar shape to the one that existed at the time of the whitepaper.

The system has primarily changed via addition, mostly growing upward in the stack: runtime and kernel infrastructure to support distributing, installing, running, and upgrading userspace applications. Changes have been made to peer-to-peer networking, HTTP serving, source code management in Clay, and Arvo kernel upgrades. Permanent network continuity was achieved at the end of 2020, largely by deleting the Ford build system vane, integrating the build system into the source control system in the Clay vane, and modifying the Arvo kernel's upgrade semantics.

Urbit's "scry" data namespace is also much more real and fleshed out than it was when the whitepaper was written. The concept of scrying predates Urbit itself (`~sorreg-namtyv` (2006); compare `~sorreg-namtyv` (2010)), but was not used heavily by applications.

The other big change is the public key infrastructure (PKI). The whitepaper's PKI, which was effectively live from 2013 to 2019, had no double-spend protection and was effectively centralized. To fix this, the PKI was moved to a set of Ethereum contracts. Going on-chain made the ownership ledger for Urbit IDs censorship-resistant … until a year later when high transaction fees on Ethereum brought network growth to a standstill and the core developers were forced to write a custom "naive rollup" Ethereum contract. This brought the price of spawning a new Urbit node down from hundreds of dollars to under a dollar, at the cost of on-chain interoperability.

Going on-chain also enabled "personal continuity

breaches", in which a ship (instance) owner can notify the network through the chain that their Urbit node was rebooted from scratch, due to data loss, unrecoverable error, or compromised security. This feature necessitated some rework of the kernel to ensure data consistency across vanes when receiving a notification that a peer had breached.

Finally, a ton of work has been done on Urbit's runtime. Lots of 2016 vaporware is corporeal in 2024.

2.1 Solid-Stateness

> Urbit is a "solid-state interpreter": an interpreter with no transient state. The interpreter is an ACID database; an event is a transaction in a log-checkpoint system. (~sorreg-namtyv et al., 2016)

Urbit is a solid-state interpreter but some of its higher layers are more solid-state than others. The scry namespace, hardly mentioned in the whitepaper, provides the primary means of preserving solid-stateness higher up the stack, especially into userspace.

A program is considered more solid-state if it has fewer moving parts, fewer complex asynchronous code paths, less transient state, more idempotence, more immutability, more referential transparency, clearer transaction boundaries, less ordering dependence, more declarativeness, and more determinism.

Much of Urbit's kernel modules and application model were originally written in ways that were less solid-state than they could have been. As the reliability requirements for Urbit have increased, more attention has been devoted toward making the system more solid-state, as the main pathway toward increasing simplicity and reliability. More attention has been paid to the scry namespace in particular, to increase its capabilities to allow it to be used more heavily.

2.2 Clay "Theory of a Desk"

A relatively early change that increased the solid-stateness of the system was called the "Theory of a Desk". This was implemented as part of the Ford Fusion project in early 2020. Ford, the build system, was moved from being a standalone vane (kernel module) into the Clay source management vane. When Ford stood alone, despite – or maybe exacerbated by – multiple rewrites, it was difficult to achieve full correctness of source builds during a kernel upgrade. It was common for post-upgrade files to be built erroneously using the old kernel. This behavior was effectively nondeterministic because it relied on the ordering of short hashes ("mug"s) that identified subscribers to Clay file-changed notifications. Sometimes it would work correctly by luck; other times it would fail in various hard-to-reason-about ways.

Moving Ford into Clay removed asynchrony from the system: instead of Ford requesting files one by one from Clay and building them, Ford became a subroutine in Clay that Clay could call synchronously. This "fusion" allowed the Ford build semantics to be modified to depend only on files in the current revision of the "desk" (Clay's version of a Git repository).

Previous versions of Clay only allowed files of a newly defined filetype ("mark") to be added *after* a commit that added a "mark file" that defined the semantics of that filetype. This meant that no Clay revision was ever truly self-contained: a deterministic understanding of how its files would be validated required knowledge of the previous commit as well.

The "theory of a desk" paradigm uses only the current snapshot of the desk to build any mark files needed to validate the files in the desk. This was not technically impossible while Ford was a separate vane, but the asynchronous communication back and forth made it exceedingly difficult. The new Ford shrank from 6,000 lines of code down to about 500, and became much easier to reason about and debug.

A year later, near the end of 2021, the "software distribution" project completed, allowing third-party developers to distribute userspace applications to their users over the Urbit network, storing and building the code in Clay. This necessitated

reproducible builds, which is a notoriously solid-state requirement. Ford Fusion builds were reproducible, but the system had to be extended quite a bit to practically support multiple userspace installations. These extensions caused a new set of problems.

The interaction between Gall (which runs applications) and Clay (which stores and builds their source code) was also complex, asynchronous, and bug-prone. A year after software distribution, in the second half of 2022, the "agents in Clay" project moved the locus of control over which userspace applications were supposed to be running from Gall itself to Clay. This allowed Clay to send a single "move" (effect) to Gall containing the full list of all agents that Gall should now be running.

Note that this latest iteration, with a single large effect, is more declarative, has less transient state, is more deterministic, has less ordering dependence, and is more idempotent – in other words, significantly more solid-state. As expected, this system has been much more reliable than the initial software-distribution system.

2.2.1 Remote Scry Protocol

A major part of making the namespace more usablae was to expose namespace queries over the network. Designed in 2021 and deployed in 2023, the "Fine"[1] remote scry network protocol was added to Urbit. The first use cases were Clay, which uses it for downloading desks, and the Gall `%grow` namespace, described in more detail in a later section.

This protocol lets one ship fetch data at a namespace path bound by another ship. The runtime includes a cache so later requests for the same data are typically served from the runtime cache rather than requiring Arvo's `+peek` arm to be called again. In addition to the scalability benefits this provided, it also represented a fairly large conceptual shift: instead of Urbit's networking only being able to express commands (writes) over the network, it could now also support reads.

[1] Pronounced *fee-NAY*, after French cartographer and mathematician Oronce Fine.

These reads are from the immutable namespace, so they're quite solid-state.

Two-Party Encrypted Remote Scry: `%chum` The remote scry protocol authenticates responses (scry bindings), but it does not have any concept of encryption built in – everything published over the network is public. Since most Urbit applications deal with private data, there needed to be a way to distribute data privately. The `%chum` namespace overlay was the first way to do this.

The `%chum` namespace overlay allows a ship to make an authenticated and encrypted scry request to another ship. Only the publisher ship can decrypt the request, which is symmetrically encrypted using the Diffie–Hellman of the two ships' Azimuth networking keys. The scry response is also encrypted symmetrically, so only the requesting ship can decrypt it. Crucially, this system uses the Arvo kernel's security mask (which was described in the whitepaper but not enforced until this protocol made use of it) to inform the publishing module about which ship is making the request.

The first version of this system uses the Fine protocol unmodified, although the Directed Messaging project's implementation has explicit support for symmetric encryption, which it also uses for pokes (commands).

Multi-Party Encrypted Remote Scry The symmetric encryption of the `%chum` namespace overlay has a problem in a multi-party context like a social group: no two requesting ships can share entries in the runtime cache, meaning each subscriber has to make a separate call to Arvo's `+peek` arm. To fix this, a multi-party form of encrypted remote scry was added.

To use this system, a publishing application must register a scry path prefix as a security domain, to which the kernel associates a (rotatable) symmetric key. Subscriber ships request access to this key. The publisher ship's kernel asks the publishing application whether that ship should be allowed to view the content at that path prefix. Once the subscribing ship has the key for a path prefix, it requests data under that prefix by

encrypting the request path and placing the key's identifier in cleartext so the publishing ship knows which key to use to try to decrypt the path.

Querying the publishing application dynamically like this allows the publishing application to avoid materializing the set of all ships who are allowed to see this content. Materializing that set in a complex application with layers of roles, whitelists, blacklists, and other criteria could be prohibitively slow, superlinear in the number of allowed subscribers. This arrangement should allow the application to keep the query time to roughly logarithmic in the number of subscribers, which is much more practical.

The next steps in increasing the utility of the scry namespace include `%pine` (a non-referentially-transparent "request-at-latest" network protocol) and "sticky scry", which would facilitate scry-based network subscriptions by allowing requests for unbound data to persist on a publisher until the data is published, at which point it will be pushed down to subscribers with low latency.

2.3 Gall `%grow` Namespace

Gall agents were not able to directly bind data into the scry namespace until the `%grow` namespace was added in early 2023 in the same release as the first remote scry protocol. A Gall agent can emit an effect to bind a path to a value in the `%grow` namespace.

This allows applications to publish data that can be read remotely using the remote scry protocol, making applications more legible and solid-state.

2.4 Persistent Nock Memoization

For around a decade, Nock interpreters have supported the `%memo` memoization hint, informing the interpreter that the enclosed Nock computation should be cached. Since Nock is purely functional, these caches will never need invalidation, an important property. However, until early 2024, runtime Nock

caches were ephemeral – they did not last past a single Arvo event, or were even more short-lived.

Persistent Nock caching allows for removal of most caching-related state from the kernel and applications, allowing code to focus on business logic. Making more logic stateless contributes strongly to solid-stateness. Persistent Nock caching also opens up new architectural possibilities, such as userspace build systems that share a runtime cache to avoid rerunning compilations or duplicating libraries, even if they have varying semantics.

2.5 Breadth-First Arvo Move Ordering

Arvo has traditionally used "depth-first" ordering when processing "moves", i. e. messages passed between parts of the system. Since each module in Arvo emits a list of moves whenever it runs, and then the moves are delivered one at a time to their destination modules for processing, at any given time there is a tree of unprocessed moves that have been emitted but not delivered.

Processing moves in depth-first order is planned on being changed. A version of Arvo that instead uses breadth-first move order has been written and is intended for deployment in Zuse 410 or 409. This remains somewhat controversial, though.

Depth-first move ordering has the advantage of a certain kind of consistency. Consider the case where as the result of a single activation, Gall agent `%foo` "pokes" (sends a command to) `%bar` and another to `%baz`. With depth-first move order, any moves emitted by `%bar` upon receiving `%foo`'s poke will be processed before `%baz` receives `%foo`'s poke. This applies recursively, so if upon receiving `%foo`'s poke, `%bar` poked `%qux` and `%qux` poked `%qux2`, all that still happens before `%baz` receives `%foo`'s poke.

This means `%foo` can rely on `%bar` completely handling the first poke before `%baz` receives the second poke, even if `%bar` has to issue further commands in order to do so – at least as long as all the pokes are, one, domestic (in the same ship) and two, processable synchronously, i. e. no operations need to be spread across multiple Arvo events, such as waiting for a timer

to elapse or waiting for a user to type input on the command line. If both these conditions are met, then the poke to `%bar` has one view of the system's state, and the poke to `%baz` has a newer, updated view, but both views are consistent in the sense that for either one of them, `%bar`'s move processing has either not started at all or completely finished.

A major issue with depth-first move order is that many common interactions, including anything social that involves peer nodes, cannot preserve both of those properties that let each module send commands to other modules without acknowlegment but still rest assured that those commands will complete in order and with each having a consistent view of the state. This means the system has needed to incorporate the concept of an acknowledgment for a poke, since a poke could be over the network rather than local. In the current formulation, local pokes are also acknowledged, losing some of the streamlining one would hope for with depth-first ordering.

Another problem with depth-first ordering is that it makes consistency among modules difficult to guarantee. Jael delivers a notification that a peer has breached to Ames and Gall. When Ames hears the notification, it deletes all state regarding the old pre-breach ship and updates its state to store a cache of the peer's new post-breach PKI state (public key, etc.). When Gall hears the notification, for each agent that had subscriptions to the old ship, it closes the subscriptions and notifies the agent that the subscription was closed.

The problem comes up when an agent wakes up with the notification that its subscription was closed and emits a move to resubscribe. Gall then sends this move to Ames. If Gall heard the breach notification before Ames, then the resubscribe request would be sent to the old pre-breach ship, since in depth-first move ordering, Ames hears Gall's requests before Jael's breach notification. Since the ship had breached, it ignores the resubscribe request to its old key.

This can be band-aided by baking the specifics of the interaction into Jael, so it knows to tell Ames about the breach before Gall. This approach works as long as the data dependencies among modules remains acyclic, but as soon as two modules could both send moves to each other upon being notified

– meaning they need to be notified "at the same time" – it becomes nontrivial to provide consistency. It could be achieved through a two-phase commit system where Jael delivers two moves for each notification, and the receiving modules would know to update their state in response to the first notification but only emit moves upon hearing the second notification.

The final problem with depth-first ordering is that it is empirically difficult to reason about. Moves are not processed chronologically – later moves can jump to the front of the queue. This has led to a number of subtle bugs due to later moves invalidating earlier moves, such as canceling a subscription that hadn't been established yet because the unsubscribe move jumped in front of the subscribe move.

An alternative has been built which uses breadth-first move ordering. This is strictly chronological, does not reorder moves, has a concept of simultaneity for maintaining consistency when delivering notifications, and works equally well locally and over the network. The major disadvantage of breadth-first ordering is that there is no automatically growing transaction: everything requires explicit acknowledgment.

The latest "shrubbery" (q. v.) experiment uses a hybrid move ordering: depth-first for local pokes within the same application, and breadth-first otherwise.

2.6 Shrubbery

"Shrubbery" refers to an architectural concept in which all userspace data is addressible in the scry namespace. This would include source code and application state in addition to inert file-like data.

A well-functioning shrubbery vane would replace Clay and Gall, unifying the storage and management of data and code into a single namespace-backed system.

A shrubbery model has the potential to make application programming significantly more solid-state. Each piece of data in a shrubbery system has a canonical name to which it is immutably bound. The state of pending I/O will also live in the namespace, facilitating inspection and recovery after a suspension or breach. Subscription and state synchronization become

declarative. Functional reactive programming features would allow for declarative composition of programs.

2.6.1 Move Ordering in Shrubbery

The latest "shrubbery" experiment attempts a hybrid solution to get the best of both modalities. A typical shrubbery "app" will be broken into a hierarchy of event handlers at various subpaths. When sending a move "downward" into a subpath, the system will handle that move depth-first, since an app is presumed to know the characteristics of its sub-apps. These moves will not need to be acknowledged, since they will be truly depth-first. When passing a move "upward" or "sideways", i. e. outside of the sub-tree, that move will be appended to one global queue, to be handled breadth-first.

This hybrid approach can be thought of as the lowest layers of the move tree being depth-first, with breadth-first at higher layers. Core developers expect the shrubbery experiment to be runnable within a month of this writing.

3 Continuity

Since the whitepaper was written in 2016, quite a bit of effort has gone into improving Urbit's continuity. Continuity in this context might best be defined in contrast to its opposite: a "continuity breach" refers to the act of restarting an Urbit from scratch, throwing away all its previous data and the states of its connections with other ships.

Until the end of 2020, core developers would routinely conduct network-wide breaches. Every ship on the network would need to be restarted from scratch in order to communicate with other ships.

For a user, the experience of discontinuity is one of loss of data and possibly even of friends. This document will use the term "continuity" in a broad sense.

3.1 Azimuth: On-Chain PKI (2019)

One major event that improved the level of continuity was moving Urbit's public key infrastructure (PKI) from an informal spreadsheet maintained by the Tlon Corporation to a set of smart contracts on the Ethereum blockchain (`~ravmel-ropdyl`, 2019).

This ensured Byzantine fault-tolerant ownership of an Urbit ship and permissionless entry – anyone can buy an Urbit address, without needing to communicate with any particular person or organization. This had always been the goal, but it wasn't feasible until programmable smart-contract blockchains became production-worthy.

In addition to moving the address ownership and sponsorship information on-chain, other features were added.

3.1.1 Azimuth: Individual Continuity Breaches

The capability to perform an "individual continuity breach" was added. Before going on-chain, if an Urbit suffered any major data loss, it would be permanently dead and could never communicate with peers again. An individual continuity breach allows a user to post a transaction to chain that increments the "rift" number for their address, signaling to the network that the ship has been restarted from scratch. Peer ships that were storing message sequence numbers for the breached ship reset their networking state for the breached ship.

This allows a user to continue to use the address they purchased even if their Urbit loses data. The newly written `%phoenix` app implements a system of encrypted backups, so that after a breach, a user can recover application data from the latest backup (`~midden-fabler`, 2024).

3.1.2 Azimuth: Constitution

The Urbit galaxies have always been considered to form a senate, acting as the Shelling point for network governance. Before going on chain, this was merely an idea – now it is a formalized reality. Galaxies can vote on-chain, using a smart contract that has control over the other smart contracts.

There are two ways galaxies can vote: on (the hash of) a document to signal support of that document, and on a new constitution contract, which will trigger the old constitution contract to cede control to the new constitution contract that replaces it. This means the galaxies can vote to upgrade the constitution as they see fit – an elegant example of social meta-circularity.

It should be noted that galactic governance is purely voluntary: because users have full control over their own personal servers, they retain the power to "vote with their feet" and decide to use a different PKI or a different governance structure.

It is incumbent upon the galaxies to make decisions that are prudent enough not to fork the network – or prudent enough to fork the network, if called for. Using a blockchain means a fork could be performed in an orderly manner without losing consistency. This added technical feasibility of forking puts virtuous pressure on the galaxies to govern reasonably.

3.1.3 Azimuth: Naive Rollup

In 2020, Ethereum transaction fees rose to levels high enough that Urbit network growth ground to a halt. The core developers, after evaluating then-nascent "zero-knowledge" and "optimistic" rollup technologies for reducing fees, determined they were not yet production-worthy and wrote a simpler rollup that they called a "naive rollup". This brought the price of spawning a planet down from roughly $100 to under $1.

The naive rollup treats Ethereum as an inert data store for storing a series of transaction inputs. Each Urbit ship then reads those transaction inputs from Ethereum and executes them itself, arriving at the latest state of the rollup. This can be thought of as a simplification of an optimistic rollup, where the chain does not have any representation of the latest state of the rollup, only the inputs that would lead to that state. This obviates the need for fraud proofs and bond slashing, yielding a substantially simpler system at the cost of some interoperability between Urbit addresses in the rollup and other Ethereum contracts, which do not have any cryptographic proof on chain regarding which Ethereum address owns those addresses.

Each ship's Azimuth state is stored either in the rollup or in the original contract. Galaxies cannot be moved into the rollup. For the time being, addresses cannot be moved back out of the rollup.

3.2 New Ames & New Jael (2019)

In order to take advantage of the new PKI, the Ames networking vane and Jael secret storage vane both had to be rewritten. This was done in 2019, with new Ames being deployed later, near the end of the year.

The "new" Ames and Jael – now in their fourth year of use, with Ames having had some refactoring and extension – support individual continuity breaches and on-chain sponsorship changes ("escapes").

3.3 Ford Fusion (2020)

As described earlier, the Ford Fusion project moved Urbit's build system from a standalone vane to a synchronously callable function inside Clay and rectified over-the-air Arvo kernel upgrades.

This was crucial for performing the last network breach at the end of 2020, establishing permanent continuity – since then, many dozens of over-the-air upgrades have been deployed. Without a reliable upgrade process, this continuity era would be infeasible.

3.4 Essential Desks (2023–24)

Having gained experience with managing deployments of both kernelspace and userspace desks, it became clear that the original way that kernel updates synchronized with app updates needed more flexibility. Any app could block a kernel upgrade if its developer had not also pushed out a new version that was compatible with the new kernel. A user could always suspend a stale app to allow the upgrade, but this had to be done manually.

A later update modified this behavior. A newly installed app is now marked as non-essential by default. The kernel will only block an upgrade if an essential app is stale. A stale non-essential app will be suspended automatically on upgrade. If a user wants to ensure an app will never be suspended, they can mark it as essential.

3.5 Future Work

The next major frontier in continuity will be "kelvin shims", i. e. a kernel that can keep stale apps designed to work with previous kernels running (through a "shim" for a previous kelvin version), not just apps designed to run at exactly the current version. This is a thorny problem that the core development team experimented with in early 2023 but abandoned.

The hard part of kelvin shimming turned out to be that the current version of the userspace/kernelspace API is complex. Taming that interface and localizing application-related state to make it easier to migrate will be critical for enabling kelvin shims. The shrubbery experiment is a step in this direction.

More sophisticated kernel support for migrating applications is also likely to be a fruitful line of work. This involves migrating application state from an old type to a new type, as well as supporting protocol negotiation to upgrade or downgrade commands and subscription updates to maintain compatibility between old and new versions of the same app on different ships.

4 Maturation

Urbit has matured along several dimensions since the whitepaper was written. This work can be roughly categorized into organizational changes, scaling improvements, reliability work, work on the runtime, and adding an explicit control plane.

4.1 Changes to Development Practices

Since 2016, a much larger number of developers are involved in core development, and the development practices have evolved accordingly.

4.1.1 Organizational Changes

In early 2023, the responsibility for core development transferred from the Tlon Corporation, which had managed it since 2013, to the newer Urbit Foundation, a nonprofit. The Urbit Foundation has become the custodian of the Urbit kernel and ecosystem. UF adopted and continued to modify Tlon's release workflow, including CI/CD (continuous integration/continuous deployment), multi-stage canary deployments, and repository management practices. This process has more recently been extended to include a liaison period for each kelvin update where organizations that use Urbit can use the prerelease software on their own infrastructure first and report any bugs to UF.

4.1.2 Urbit Improvement Proposals

The Urbit Improvement Proposal system was introduced in mid-2023. This is an RFC system based heavily on Bitcoin's BIP system and Ethereum's EIP system. On a biweekly basis, proposals are reviewed by core developers in group calls. Since its launch, over twenty UIPs have been written, of which thirteen have been ratified and built.

4.2 Scaling

In 2016, the Urbit network could barely handle a few hundred online ships. In 2024, at 411 kelvin, the network can easily handle many tens of thousands of online ships, if not more. Packet forwarding is now stateless and does not run Nock. Sponsor pinging now uses the STUN protocol, incurring no disk writes on sponsoring galaxies.

The Directed Messaging project will increase network throughput by roughly a factor of a thousand, fast enough to

saturate a gigabit ethernet link. The Ares project will increase the maximum persistent data storage by a factor of several thousand, up to 64 terabytes. Ares should also improve Nock execution speed dramatically, paving the way toward Nock being able to be compiled to machine code comparable to that emitted by a C compiler.

Heavy use of the scry namespace facilitates scalable data dissemination through the network, following a diffusion model, where intermediate relays can cache arbitrary data, minimizing duplicate packets across the network.

Future work will likely include multithreaded scry requests, shared-memory interprocess communication in the runtime, and more advanced Nock interpreter optimizations.

4.3 Reliability

All through 2017 and 2018, Urbit would crash for the typical user on a roughly weekly basis. As of 2024, that frequency seems to be anecdotally once every few months, usually from running out of disk space due to excessive event log buildup; this has already been addressed on NativePlanet's systems, which perform automatic event log rollover and truncation.

Kernel and app upgrades are much more reliable. The system runs out of memory much less often. The network maintains connectivity much better. Commonly used applications have fewer bugs. Browser page loads are much faster and more reliable. Individual continuity breaches cause fewer downstream bugs in the kernel and applications. Arvo error handling has fewer known flaws. Gall subscriptions underwent a major rearchitecture in 2022 to fix a queue desynchronization bug. The Behn timer module was rewritten multiple times to shake out several different bugs.

The list goes on. While Urbit still has flaws and occasional crashes, its level of robustness is strikingly higher than it was when the whitepaper was written. By early 2025, we expect Urbit to be reliable enough to pass a security audit that includes penetration testing. At that point, custodial cryptographic applications will be viable on the platform.

4.4 Runtime Work

The runtime has absorbed a large percentage of the work that has been done in the system in the last eight years. The naive tree-walking Nock interpreter was replaced by a custom bytecode interpreter. The system was made to recover from out-of-memory errors and automatically reclaim memory. A memory allocator trick called "pointer compression" was used to increase data storage from 2 GB to 8 GB while remaining a 32-bit interpreter. Demand paging was introduced, reducing the cost of running an Urbit ship from over $10 per month to $0.15 per month on some hosting platforms.

The runtime was broken into two processes, allowing I/O and Arvo event processing to run in parallel. The event log was moved out of a custom file into an LMDB database, then broken up into "epochs" allowing for event log rollover and truncation. The I/O drivers in the runtime have continued to be developed, including full rewrites of the HTTP, Ames, and terminal drivers.

The Ares project, close to initial deployment as of writing, includes a clean-slate rewrite of the Nock interpreter, using an all-new allocator, a novel static analysis technique called Subject Knowledge Analysis to achieve substantial Nock execution speed improvements, and a new persistence module called the Persistent Memory Arena, using a copy-on-write B+ tree to keep memory pages synchronized between disk and RAM.

4.5 Control Plane

Since Urbits are now commonly run in one of a few cloud hosting platforms, the Khan I/O driver and vane were added, giving Urbit an explicit external control plane. Khan can manage runtime lifecycle events and send arbitrary commands and queries to a running ship, all over a Unix domain socket. This has facilitated cloud orchestration of large fleets of Urbit ships. NativePlanet's GroundSeg user-facing ship management interface (NativePlanet, 2024) also uses Khan, in a self-hosted setting.

5 Extensibility

A large portion of development efforts since the whitepaper has gone into various improvements in making the system extensible. Mostly this has been in the direction of userspace applications, but some runtime extensibility has also been added.

5.1 Software Distribution

In 2021, a major incision was made into Clay, Gall, Kiln (the userspace agent that manages agent upgrades), and the Arvo kernel to enable application developers to publish bundles of source code that could be installed and run as applications.

Before that, a user would have to run a command-line merge in order to install any software other than the base distribution. This effectively meant forking the source code of your ship, and it commonly caused downstream issues when trying to stay up to date. Within a few months of launching this feature, several Urbit application companies formed to take advantage of the new capability.

This system has evolved some in the ensuing few years, but the basic concepts and components remain the same as the ones introduced in 2021.

5.2 Clay Static Linking and Deduplication

Once users started to have multiple desks with complex applications in them, the Ford build system started to have quite a bit of duplication, where multiple desks would build the exact same library or filetype definition, leading to several copies of common libraries in memory.

Having many copies of the same library in memory is a well-known problem with static linking, which is logically what Ford does to link Hoon programs together. The linking step is a trivial dynamically typed ("vase-mode") 'cons' operation, since Hoon's variable lookup simply traverses the "subject" (environment) type left-to-right. Importing a library means prepending that library to the environment.

Fortunately, because Urbit supports structural sharing and Ford is a purely functional build system, a referentially transparent cache was added to Ford. This cache does not know which desk a build is being run in, so it can cache and deduplicate builds across desks seamlessly without losing the logical static linking (see `~rovnys-ricfer` and `~wicdev-wisryt` (2024), pp. 75–82 in this issue).

Urbit thus gets the best of both worlds: the sanity of static linking and the memory deduplication of dynamic linking.

5.3 Clay Tombstoning

Besides noun deduplication, Clay has also introduced a tombstoning system. Files in all revisions before the current one may be "tombstoned", meaning that the old data has been deleted and just the reference remains. Tombstoning policy can be configured for files, directories, etc, so that large files aren't duplicated while full revision history can be retained for other things where it's appropriate.

Tombstoning is not at odds with the referential transparency of scries, since a scry attempt may return `[~ ~]`, a permanent scry failure (that resource will never be available).

5.4 Userspace Permissions

As users install more applications, it becomes increasingly important for the system to be able to protect itself and other apps from a malicious app. A basic measure was introduced in 2023 so that a Gall agent can tell which other local agent sent it a command. This "poke provenance" was enough to prevent certain kinds of confused deputy attacks involving ships delegating some functionality to their moons.

For full protection, however, a system of "userspace permissions" is required, where the user must approve an agent's access to kernel features and other applications with some granularity. Such a system was prototyped in late 2022 and is scheduled for further development and deployment later in 2024.

6 Conclusion

Urbit development has begun to realize many of the promises of the 2016 whitepaper. The system is more reliable, more scalable, more extensible, and more secure than it was then. The network has grown to include thousands of users, and the system is now capable of supporting a much larger userbase. Indeed, this article omits many quality-of-life improvements, such as scrying over HTTP, EAuth, `%bout` timing hints, bootstrap pill management, reliability and usability of Tlon's social media application, color printing, ordered map data structures, compile-time evaluation in Hoon, the Bridge contract operations tool for Azimuth, and details of the Urbit HD wallet. More detail on past and planned accomplishments are provided at the published roadmap, `https://roadmap.urbit.org/`.

While more work remains to be done, the kernel of Urbit is sound and the contours of the kelvin-cooled system have begun to assume more resolution. ⊠

Appendix

This appendix addresses developments from content in the whitepaper by section. While this is somewhat narratively unsatisfactory, it provides an easy reference. In cases where no significant changes have been made, the section has been omitted.

Accessing Urbit's namespace as a FUSE filesystem still has not been implemented. Some of the whitepaper was written in the "prophetic present tense", and while a lot of those prophecies have come true, that one remains aspirational.

Web 2.0 interoperability (as the whitepaper discusses, Urbit as a "browser for the server side") has been partially superseded by Web3 interoperability. In 2016 Web3 barely existed, but by 2024 it has become a relatively established industry. Over the last few years, acceptance of Urbit's complementary nature to blockchains has broadened.

Web3 is based on the idea of individual sovereignty: open entry and permissionless extensibility. Blockchains strive to

provide these properties for public, global state. For individual sovereignty over personal and private social computing, something like Urbit is needed. Web3 has an Urbit-shaped hole.

Urbit is still "neither secure nor reliable", but it is much more reliable than back then. We estimate that about two more "nines" have been added to its uptime percentage. Error handling has improved dramatically. Userspace apps can be managed by a user. The runtime and kernel both clean up after themselves better in several different ways, so crashes due to resource exhaustion are much less common.

All of the commands that users now lean on for managing resource usage were written after the whitepaper:

- `|trim` cache pruning
- `|pack` defragmentation
- `|meld` deduplication
- `chop` event log truncation
- `roll` event log rollover

Nock's changes going from 5K to 4K resulted from `~fodwyt-ragful`'s bytecode interpreter in 2018. Three changes were made:

1. The Nock 5 equality check opcode was tweaked to enforce taking two subexpressions, rather than optionally taking a single expression that could return a cell (pair), whose head and tail would then be compared for equality. This optionality turned out to be annoying when writing an interpreter, and it provided little utility.

2. Nock version 4 added a new Nock 10 opcode to replace part of a tree with a new value. Creating mutant trees like this is a basic operation that Hoon code performs regularly, using the `%=` rune and its variants (`%*`, `%_`, `=.`, and `=:`). Before the addition of the Nock 10 opcode, the Hoon compiler would use the type of the trees involved in a mutation expression to codegen a tuple of

Nock "auto-cons" expressions (cells of expressions that return cells of their results) that would reconstruct the mutant tree out of the new subtree cons'd together with the remaining pieces of the original tree.

This had a few problems. It allocated more Nock cells at runtime than needed. It prevented the interpreter from optimizing in-place mutation of a noun with a reference count of one. If no one else is looking, creating a mutant copy can be done by overwriting the noun in-place and nobody can tell on you. APL interpreters, which have a similar memory model to Nock nouns, usually include this optimization, and Urbit's current runtime does as well.

3. A tertiary benefit to adding Nock 10 was that the calling convention for "slamming a gate" (providing an argument to an anonymous function and running the function's body) could now be expressed in statically generated Nock code without needing to invoke the Hoon compiler. This allowed a bit of additional expressivity in some Hoon standard library routines that call into untyped Nock.

Parsing and compiling are still slow, unfortunately, but there have been massive performance improvements to Hoon compilation, especially in memory usage. In 2017 compiling kernelspace would use around a gigabyte of RAM; now it uses on the order of a hundred megabytes. Reducing this memory pressure alleviated a lot of crashes from out-of-memory errors (`bail:meme` errors) – it was a major contributor to the increase in overall reliability.

The Udon and Sail domain-specific languages were added to Hoon, for Markdown-like rich text and XML/HTML, respectively. There is a tentative consensus among core developers that these DSLs should be kicked up a layer out of the base Hoon language, but making Hoon syntax recursively extensible likely requires splitting out the definition of the Hoon AST into a recursion scheme, which is nontrivial.

Hoon's recent changes have reduced the prevalence of the

"TMI problem". Since Hoon's type system supports subtyping, it is common for code to fail to compile due to having "too much information" (TMI) – sometimes a more specific type cannot be used in place of a more general one, namely when that type is in "contravariant position". For example, if the type of some field in the Hoon subject was a list but was specialized in a conditional branch to be a non-null list, attempting to "overwrite" (really, create a mutant copy of) that field in the subject with a normal list, the compiler will throw a type error because it cannot guarantee that the normal list is not null. If the compiler did not know the subject's list field was non-null, then overwriting it with some other list would "just work". Almost always, the eventual result of the expression admits a null list, so the type error is an annoyance rather than an assurance.

Hoon type system changes have also included multiple attempts at resolving another difficulty with subtyping: "wetness". A "wet core" is a Hoon core (pair of code and data, where the code is run using the whole core as the subject) with a "sample" (argument slot – the head of the "data" tail by convention in the type system) whose type can vary. This is very closely related to the mathematical idea of parametric polymorphism. Like some other languages (including Haskell), Hoon does not strictly guarantee "parametricity", but wetness is used to implement parametrically polymorphic functions and even type constructors (called "mold builders" in Hoon).

Wetness has an issue where the "faces" (names for particular slots within the tree) in the sample type in the definition of the wet core can differ from the faces in the sample type at any given call site. Deciding which of those faces to use in the body of the code is a subtle, thorny problem. The Hoon at the time of the whitepaper employed one solution; another was written afterward, around 2017; a third was deployed in 2019. This last revision, called "repainting", is generally thought to be the right strategy, but its implementation remains unfinished in 2024 – it has an issue where the precedence of call-site vs. definition-site face names is not preserved under recursion. This issue will need to be fixed before we can say the right answer to this issue has been achieved.

Wetness in general is an interesting and novel approach to

both polymorphic and recursive types, both of which are directly represented as lazily evaluated types (Urbit core developers often call the "evaluation" step "type inference" informally, but it corresponds to the "type synthesis" direction of a bidirectional typechecker in type theory terminology). This has pros and cons.

On one hand, this representation allows for more expressivity than many comparable type systems at around the same spot on Barendregt's lambda cube (λω: types and values can both depend on types). For example, the `$_` rune allows the programmer to declare the type of one expression in terms of the return type of any other expression, without that expression ever needing to be run, only "type-inferred" (synthesized). This is very convenient for programs that return modified versions of themselves, which is ubiquitous in Urbit.

On the other hand, wetness can be difficult to reason about – it is the hardest part of Hoon to understand fully. In addition, it likely slows down the Hoon compiler considerably, since every call-site needs to run a separate full type synthesis step, and in chains of wet calls, that synthesis becomes recursive. It is not difficult to imagine running up against exponential asymptotics in a system like this.

There is some interest among core developers in experimenting with a more constrained type scheme that could be compiled to a fully evaluated data structure with de Bruijn indices. Such a quantified type scheme with binders could make Hoon's type system easier to learn and reason about.

The formal Arvo boot sequence as described in the whitepaper was not implemented at the time it was written. It was deployed in 2019. Around the same time a lot of other runtime changes were made. The runtime was split into an I/O process and an Arvo worker process. The event log was switched from a custom binary file to the LMDB database.

Meanwhile, in userspace, the `%spider` agent was added, to run a new kind of userspace program: a `%thread`. This is not a separate processor thread of execution, but rather a system of monadic I/O based heavily on Haskell's I/O monad. Unlike agents, which are designed to allow a programmer to implement multiple long-lived asynchronous operations concur-

rently using non-blocking I/O, a thread is designed for building a linear sequence of asynchronous operations, more akin to a coroutine.

Arvo's "duct" as a reified vane call stack is still very much in use. The patterns of use have evolved somewhat. It is now much more common for vanes to use "synthetic ducts" as the origins of certain operations to disconnect them from their cause.

For example, Ames uses a synthetic duct for each "flow" (Ames's version of a socket connection) to set packet re-send timers for that flow and pass requests to other vanes. This disconnects the flow's duct from the duct that the runtime used to inject the event into Arvo, preventing Unix process restarts from causing desynchronization. As another example, Gall now uses a per-agent synthetic duct for the moves coming out of that agent.[2]

The `.^` dotket rune no longer blocks until complete. Because the agent model expects non-blocking I/O, a single dotket could make an entire agent stuck if it can't be completed. While a lot of work has gone into fleshing out the scry namespace, core developers are now entertaining the notion that the expectation of synchronous access to the namespace might not be worth it for agents. An Urbit Improvement Proposal has been submitted that would remove the `.^` dotket rune in favor of a move-based interface to the namespace (see `~rovnys-ricfer`, `~fodwyt-ragful`, and `~mastyr-bottec` (2024), pp. 47–58 in this issue).

This shift reflects a subtler attitude shift toward being more wary about asynchronicity. Earlier Urbit designs were bolder about introducing asynchronous operations; more modern ones treat asynchronous operations as complex and bug-prone, to be avoided if possible. This shift has come from experience – it is surprising to find race conditions in a single-threaded, purely functional operating system, but we have found and fixed plenty over the years.

[2]Note that this approach currently has a known flaw, wherein if an agent sets a timer in response to hearing a request from another ship, it needs to remember that ship in order to obtain the same duct to cancel the timer, cf. issue #6884.

The whitepaper states that "most data is now in Clay". This is no longer true. Now most data lives in Gall as agents and their state. Clay has proven difficult to use for storing application data for a number of reasons, and Gall has become more featureful. There is consensus among the core developers that a better userspace model should be devised to unify Clay and Gall. One such model, called "shrubbery", is now being investigated.

Urbit is still a non-preemptive OS, but it could be made preemptive by letting the kernel emit a Nock hint to the runtime to enforce a time limit on the enclosed computation. If it failed to complete within the given time limit, the runtime would abort the entire Arvo event. This would be considered a nondeterministic error (`bail:fail`), like a user-initiated event interruption (Ctrl-C). This could potentially even be used to turn Urbit into a real-time OS.

3.1 Uniform Persistence

Significant work has gone into improving the runtime's implementation of uniform persistence:

- demand paging to allow storage of more data on disk than fits in RAM;
- a guard page to efficiently prevent stack smashing;
- a snapshot migration framework to facilitate upgrades;
- separating ephemeral and persistent state to avoid extraneous disk writes;
- using the LMDB database for the event log, instead of a custom binary file;
- the Ares project's Persistent Memory Arena (PMA), a general-purpose single-level store that can efficiently manage the persistence and paging for 64 terabytes of data, using a copy-on-write B+ tree;
- the "epoch system", which broke the event log into epochs, allowing event log rollover and truncation;

- persistent Nock caches, allowing arbitrary computations to be memoized in the runtime in a way that persists across Arvo events and Unix process restarts.

3.2 Source-Independent Networking Ames is still sequential and idempotent for commands. Despite having been described as "content-centric" and "source-independent" in the whitepaper, those two properties are only now being realized, in the "directed messaging" project that is currently underway. Directed messaging refactors Ames to be much more like the Named Data Networking project (Zhang et al., 2014): packets and messages are both constrained to be request/response, with routing being achieved using a Pending Interest Table just like in NDN.

Directed messaging supports network reads in the form of namespace reads. A ship can send a network request to read from another ship's namespace. The requesting ship sends request packets containing a scry path, and the publishing ship responds with an authenticated scry binding: a pair of the path and the data bound to that path.

Network writes (commands) are factored in terms of reads: a write request is sent as a read request for an acknowledgment that the command has been performed by the receiving ship. When the receiving ship hears this request, it sends a network read request to download the command datum from the sending ship, who has published it in its scry namespace. Once the command finishes downloading, the receiving ship performs the command and makes the acknowledgment available in its own namespace.

As an optimization, the requesting ship places the first fragment of the command datum in the first request packet it sends. In the common case of a 1 KB command datum, the entire command+ack interaction consists of a single packet roundtrip.

This is only the latest in a series of changes that have been made to Ames since the whitepaper:

- it is now backed by Azimuth, on the blockchain
- it supports personal continuity breaches

- "stateless forwarding" brought orders of magnitude more scalability
- the `%clog` backpressure system was added to prevent unbounded packet queueing on publisher ships; this was later tuned for better performance
- flows can now be closed, so their memory resources can be reclaimed
- "dead flow consolidation": outbound requests to unresponsive ships now share one global timer every two minutes, dramatically reducing the number of Arvo events
- the `%flub` system was added to close a DoS vulnerability: the system can drop incoming requests until it is ready to process them, without any queueing
- the `|snub` and `|ruin` commands were added to allow manual blacklisting of peers
- network reads have been implemented
  - the Fine remote scry protocol
  - the `%chum` one-to-one encrypted remote scry protocol
  - the one-to-many encrypted remote scry protocol

3.4 Interruption, etc. There are still no worker threads, but a potential strategy has been devised for them, and the Ares PMA should support parallel execution. There are still no event timeouts, but we do not expect them to be difficult to implement.

3.5 Solid-State Interpreter While there is not yet a unikernel, a specification has been written for one. There is no Raft cluster but the `%phoenix` app now uses encrypted remote scry to back up application state. There have also been proposals to use FoundationDB to replicate event log storage.

4.2 Larval State This section is still applicable. Arvo still has a larval stage, and it does more or less the same thing.

The "bring your own boot system" work, which has been built but not deployed, will allow a user to assemble a boot sequence out of kernel source and a list of userspace desks (app source packages). This could also be used to facilitate "quick-boot". Nock caches could be injected into the runtime along with kernel or app source. These caches could contain all invocations of the Hoon parser and compiler, reducing boot time to well under a second on common machines.

5.1 Nouns The Murmur3 system was aspirational but has been subsequently realized. A few years ago, a horrendous bug was found and fixed in the "mug" short hash for cells: the algorithm was symmetric, so `[1 0]` hashed to the same value as `[0 1]`. This led to far more hash collisions than there should have been, degrading the asymptotics of a number of data structures.

6.3 Nock Optimization The "jet dashboard" (the module in the runtime responsible for registering, matching, and running jets) has been rewritten multiple times. `~fodwyt-ragful` has proposed a jet dashboard that would match Nock formulas on the basis of noun equality rather than matching Nock cores (a code/data pair used as the unit of code organization) while ignoring mutable slots, the way the previous ones have worked. This would have the advantage of simplicity, but it could be more difficult to implement more complex pieces of code, such as closures.

The Ares project has another new approach to jet matching, which is to resolve the matching statically during a transpilation step from Nock to a more optimized representation. This is facilitated by a novel static analysis technique for Nock, called Subject Knowledge Analysis (SKA), the brainchild of `~ritpub-sipsyl`. SKA analyzes the data dependencies within a Nock formula, allowing inference about the tree shape of the "subject" (environment, or scope) at different points of the computation. SKA should allow for more efficient registerization of intermediate values, and will also provide enough information

to statically compute whether a jet will match. The goal is for Ares to convert a hot loop in Nock into similar machine code that would be emitted by a C compiler, including direct jumps back to the start of the loop.

The Rust interpreter project mentioned in the whitepaper is dead, but in the meantime several other interpreters have been written:

1. Jaque, written using the Graal/Truffle JIT framework;
2. NockJS, an interpreter in JavaScript that uses trampolining;
3. `cl-urbit`, a Common Lisp interpreter used as a jet dashboard testbed; and
4. Ares, written in Rust and C, intended as the next mainline interpreter version.

In 2019, a custom Nock bytecode was written and deployed. This was so much faster than the old tree-walking interpreter that we were able to delete tens of thousands of lines of C code that jetted the Hoon compiler. Until then, the Hoon compiler was almost entirely jetted, since the Nock interpretation overhead was so bad. Since then, the allocator and Nock function calls have been slow, but the speed of Nock noun manipulation has been comparable to manual manipulation in C.

An interesting optimization that was added to Vere more recently is called "tail-call modulo cons" (Leijen and Lorenzen, 2023). This approach allows recursive loops that are almost tail-recursive to be optimized as if they are tail-recursive, as long as the only operation that happens after each recursive call is a "cons" that prepends or appends another value to the result of the recursion.

Another optimization, introduced with the bytecode interpreter and refined recently, was to optimize the Nock 10 opcode that overwrites a slot in a tree. In the case that there is only one reference to the tree being modified, the mutation can be performed in-place, by overwriting memory, rather than by allocating a new mutant tree and then decrementing the refcount on the old one.

6.5 Transition Function The whitepaper mentions off-handedly that Nock could not be replaced. While this author does not expect Nock to be changed, much less replaced, this could be done if needed. Such implementation is left as an exercise for the reader.

7.1 Hoon Semantics Hoon has grown, but the semantics have largely remained intact. `$span` and `$twig` have been renamed, but this may be reverted at some point. (Most core developers prefer the older names.) Core types have "chapters" added to them, mostly for documentation purposes.

Head recursion still requires a cast in Hoon. The compiler can go into an infinite loop if given malformed input. Interestingly, of all the challenges experienced by Hoon developers, this has turned out not to be a severe issue in practice. (Future language designers take note: needing to hit Ctrl-C if your compilation takes a lot longer than normal is not a show stopper.)

Hoon keywords, employed in the whitepaper appendices, was removed due to disuse and disinterest. Keyword Hoon may be re-introduced for pedagogical purposes, but for all essential purposes, runes are the most expressive and intuitive way to think about Hoon as a mature developer.

7.2 Type Definition All of the whitepaper's high-level claims about Hoon are still true, but a few changes to mold (type) declaration and the scope of the language's keyword equivalent (runes) has been expanded.[3] There have been two or three revisions of syntax and type system. The latest change added "doccords", a form of docstring, to the syntax.

The syntax for declaring molds has changed, and as of the last major revision to the mold system in 2019, it is now possible to add a predicate gate to a mold definition that will be run whenever the derived mold function ("gate") is run to coerce a raw noun into a noun of that type. This is useful for ingesting data that has constraints other than tree shape, such as the

[3] We consider the growth of Hoon to be somewhat unfortunate, and envision that it eventually slims back down somewhat as it is refined.

ordering of keys in a map.

A number of runes have been added to Hoon:

1. `|$` barbuc (produce mold builder wet gate)
2. `|@` barpat (produce wet core)
3. `$|` bucbar (produce structure to satisfy validator)
4. `$<` bucgal (filter a mold, exclude)
5. `$>` bucgar (filter a mold, match)
6. `$:` bucmic (enter manual value mode)
7. `$~` bucsig (define default custom type value)
8. `#/` haxfas (produce typed path)
9. `+$` lusbuc (type constructor arm)
10. `+*` lustar (door alias)
11. `;<` micgal (monadic do notation)
12. `^~` tissig (compose many expressions)
13. `?#` wuthax (match type, experimental)
14. `!<` zapgal (extract vase to mold)

The canonical pronunciations for syllables was changed in 2019, then reverted due to effectively unanimous demand. The one modification that persisted was the change of `;` from 'sem' to 'mic', since 'sem' sounded too close to 'cen' (`%`).

The Hoon type system used to use an idempotence check when coercing a raw noun into a typed noun. As of the 2019 type system revision, that check has been removed, and the mold gate will crash (`!!` zapzap) if the noun fails to validate.

8.1 Arvo Kernel Interface Arvo's `+peek` arm is employed much more than it used to be, since we make heavier use of the namespace, and because as Urbit matures, the runtime needs to do more inspection of the Arvo state.

The "wires" (request identifier paths) that Vere uses were cleaned up to share the same conventions and match the name of the target vane.

8.2 Arvo Kernel Modules Vane routing has long used only the first letter of the vane; this is now specialized to the standard vanes, and new vanes can be added that have an arbitrary number of letters, although four-letter vane names remain standard. This allows for an arbitrary number of vanes, not 26 maximum.

The vanes have changed quite a bit since the whitepaper was written. No vane has remained untouched.

1. Ames (p2p networking) was rewritten twice.

2. Behn (timers) was added, rewritten, and then heavily modified.

3. Clay (revision control) has undergone several major transformations (discussed elsewhere throughout this article).

4. Dill (terminal interaction) has been rewritten.

5. Eyre (HTTP) was rewritten and its HTTP client functionality was split into a new vane, Iris, leaving Eyre as just the server side.

6. Ford (build system for Hoon) was written, rewritten, and finally moved into Clay.

7. Gall (application runner) has been rewritten and has had many major changes made to it.

8. Jael (PKI) was added for secret management, rewritten to track the PKI, then modified to work with the PKI rollup.

9. Khan (control plane) was added to allow scripting and lifecycle management from Unix.

10. Lick (Unix inter-process communication) was added recently, so Urbit can communicate using nouns with other local Unix processes.

8.3 Structured Events Depth-first move ordering is being revised, as described in more detail in an earlier section.

The whitepaper mentions it is "not conventional" in Urbit to use promises. `%spider` threads are now a major exception to this. Threads are all killed on upgrade to prevent leaking the old kernel inside the closures.

8.4 Security Mask "Arvo is single-homed" per the whitepaper. This property was subsequently changed to multi-homed to support onboarding as a comet and upgrading to a planet, then removed again when Azimuth was put on-chain.

The security mask is just now starting to be enforced by the vanes. This was necessitated by the `%chum` one-to-one encrypted remote scry protocol. Agent provenance was added to pokes, so an agent now knows which local agent poked it.

This represents a shift in thinking among core developers. Rather than a more monolithic, static, trusted userspace in earlier designs, userspace is now thought of as more multicentric, dynamic, and untrusted. If apps can all be trusted, there is no need to protect one app against the others. In practice, however, there is a compelling need to prevent a single malicious (or, equivalently, badly written) app from breaking the security properties of any other app. Allowing an app to see which other local app is poking it provides enough information to resolve a major class of confused deputy problems without implementing a full-fledged capabilities system.

The security mask model is not expressive enough to encode the idea of a private group with a dynamically calculated set of participants. This was needed for the one-to-many encrypted remote scry protocol, since otherwise an app would need to fully materialize the set of allowed peers whenever it published a piece of data. The computational cost of this would

have superlinear asymptotics, which was deemed unacceptable at the base layer.

Instead, the encrypted remote system has its own separate concept of a "security domain" as a path prefix, and the kernel performs a dynamic query when a peer asks for access to private data with that prefix. While the performance is not a direct comparison with the security mask solution, the dynamic query can typically be implemented in logarithmic time by the application, e. g. a binary search through a Hoon `$map` data structure (a treap). For more complex permissions models, this approach should scale much better.

9.2 Cryptosuite The cryptosuite is still not upgradeable through the initially intended mechanism, where an interface core would be sent to Ames and stored there until a kernel update, when it would need to be deleted and re-sent to avoid leaking the old kernel that would be in the closure of the crypto interface core. The "new" `crub` cryptosuite core has been in use for several years now, though, with standard cryptographic algorithms jetted by reference implementations. The Ames cryptosuite currently in use was audited by a security firm in 2020 (`~rovyns-ricfer` and `~poldec-tonteg`, 2020).

In practice, the Ames protocol has changed a few times since the whitepaper, but the cryptosuite has not. Allowing the cryptosuite to continue to upgrade after kelvin zero will probably still need to be implemented, but we have not yet done so.

For simplicity, when going on chain, the Ames protocol key handshake was changed to a stateless Diffie-Hellman key exchange between the public keys listed on chain. This has the advantage that two ships can start sending encrypted, authenticated packets to each other without any handshake at all. The downside is that forward secrecy is limited to on-chain key updates, which have a cost, so in practice there is very little forward secrecy. This will likely be revised in a later protocol version to support better forward secrecy.

With the Directed Messaging rewrite, a new packet authentication protocol has been added, called LockStep. This

protocol is based on the BAO streaming authentication protocol (O'Connor, 2024). Like BAO, it uses the Blake3 tree hash to establish fingerprints for both individual packets and the full message datum. It differs from BAO in two main ways: it is packet-oriented rather than byte-oriented, and it guarantees that as long as packets are received in order, each packet can be authenticated immediately without waiting for any other packets.

The encrypted remote scry protocol also uses the `+crub` cryptosuite with deterministic encryption, but it does have forward secrecy, tunable by each application.

9.3 PKI Azimuth, the set of Ethereum contracts, replaced the whitepaper's original PKI. About two years later, the "Layer 2" naive rollup was added to reduce the cost of spawning planets.

"Wills" and "deeds", as described in this section, are no longer part of the system. 8-bit galaxies, 16-bit stars, and 32-bit planets all have their public keys listed on Ethereum. A 64-bit moon is assigned its public key by the planet that spawned and owns it, and other ships find the moon's public key by asking the planet. A 128-bit comet signs its address with its own key, and other ships validate the comet's key by checking that it hashes to the comet's address. This prevents comets from breaching, but it allows them to self-authenticate without being part of the on-chain PKI.

9.4 Update, Routing, and Parental Trust Urbit's federated routing system can still be analogized to a combination of STUN and TURN approaches to NAT traversal.

The STUN protocol is now used explicitly: a sponsee ship sends a STUN request to its sponsor every twenty-five seconds. It only sends an Ames command to its sponsor to tell the sponsor it has moved to a new transport address (IPv4 address and port) when the STUN response changes, indicating that from the perspective of the sponsor, the sponsee is at a new transport address. A caveat "from the perspective of the sponsor" is due to the policies of some routers to expose a different source address for each outgoing destination ("source NAT") –

if the sponsee is behind a source NAT, each ship on the network will see a different source address on packets coming from the sponsee.

With directed messaging, routing is now "directed": if ship A sends a request to ship C through C's sponsor B, then the request takes the path $A \rightarrow B \rightarrow C$ through the network, and the response takes the reversed path $C \rightarrow B \rightarrow A$. This is in contrast to all previous versions of Ames, in which the response path will ignore any relays in the request's path and instead get relayed through A's sponsor, in so-called "criss-cross routing".

This directedness promises to reduce routing complexity and handle more NAT configurations, including source NAT. It has already proven to reduce complexity in several other layers of the networking stack.

As for parental trust, the trend has been toward minimizing the amount a parent needs to be trusted. Code signing could be done by a consortium of galaxies or other nodes designated with such a task, such as a set of core developer ships. Urbit's hierarchical deterministic wallet system could require a signature by an upstream key – associated with an Urbit ID but kept "cold" by not being used for any other purpose – to authenticate sensitive updates, such as kernel updates. That way, even if a vulnerability is found that compromises a planet's sponsoring galaxy or star, the planet would validate the signatures by the upstream keys, so that attack on its own would not be enough to load malicious source code onto the planet.

There is support at both the CLI and in Urbit's standard user interface, Tlon's "landscape" app, for turning kernel updates on and off and switching the download source of kernel source code to any ship the user likes.

Userspace apps can be downloaded from any ship on the network, not just the ship's sponsor. This is another step away from obtaining all software from the parent. Userspace apps could use the same kind of code signing mechanism described above – which to be clear, is still vaporware as of the time of this writing.

9.5 Deed Distribution This section is now obsolete except for comets, which as described earlier, are self-signed. It is worth noting that the double-spend problem inherent in the whitepaper's PKI design has now been solved, through using a blockchain.

9.6 Permanent Networking Urbit's networking is still permanent, but edging toward more flexibility and recoverability.

Ames flows (Ames's version of socket connections) can now be killed, allowing both participating ships to reclaim memory resources used for storing the sequence numbers and other metadata about that flow. This is a good example of Urbit doing more to clean up after itself than it did when the whitepaper was written.

The whitepaper reads "brain damage is fatal by default", and "event backup makes no sense", but these claims are no longer considered true. Various forms of recoverability have been added. Personal breaches allow a ship to reset its communications with other ships after losing information. The `%phoenix` app lets app state be backed up and restored – if the restoration occurs after a breach, it is heuristic and could be lossy, but apps can be written defensively to maintain as much state as possible across a breach. Future work will likely push harder on recoverability.

Moving more network activity from writes to reads, through the scry namespace, has been shifting our conception of exactly-once delivery. It now seems likely that by a year or two from now, Urbit's traditional notion of exactly-once delivery will be opt-in, built on top of the more general substrate of publishing scry bindings.

This requires another vaporware warning, but one could imagine a userspace application sending three poke commands to `~zod` by binding the commands to three paths like `/poke/~zod/foo/1`, `/poke/~zod/foo/2`, and `/poke/~zod/foo/3`, into a subtree of its own scry namespace. The kernel would ensure `~zod` is notified of those bindings. `~zod` would download the values at those paths and notify its `%foo` application. The app would interpret the values as a se-

quence of poke commands, to be applied once and in order. Once it processes each poke, it would bind an acknowledgment value to a predetermined path in its own namespace, such as `/ack/~nec/1`.

9.7 Permanent Applications Applications are generally no longer considered permanent. The current Gall model lets a user suspend an application and revive it later. Unfortunately the current model is known to be unsound: "signs", i. e. responses to outgoing commands or queries, to a Gall agent are dropped while the agent is suspended, leading to an inconsistent state.

Future designs for Urbit userspace will likely include a suspension notification for agents. The kernel also needs to intercept and store all pending I/O in and out of agents, so it can cancel the I/O and let the agent know it was canceled.

This goal is at least somewhat at odds with exactly-once network messaging, since in order for the invariants of exactly-once messaging to hold, the sending agent needs to be delivered the acknowledgment reliably. If the ack is dropped while the agent is suspended, that could lead to an inconsistent state between the requesting and receiving ships.

Something namespace-oriented, like the example with poking `~zod` by binding paths in our local namespace, could be more amenable to maintaining exactly-once delivery across suspension and revival without the kernel needing to maintain an unbounded queue of acks to deliver to the agent. It could also help with recoverability after a breach, since storing userspace data with absolute references (i. e. scry paths) allows for more seamless social recovery – if any peer still has the scry binding, the breached ship could download and reinstate it. The importance of immutable absolute references for data is one of the key insights leading to the "shrubbery" noun-maximalist way of thinking.

One way in which applications are more persistent now than at the time of the whitepaper is that upgrades are now quite real. The kernel and userspace apps can both upgrade "over the air" (OTA) by receiving new source code over the Ur-

bit network. The kernel's upgrade system was too buggy for production use until 2020, when the Ford Fusion project moved Ford into Clay – ridding Ford of all its buggy asynchronicities – and enforced a bottom-up upgrade order: Hoon, Arvo, vanes, then userspace.

A year later, near the end of 2021, the ability for Urbit to distribute and install userspace apps was added, enabled by major internal changes to Gall, Clay, and the Arvo kernel to ensure each Clay desk (unit of source distribution) is independently buildable, bootstrapped from its own source code without any external dependencies.

App publishers can push down source code to their users for the next version of their app, intended to be built and run using the next kelvin version of the kernel. When the user's ship receives the kernel's source code for the new kelvin, it upgrades the apps and the kernel atomically with each other, ensuring all apps (soon: only apps marked as "essential") update succesfully – if an app fails to upgrade to the new kelvin version, the Arvo event is aborted, rolling back the kernel upgrade and any other app upgrades that had happened during the event.

Eventually, the kernel should be able to keep agents designed for older kernels running. We call this idea a "kelvin shim". A project along these lines was attempted at the start of 2023, but the current userspace model turns out to be to be too tightly coupled to the kernel for shimming to be feasible. This experience has informed our next generation of ideas for the kernelspace/userspace interface.

9.8 Acknowledgment Acknowledgments have changed in a number of ways. Packet-level acks are being (mostly) removed. Authentication for both packets and messages is being redone in the Directed Messaging project. Negative acknowledgments are no longer a space leak and are cleaned up without loss of correctness after a certain number of later messages. Finally, congestion control is being moved into the runtime.

Some things have stayed the same. A message is still an application-level transaction. Ignoring the computation time

to process a message is still a useful optimization for congestion control. Negative acknowledgments are still considered to be string-like data describing an error, rather than a programmatically manipulable exception data structure.

9.9 Packet Rejection Cryptography failures (failure to decrypt or authenticate a packet) still cause a silent packet drop. To streamline the implementation of this, the `bail:evil` error tag was added to the system so the runtime can know not to inject an error notification event into Arvo in the case of cryptographic failure.

The `%flub` move is a new reason to drop a packet. When Ames passes a message to a local vane, the vane can now respond with a `%flub` move, indicating it is not yet ready to process the message. If a message sent to a Gall agent that is not running, for example, then Gall will respond with a `%flub`. The sending ship will keep retrying, slowly, on a timer, until the message goes through, but the receiving ship does not have to update any state in response – previous versions of the system required the receiving ship to maintain an unbounded queue of incoming pokes to agents that were not running, which was a denial-of-service vector.

10 Ames Part 2: Bitstream The "jam" serialization format for nouns remains unchanged. Its implementation in Vere is now much more flexible and optimized. There are now also implementations in Rust, JavaScript, Haskell, and Common Lisp.

There are more use cases for jammed nouns now. Portable Arvo snapshots are jammed nouns. Interprocess communication between the two Vere processes uses jammed nouns for its messages. The Khan and Lick vanes also use jammed nouns for IPC, and Eyre, the HTTP vane, now supports HTTP subscriptions using jammed nouns in addition to JSON.

10.4 Packet Structure Modern Ames packets are still jammed nouns with a 32-bit header but other than that almost everything is different. For one thing, their internal fields are now

byte-aligned, making them more human-readable in Wireshark or other Terran packet inspection programs.

A bigger shift is that as of Directed Messaging, packets are actually source-independent for the first time.

10.5 Message Decoding This has been totally rewritten multiple times. There is no more unencrypted first packet. Routing will now be directed.

10.7 Poke Processing Pokes are still typed and transactional. Mark lookup is now performed on a per-desk basis, rather than globally, since each app can define its own marks.

Deferred acknowledgments remain vaporware.

11 System Status and Roadmap The whitepaper's 30,000 source lines of code has expanded to roughly 40,000. This number should go down a bit over time. Other aspects of reliability and performance have been extensively discussed above.

References

Barendregt, Henk (1991). "Introduction to generalized type systems." In: *Journal of Functional Programming* 1.2, pp. 125–154. DOI: 10.1017/s0956796800020025.

Leijen, Daan and Anton Lorenzen (2023). "Tail Recursion Modulo Context: An Equational Approach." In: *Proceedings of the ACM on Programming Languages* 7.POPL 40, pp. 1–30. DOI: 10.1145/3571233.

~midden-fabler, Scott Wilson (2024) "%phoenix". URL: https://github.com/urbit/phoenix (visited on ~2024.4.24).

NativePlanet (2024) "GroundSeg". URL: https://github.com/Native-Planet/GroundSeg (visited on ~2024.4.8).

O'Connor, Jack (2024) "Bao". URL: https://github.com/oconnor663/bao (visited on ~2024.4.8).

~ravmel-ropdyl, Galen Wolfe-Pauly (2019) "Azimuth is On Chain". URL: https://urbit.org/blog/azimuth-is-on-chain (visited on ~2024.4.24).

~rovnys-ricfer, Ted Blackman, Paul Driver ~fodwyt-ragful, and Matthew Levan ~mastyr-bottec (2024). "The Stakes of Scrying: .^ Dotket and the Seer Proposal." In: *Urbit Systems Technical Journal* 1.1, pp. 47–58.

~rovnys-ricfer, Ted Blackman and Philip C. Monk ~wicdev-wisryt (2024). "A Solution to Static vs. Dynamic Linking." In: *Urbit Systems Technical Journal* 1.1, pp. 75–82.

~rovyns-ricfer, Ted Blackman and Anthony Arroyo ~poldec-tonteg (2020) "Ames Security Audit and the Future of the Protocol". URL: https://urbit.org/blog/security-audit (visited on ~2024.4.8).

~sorreg-namtyv, Curtis Yarvin (2006) "U, a small model". URL: http://lambda-the-ultimate.org/node/1269 (visited on ~2024.2.20).

— (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).

Zhang, Lixia et al. (2014). "Named Data Networking." In: *ACM SIGCOMM Computer Communication Review* 44.3, pp. 66–73. DOI: 10.1145/2656877.2656887.