(This is Urbit whitepaper DRAFT 41K. Some small details remain at variance with the codebase.)

Urbit: an operating function

Abstract

Urbit is a clean-slate, full-stack redesign of system software. In 25K lines of code, it's a packet protocol, a pure functional language, a deterministic OS, an ACID database, a versioned filesystem, a web server and a global PKI. Urbit runs on a frozen combinator interpreter specified in 200 words; the rest of the stack upgrades itself over its own network.

Architecturally, Urbit is an opaque computing and communication layer above Unix and the Internet. To the user, it's a new decentralized network where you own and control your own general-purpose personal server, or "planet." A planet is not a new way to host your old apps; it's a different experience.

Objective

How can we put users back in control of their own computing?

Most people still have a general-purpose home computer, but it's atrophying into a client. Their critical data is all in the cloud. Technically, of course, that's ideal. Data centers are pretty good at being data centers.

But in the cloud, all users have is a herd of special-purpose appliances, not one of which is a general-purpose computer. Do users want their own general-purpose personal cloud computer? If so, why don't they have one now? How might we change this?

Conventional cloud computing, the way the cloud works now, is "1:n". One application, hosted on one logical server by its own developer, serves "n" users. Each account on each application is one "personal appliance" - a special-purpose computer, completely under the developer's control.

Personal cloud computing, the way we wish the cloud worked, is "n:1": each user has one logical server, which runs "n" independent applications. This general-purpose computer is a "personal server," completely under the user's control.

"Personal server" is a phrase only a marketing department could love. We prefer to say: your *planet*. Your planet is your digital identity, your network address, your filesystem and your application server. Every byte on it is yours; every instruction it runs is under your control.

Most people should park their planets in the cloud, because the cloud works better. But a planet is not a planet unless it's independent. A host without contractual guarantees of absolute privacy and unconditional migration is not a host, but a trap. The paranoid and those with global adversaries should migrate to their own closets while home computing remains legal.

But wait: isn't "planet" just a fancy word for a self-hosted server? Who wants to self-host? Why would anyone want to be their own sysadmin?

Managing your own computing is a cost, not a benefit. Your planet should be as easy as possible to manage. (It certainly should be easier than herding your "n" personal appliances.) But the benefit of "n:1" is *controlling* your own computing.

The "n:1" cloud is not a better way to implement your existing user experience. It's a different relationship between human and computer. An owner is not just another customer. A single-family home is not just another motel room. Control matters. A lot.

Perhaps this seems abstract. It's hard to imagine the "n:1" world before it exists. Let's try a thought-experiment: adding impossible levels of ownership to today's "1:n" ecosystem.

Take the web apps you use today. Imagine you trust them completely. Imagine any app can use any data from any other app, just because both accounts are you. Imagine you can "sidegrade" any app by moving its data safely to a compatible competitor. Imagine all your data is organized into a personal namespace, and you can compute your own functions on that namespace.

In this world, no app developer has any way to hold your data hostage. Forget how this works technically (it doesn't). How does it change the apps?

Actually, the rules of this thought-experiment world are so different that *few of the same apps exist*. Other people's apps are fundamentally different from your own apps. They're not "yours" because you developed them -- they're your apps because you can fire the developer without any pain point. You are not a hostage, so the power dynamic changes. Which changes the app.

For example: with other people's apps, when you want to shop on the Internets, you point your browser at amazon.com or use the Google bar as a full-text store. With your own apps, you're more likely to point your browser at your own shopping assistant. This program, which *works entirely for you and is not slipping anyone else a cut*, uses APIs to sync inventory data and send purchase orders.

Could you write this web app today? Sure. It would be a store. The difference between apps you control and apps you don't is the difference between a shopping assistant and a store. It would be absurd if a shopping assistant paid its developer a percentage of all transactions. It would be absurd if a store didn't. The general task is the same, but every detail is different.

Ultimately, the planet is a different user experience because you trust the computer more. A program running on someone else's computer can promise it's working only for you. This promise is generally false and you can't enforce it. When a program on your computer makes the same promise, it's generally true and you can enforce it. Control changes the solution because control produces trust and trust changes the problem.

Could we actually add this level of user sovereignty to the "1:n" cloud? It'd take a lot of engineers, a lot of lawyers, and a whole lot of standards conferences. Or we could just build ourselves some planets.

Obstacles

If this is such a great product, why can't you already buy it?

In 2015, general-purpose cloud servers are easily available. But they are industrial tools, not personal computers. Most users (consumer and enterprise) use personal appliances. They choose "1:n" over "n:1". Why does "1:n" beat "n:1" in the real world?

Perhaps "1:n" is just better. Your herd of developer-hosted appliances is just a better product than a self-hosted planet. Or at least, this is what the market seems to be telling us.

Actually, the market is telling us something more specific. It's saying: the appliance herd is a better product than a self-hosted *Unix server on the Internet*.

The market hasn't invalidated the abstract idea of the planet. It's invalidated the concrete product of the planet *we can actually build on the system software we actually have*.

In 1978, a computer was a VAX. A VAX cost $50K and was the size of a fridge. By 1988, it would cost $5K and fit on your desk. But if a computer is a VAX, however small or cheap, there is no such thing as a PC. And if a planet is an AWS box, there is no such thing as a planet.

The system software stack that 2015 inherited -- two '70s designs, Unix and the Internet -- remains a viable platform for "1:n" industrial servers. Maybe it's not a viable platform for "n:1" personal servers? Just as VAX/VMS was not a viable operating system for the PC?

But what would a viable platform for a personal server look like? What exactly is this stack? If it's not a Unix server on the Internet, it's an X server on network Y. What are X and Y? Do they exist?

Clearly not. So all we have to replace is Unix and the Internet. In other words, all we have to replace is everything. Is this an obstacle, or an opportunity?

A clean-slate redesign seems like the obvious path to the levels of simplicity we'll need in a viable planet. Moreover, it's actually easier to redesign Unix and the Internet than Unix *or* the Internet. Computing and communication are not separate concerns; if we design the network and OS as one system, we avoid all kinds of duplications and impedance mismatches.

And if not now, when? Will there ever be a clean-slate redesign of '70s system software? A significant problem demands an ambitious solution. Intuitively, clean-slate computing and the personal cloud feel like a fit. Let's see if we can make the details work. We may never get another chance to do it right.

Principles of platform reform

To review: we don't have planets because our antique system software stack, Unix and the Internet, is a lousy platform on which to build a planet. If we care about the product, we need to start with a new platform.

How can we replace Unix and the Internet? We can't. But we can tile over them, like cheap bathroom contractors. We can use the Unix/Internet, or "classical layer," as an implementation substrate for our new layer. A platform on top of a platform. Under your Brazilian slate, there's pink Color Tile from 1976.

"Tiling over" is the normal way we replace system software. The Internet originally ran over phone circuits; under your transatlantic TCP packets, there's an ATM switch from 1996. And of course, under your browser there's an OS from 1976.

After a good job of tiling over, the obsolete layer can sometimes be removed. In some cases it's useless, but would cost too much to rip out. In some cases it's still used -- a Mac doesn't (yet) boot to Safari. But arguably, that's because the browser platform is anything but a perfect tiling job.

One property the browser got right was total opacity. The old platform implements the new platform, but can't be visible through it. If web apps could make Unix system calls or use Unix libraries, there would be no such thing as web apps.

(In fact, one easy way to think of a planet is as "the browser for the server side." The browser is one universal client that hosts "n" independent client applications; the planet is one universal server that hosts "n" independent server applications.)

And the bathroom remains a bathroom. The new platform does the same *general* job as the old one. So to justify the reform, it has to be *much* better at its new, specific job. For instance, the browser is easily two orders of magnitude better than Unix at installing untrusted transient applications (ie, web pages).

Abstract targets

Again, we have two problems: Unix and the Internet. What about each do we need to fix? What exactly is wrong with the classical layer? What qualities should our replacement have?

Since we're doing a clean-slate design, it's a mistake to focus too much on fixing the flaws of the old platform. The correct question is: what is the right way to build a system software stack? Besides cosmetic details like character sets, this exercise should yield the same results on Mars as Earth.

But we come from Earth and will probably find ourselves making normal Earth mistakes. So we can at least express our abstract design goals in normal Earth terms.

A simpler OS

One common reaction to the personal-server proposition: "my mother is not a Linux system administrator." Neither is mine. She does own an iPhone, however. Which is also a general-purpose computer. A usability target: a planet should be as easy to manage as an iPhone.

(To go into way too much detail: on a planet with the usability of iOS, the user as administrator has four system configuration tasks for the casual or newbie user: (a) deciding which applications to run; (b) checking resource usage; (c) configuring API authentication to your existing Web apps (your planet wants to access, rip or even sync your silo data); and (d) maintaining a reputation database (your friends, enemies, etc). (a) and (b) exist even on iOS; (c) and (d) are inherent in any planet; (d) is common on the Web, and (c) not that rare.)

Could a planet just run iOS? The complexity of managing a computer approximates the complexity of its state. iOS is a client OS. Its apps are not much more than webpages. There is much more state on a server; it is much more valuable, and much more intricate, and much more durable.

(Moreover, an iOS app is a black box. It's running on your physical hardware, but you don't have much more control over it than if it was remote. iOS apps are not designed to share data with each other or with user-level computing tools; there is no visible filesystem, shell, etc. This is not quite the ownership experience -- it's easy to get locked in to a black box.)

And while an Apple product is a good benchmark for any usability goal, it's the exception that proves the rule for architecture. iOS is Unix, after all -- Unix with a billion-dollar makeover. Unix is not a turd, and Cupertino could probably polish it even if it was.

Simplicity is the only viable path to usability for a new platform. It's not sufficient, but it is necessary. The computer feels simple to the user not because it's presenting an illusion of simplicity, but because it really is simple.

While there is no precise way to measure simplicity, a good proxy is lines of code -- or, to be slightly more sophisticated, compressed code size. Technical simplicity is not actually usability, just a force multiplier in the fight for usability. But usability proper can only be assessed once the UI is complete, and the UI is the top layer by definition. So we assume this equation and target simplicity.

A sane network

When we look at the reasons we can't have a nice planet, Unix is a small part of the problem. The main problem is the Internet.

There's a reason what we call "social networks" on the Internet are actually centralized systems -- social *servers*. For a "1:n" application, social integration - communication between two users of the same application - is trivial. Two users are two rows in the same database.

When we shift to a "n:1" model, this same integration becomes a distributed systems problem. If we're building a tictactoe app in a "1:n" design, our game is a single data structure in which moves are side effects. If we're building the same app on a network of "n:1" model, our game is a distributed system in which moves are network messages.

Building and managing distributed Internet systems is not easy. It's nontrivial to build and manage a centralized API. Deploying a new global peer-to-peer protocol is a serious endeavor.

But this is what we have to do for our tictactoe app. We have to build, for instance, some kind of identity model - because who are you playing against, an IP address? To play tictactoe, you find yourself building your own distributed social network.

While there are certainly asocial apps that a planet can run, it's not clear that an asocial planet is a viable product. If the cost of building a distributed social service isn't close to the cost of a building its centralized equivalent, the design isn't really successful.

Fortunately, while there are problems in "n:1" services that no platform can solve for you (mostly around consistency) there are set of problems with "1:n" services that no platform can solve for you (mostly around scaling). Scaling problems in "n:1" social services only arise when you're Justin Bieber and have a million friends, ie, a rare case even in a mature network. Mr. Bieber can probably afford a very nice computer.

Concrete requirements

Here are some major features we think any adequate planet needs. They're obviously all features of Urbit.

Repeatable computing

Any non-portable planet is locked in to its host. That's bad. You can have all the legal guarantees you like of unconditional migration. Freedom means nothing if there's nowhere to run to. Some of today's silos are happy to give you a tarball of your own data, but what would you do with it?

The strongest way to ensure portability is a deterministic, frozen, non-extensible execution model. Every host runs exactly the same computation on the same image and input, for all time. When you move that image, the only thing it notices is that its IP address has changed.

We could imagine a planet with an unfrozen spec, which had some kind of backward-compatible upgrade process. But with a frozen spec, there is no state outside the planet itself, no input not input to the planet itself, and no way of building a planet on one host that another host can't compute correctly.

Of course every computer is deterministic at the CPU level, but CPU-level determinism can't in practice record and replay its computation history. A computer which is deterministic at the semantic level can. Call it "repeatable computing."

Orthogonal persistence

It's unclear why we'd expose the transience semantics of the hardware memory hierarchy to either the programmer or the user. When we do so, we develop two different models for managing data: "programming languages" and "databases." Mapping between these models, eg "ORM," is the quintessence of boilerplate.

A simple pattern for orthogonal persistence without a separate database is "prevalence": a checkpoint plus a log of events since the checkpoint. Every event is an ACID transaction. In fact, most databases use this pattern internally, but their state transition function is not a general-purpose interpreter.

Identity

In the classical layer, hosts, usernames, IP addresses, domain names and public keys are all separate concepts. A planet has one routable, memorable, cryptographically independent identity which serves all these purposes.

The "Zooko's triangle" impossibility result tells us we can't build a secure, meaningful, decentralized identity system. Rather than surrendering one of these three goals, the planet can retreat a little bit on two of them. It can be memorable, but not meaningful; it can start as a centralized system, but decentralize itself over time. 100-80-70 is often preferable to 100-100-0.

A simple typed functional language

Given the level of integration we're expecting in this design, it's silly to think we could get away without a new language. There's no room in the case for glue. Every part has to fit.

The main obstacle to functional language adoption is that functional programming is math, and most human beings are really bad at math. Even most programmers are bad at math. Their intuition of computation is mechanical, not mathematical.

A pure, higher-order, typed, strict language with mechanical intuition and no mathematical roots seems best positioned to defeat this obstacle. Its inference algorithm should be almost but not quite as strong as Hindley-Milner unification, perhaps inferring "forward but not backward."

We'd also like two other features from our types. One, a type should define a subset of values against a generic data model, the way a DTD defines a set of XML values. Two, defining a type should mean defining an executable function, whose range is the type, that verifies or normalizes a generic value. Why these features? See the next section...

High-level communication

A planet could certainly use a network type descriptor that was like a MIME type, if a MIME type was an executable specification and could validate incoming content automatically. After ORM, manual data validation must be the second leading cause of boilerplate. If we have a language in which a type is also a validator, the automatic validation problem seems solvable. We can get to something very like a typed RPC.

Exactly-once message delivery semantics are impossible unless the endpoints have orthogonal persistence. Then they're easy: use a single persistent session with monotonically increasing sequence numbers. It's nice not worrying about idempotence.

With an integrated OS and protocol, it's possible to fully implement the end-to-end principle, and unify the application result code with the packet acknowledgment -- rather than having three different layers of error code. Not only is every packet a transaction at the system level; every message is a transaction at the application level. Applications can even withhold an acknowledgment to signal internal congestion, sending the other end into exponential backoff.

It's also nice to support simple higher-level protocol patterns like publish/subscribe. We don't want every application to have to implement its own subscription queue backpressure. Also, if we can automatically validate content types, perhaps we can also also diff and patch them, making remote syncing practical.

The web will be around for a while. It would be great to have a web server that let a browser page with an SSO login authenticate itself as another planet, translating messages into JSON. This way, any distributed application is also a web application.

Finally, a planet is also a great web *client*. There is lots of interesting data behind HTTP APIs. A core mission of a planet is collecting and maintaining the secrets the user needs to manage any off-planet data. (Of course, eventually this data should come home, but it may take a while.) The planet's operating system should serve as client-side middleware that mediates the API authentication process, letting the programmer program against the private web as if it was public, using site-specific auth drivers with user-configured secrets.

Global namespace

The URL taught us that any global identity scheme is the root of a global namespace. But the URL also made a big mistake: mutability.

A global namespace is cool. An immutable (referentially transparent) global namespace, which can use any data in the universe as if it was a constant, is really cool. It's even cooler if, in case your data hasn't been created yet, your calculation can block waiting for it. Coolest of all is when the data you get back is typed, and you can use it in your typed functional language just like dereferencing a pointer.

Of course, an immutable namespace should be a distributed version control system. If we want typed data, it needs to be a typed DVCS, clearly again with type-specific diff and patch. Also, our DVCS (which already needs a subscription mechanism to support blocking) should be very good at reactive syncing and mirroring.

Semantic drivers

One unattractive feature of a pure interpreter is that it exacts an inescapable performance tax -- since an interpreter is always slower than native code. This violates the prime directive of OS architecture: the programmer must never pay for any service that doesn't get used. Impure interpreters partly solve this problem with a foreign-function interface, which lets programmers move inner loops into native code and also make system calls. An FFI is obviously unacceptable in a deterministic computer.

A pure alternative to the FFI is a semantic registry in which functions, system or application, can declare their semantics in a global namespace. A smart interpreter can recognize these hints, match them to a checksum of known good code, and run a native driver that executes the function efficiently. This separates policy (pure algorithm as executable specification) from mechanism (native code or even hardware).

Repository-driven updates

A planet owner is far too busy to manually update applications. They have to update themselves.

Clearly the right way to execute an application is to run it directly from the version-control system, and reload it when a new version triggers any build dependencies. But in a planet, the user is not normally the developer. To install an app is to mirror the developer's repository. The developer's commit starts the global update process by updating mirror subscriptions.

Of course, data structures may change, requiring the developer to include an adapter function that converts the old state. (Given orthogonal persistence, rebooting the app is not an option.)

Added fun is provided by the fact that apps use the content types defined in their own repository, and these types may change. Ideally these changes are backward compatible, but an updated planet can still find itself sending an un-updated planet messages that don't validate. This race condition should block until the update has fully propagated, but not throw an error up to the user level -- because no user has done anything wrong.

Why not a planet built on JS or JVM?

Many programmers might accept our reasoning at the OS level, but get stuck on Urbit's decision not to reuse existing languages or interpreters. Why not JS, JVM, Scheme, Haskell...? The planet is isolated from the old layer, but can't it reuse mature designs?

One easy answer is that, if we're going to be replacing Unix and the Internet, or at least tiling over them, rewriting a bit of code is a small price to pay for doing it right. Even learning a new programming language is a small price to pay. And an essential aspect of "doing it right" is a system of components that fit together perfectly; we need all the simplicity wins we can get.

But these are big, hand-waving arguments. It takes more than this kind of rhetoric to justify reinventing the wheel. Let's look at a few details, trying not to get ahead of ourselves.

In the list above, only JS and the JVM were ever designed to be isolated. The others are tools for making POSIX system calls. Isolation in JS and the JVM is a client thing. It is quite far from clear what "node.js with browser-style isolation" would even mean. And who still uses Java applets?

Let's take a closer look at the JS/JVM options - not as the only interpreters in the world, just as good examples. Here are some problems we'd need them to solve, but they don't solve - not, at least, out of the box.

First: repeatability. JS and the JVM are not frozen, but warm; they release new versions with backward compatibility. This means they have "current version" state outside the planet proper. Not lethal but not good, either.

When pure, JS and then JVM are at least nominally deterministic, but they are also used mainly on transient data. It's not clear that the the actual implementations and specifications are built for the lifecycle of a planet - which must never miscompute a single bit. (ECC servers are definitely recommended.)

Second: orthogonal persistence. Historically, successful OP systems are very rare. Designing the language and OS as one unit seems intuitively required.

One design decision that helps enormously with OP is an acyclic data model. Acyclic data structures are enormously easier to serialize, to specify and validate, and of course to garbage-collect. Acyclic databases are far more common than cyclic ("network" or "object") databases. Cyclic languages are more common than acyclic languages -- but pure functional languages are acyclic, so we know acyclic programming can work.

(It's worth mentioning existing image-oriented execution environments - like Smalltalk and its descendants, or even the Lisp machine family. These architectures (surely Urbit's closest relatives) could in theory be adapted to use as orthogonally persistent databases, but in practice are not designed for it. For one thing, they're all cyclic. More broadly, the assumption that the image is a GUI client in RAM is deeply ingrained.)

Third: since a planet is a server and a server is a real OS, its interpreter should be able to efficiently virtualize itself. There are two kinds of interpreter: the kind that can run an instance of itself as a VM, and the kind that can't.

JS can almost virtualize itself with `eval`, but `eval` is a toy. (One of those fun but dangerous toys -- like lawn darts.) And while it's not at all the same thing, the JVM can load applets -- or at least, in 1997 it could...

(To use some Urbit concepts we haven't met yet: with semantic drivers (which don't exist in JS or the JVM, although asm.js is a sort of substitute), we don't even abandon all the world's JS or JVM code by using Urbit. Rather, we can implement the JS or JVM specifications in Hoon, then jet-propel them with practical Unix JS or JVM engines, letting us run JS or Java libraries.)

Could we address any or all of these problems in the context of JS, the JVM or any other existing interpreter? We could. This does not seem likely to produce results either better or sooner than building the right thing from scratch.

Definition

An operating function (OF) is a logical computer whose state is a fixed function of its input history:

```
V(I) => T
```

where `T` is the state, `V` is the fixed function, `I` is the list of input events from first to last.

Intuitively, what the computer knows is a function of what it's heard. If the `V` function is identical on every computer for all time forever, all computers which hear the same events will learn the same state from them.

Is this function a protocol, an OS, or a database? All three. Like an OS, `V` specifies the semantics of a computer. Like a protocol, `V` specifies the semantics of its input stream. And like a database, `V` interprets a store of persistent state.

Advantages

This is a very abstract description, which doesn't make it easy to see what an OF is useful for.

Two concrete advantages of any practical OF (not just Urbit):

Orthogonal persistence

An OF is inherently a single-level store. `V(I) => T` is an equation for the lifecycle of a computer. It doesn't make any distinction between transient RAM and persistent disk.

The word "database" has always conflated two concepts: data that isn't transient; structures specialized for search. A substantial percentage of global programmer-hours are spent on translating data between memory and disk formats.

Every practical modern database computes something like `V(I) => T`. `I` is the transaction log. An OF is an ACID database whose history function (log to state) is a general-purpose computer. A transaction is an event, an event is a transaction.

(Obviously a practical OF, though still defined as `V(I) => T`, does not recompute its entire event history on every event. Rather, it computes some incremental iterator `U` that can compute `U(E T0) => T1`, where `E` is each event in `I`.

In plain English, the next state `T1` is a function `U` of the latest event `E` and the current state `T0`. We can assume that some kind of incrementality will fall out of any reasonable `V`.)

In any reasonable OF, you can write a persistent application by leaving your data in the structures you use it from. Of course, if you want to use specialized search structures, no one is stopping you.

Repeatable computing

Every computer is deterministic at the CPU level, in the sense that the CPU has a manual which is exactly right. But it is not actually practical to check the CPU's work.

"Repeatable computing" is high-level determinism. It's actually practical to audit the validity of a repeatable computer by re-executing its computation history. For an OF, the history is the event log.

Not every urbit is a bank; not every urbit has to store its full log. Still, the precision of a repeatable computer also affects the user experience. (It's appalling, for instance, how comfortable the modern user is with browser hangs and crashes.)

Requirements

`V` can't be updated for the lifecycle of the computer. Or, since `V` is also a protocol, for the lifetime of the network.

So `V` needs to be perfect, which means it needs to be small. It's also easier to eradicate ambiguity in a small definition.

But we're designing a general-purpose computer which is programmed by humans. Thus `V` is a programming language in some sense. Few practical languages are small, simple or perfect.

`V` is a practical interpreter and should be reasonably fast. But `V` is also a (nonpreemptive) operating system. One universal feature of an OS is the power to virtualize user-level code. So we need a small, simple, perfect interpreter which can efficiently virtualize itself.

`V` is also a system of axioms in the mathematical sense. Axioms are always stated informally. This statement succeeds iff it accurately communicates the same axioms to every competent reader. Compressing the document produces a rough metric of information content: the complexity of `V`.

(It's possible to cheat on this test. For instance, we could design a simple `V` that only executes another interpreter, `W`. `W`, the only program written in V's language, is simply encoded in the first event. Which could be quite a large event.

This design achieves the precision of `V`, but not its stability. For stability, the true extent of `V` is the semantic kernel that the computer can't replace during its lifetime from events in the input history. Thus `V` properly includes `W`.)

There are no existing interpreters that ace all these tests, so Urbit uses its own. Our `V` is defined in 350 bytes gzipped.

Nouns

A value in Urbit is a *noun.* A noun is an *atom* or a *cell.* An atom is an unsigned integer of any size. A cell is an ordered pair of nouns.

In the system equation `V(I) => T`, `T` is a noun, `I` is a list of nouns - where a list is either `0` or a cell `[item list]`. The `V` function is defined below.

Nock

Nock or `N` is a combinator interpreter on nouns. It's specified by these pseudocode reduction rules:

```
Nock(a)          *a
[a b c]          [a [b c]]

?[a b]           0
?a               1
+[a b]           +[a b]
+a               1 + a
=[a a]           0
=[a b]           1
=a               =a

/[1 a]           a
/[2 a b]         a
/[3 a b]         b
/[(a + a) b]     /[2 /[a b]]
/[(a + a + 1) b] /[3 /[a b]]
/a               /a

*[a [b c] d]     [*[a b c] *[a d]]

*[a 0 b]         /[b a]
*[a 1 b]         b
*[a 2 b c]       *[*[a b] *[a c]]
*[a 3 b]         ?*[a b]
*[a 4 b]         +*[a b]
*[a 5 b]         =*[a b]

*[a 6 b c d]     *[a 2 [0 1] 2 [1 c d] [1 0] 2 [1 2 3] [1 0] 4 4 b]
*[a 7 b c]       *[a 2 b 1 c]
*[a 8 b c]       *[a 7 [[7 [0 1] b] 0 1] c]
*[a 9 b c]       *[a 7 c 2 [0 1] 0 b]
*[a 10 [b c] d]  *[a 8 c 7 [0 3] d]
*[a 10 b c]      *[a c]

*a               *a
```

Note that operators 6-10 are macros and not formally essential. Note also that Nock reduces invalid reductions to self, thus specifying nontermination (bottom).

A valid Nock reduction takes a cell `[subject formula]`. The formula defines a function that operates on the subject.

A valid formula is always a cell. If the head of the formula is a cell, Nock reduces head and tail separately and produces the cell of their products ("autocons"). If the head is an atom, it's an operator from `0` to `10`.

In operators `3`, `4`, and `5`, the formula's tail is another formula, whose product is the input to an axiomatic function. `3` tests atom/cell, `4` increments, `5` tests equality.

In operator `0`, the tail is a "leg" (subtree) address in the subject. Leg `1` is the root, `2n` the left child of `n`, `2n+1` the right child.

In operator `1`, the tail is produced as a constant. In operator `2`, the tail is a formula, producing a `[subject formula]` cell which Nock reduces again (rendering the system Turing complete).

In the macro department, `6` is if-then-else; `7` is function composition; `8` is a stack push; `9` is a function call; `10` is a hint.

Surprisingly, Nock is quite a practical interpreter, although it poses unusual implementation challenges. For instance, the only arithmetic operation is increment (see the implementation issues section for how this is practical). Another atypical feature is that Nock can neither test pointer equality, nor create cycles.

The fixed function

So is `V` just Nock? Almost. Where `N` is Nock, `V` is:

```
V(I) == N(I [2 [0 3] [0 2]])
```

If Nock was not `[subject formula]` but `[formula subject]`, not `N(S F)` but `N(F S)`, V would be Nock.

In either case, the head `I0` of `I` is the boot formula; the tail is the boot subject. Intuitively: to interpret the event history, treat the first event as a program; run that program on the rest of history.

More abstractly: the event sequence `I` begins with a *boot sequence* of non-uniform events, which do strange things like executing each other. Once this sequence completes, we end up in a *main sequence* of uniform events (starting at `I5`), which actually look and act like actual input.

`I0`: The lifecycle formula

`I0` is special because we run it in theory but not in practice. Urbit is both a logical definition and a practical interpreter. Sometimes there's tension between these goals. But even in practice, we check that `I0` is present and correct.

`I0` is a nontrivial Nock formula. Let's skip ahead and write it in our high-level language, Hoon:

```
=>  [load rest]=.
=+  [main step]=.*(rest load)
|-  ?~  main  step
    $(main +.main, step (step -.main))
```

Or in pseudocode:

```
from pair $load and $rest
let pair $main and $step be:
  the result of Nock $load run on $rest
loop
  if $main is empty
    return $step
  else
    continue with
      $main set to tail of $main,
      $step set to:
        call function $step on the head of $main
```

In actual Nock (functional machine code, essentially):

```
[8 [2 [0 3] [0 2]] 8 [ 1 6 [5 [1 0] 0 12] [0 13] 9 2
[0 2] [[0 25] 8 [0 13] 9 2 [0 4] [0 56] 0 11] 0 7] 9 2 0 1]
```

`I0` is given the sequence of events from `I1` on. It takes `I1`, the boot loader, and runs it against the rest of the sequence. `I1` consumes `I2`, `I3`, and `I4`, and then it produces the true initial state function `step` and the rest of the events from `I5` on.

`I0` then continues on to the main sequence, iteratively computing `V(I)` by calling the `step` function over and over on each event. Each call to `step` produces the next state, incorporating the changes resulting from the current event.

`I1`: the language formula

The language formula, `I1`, is another nontrivial Nock formula. Its job is to compile `I3`, the Hoon compiler as source, with `I2`, the Hoon compiler as a Nock formula.

If `I3`, compiled with `I2`, equals `I2`, `I1` then uses `I2` to load `I4`, the Arvo source. The main event sequence begins with `I5`. The only events which are Nock formulas are `I0`, `I1`, and `I2`; the only events which are Hoon source are `I3` and `I4`.

`I1`, in pseudo-Hoon:

```
=>  [fab=- src=+< arv=+>- seq=+>+]
=+  ken=(fab [[%atom %ud] 164] src)
?>  =(+>.fab +.ken)
[seq (fab ken arv)]
```

In pseudocode:

```
with tuple as [$fab, $src, $arv, $seq], // these are I2 I3 I4 and I5...
  let $ken be:
    call $fab on type-and-value "uint 164" and Hoon $src
  assert environment from $fab is equal to the value from $ken
  return pair of the remaining $seq and:
    call $fab on environment $ken and Hoon $arv
```

In actual Nock:

```
[7 [0 2] 8 [8 [0 2] 9 2 [0 4] [7 [0 3] [1 [1.836.020.833
25.717] 164] 0 6] 0 11] 6 [5 [0 27] 0 5] [[0 31] 8 [0 6]
9 2 [0 4] [7 [0 3] [0 2] 0 30] 0 11] 0 0]
```

(`25.717` ? Urbit uses the German dot notation for large atoms.)

We compile the compiler source, check that it produces the same compiler formula, and then compile the OS with it.

Besides `I0` and `I1`, there's no Nock code in the boot sequence that doesn't ship with Hoon source.

`I2` and `I3`: the Hoon compiler

Again, `I2` is the Hoon compiler (including basic libraries) as a Nock formula. `I3` is the same compiler as source.

Hoon is a strict, typed, higher-order pure functional language which compiles itself to Nock. It avoids category theory and aspires to a relatively mechanical, concrete style.

The Hoon compiler works with three kinds of noun: `nock`, a Nock formula; `twig`, a Hoon AST; and `span`, a Hoon type, which defines semantics and constraints on some set of nouns.

There are two major parts of the compiler: the front end, which parses a source file (as a text atom) to a twig; and the back end, `ut`, which accepts a subject span and a twig, and compiles them down to a product span and a Nock formula.

Back end and type system

The Hoon back end, (`ut`), about 1700 lines of Hoon, performs type inference and (Nock) code generation. The main method of `ut` is `mint`, with signature `$+([span twig] [span Nock])` (i.e., in pseudocode, `mint(span, twig) -> [span Nock]`).

Just as a Nock formula executes against a subject noun, a Hoon expression is always compiled against a subject span. From this `[span twig]`, `mint` produces a cell `[span Nock]`. The Nock formula generates a useful product from any noun in the subject span. The span describes the set of product nouns.

Type inference in Hoon uses only forward tracing, not unification (tracing backward) as in Hindley-Milner (Haskell, ML). Hoon needs more user annotation than a unification language, but it's easier for the programmer to follow what the algorithm is doing - just because Hoon's algorithm isn't as smart.

But the Hoon type system can solve most of the same problems as Haskell's, notably including typeclasses / genericity. For instance, it can infer the type of an AST by compiling a grammar in the form of an LL combinator parser (like Hoon's own grammar).

The Hoon type system, slightly simplified for this overview:

```
++  span  $|  $?  %noun
              %void
          ==
        $%  [%atom p=cord]
            [%cell p=span q=span]
            [%core p=span q=(map term twig)]
            [%cube p=noun q=span]
            [%face p=term q=span]
            [%fork p=span q=span]
            [%hold p=span q=twig]
        ==
```

In pseudocode:

```
define $span as one of:
  'noun'
  'void'
  'atom' with text aura
  'cell' of $span and $span
  'core' of environment $span and map of name to code AST
  'cube' of constant value and $span
  'face' of name wrapping $span
  'fork' of $span and alternate $span
  'hold' of subject $span and continuation AST
```

(The syntax `%string` means a *cord*, or atom with ASCII bytes in LSB first order. Eg, `%foo` is also 0x6f.6f66 or 7.303.014.)

The simplest spans are `%noun`, the set of all nouns, and `%void`, the empty set.

Otherwise, a span is one of `%atom`, `%cell`, `%core`, `%cube`, `%face`, `%fork` or `%hold`, each of which is parameterized with additional data.

An `%atom` is any atom, plus an *aura* cord that describes both what logical units the atom represents, and how to print it. For instance, `%ud` is an unsigned decimal, `%ta` is ASCII text, `%da` is a 128-bit Urbit date. `%n` is nil (`~`).

A `%cell` is a recursively typed cell. `%cube` is a constant. `%face` wraps another span in a name for symbolic access. `%fork` is the union of two spans.

A `%core` is a combination of code and data - Hoon's version of an object. It's a cell `[(map term formula) payload]`, i.e. a set of named formulas (essentially "computed attributes") and a "payload", which includes the context the core is defined in. Each arm uses the whole core as its subject.

One common core pattern is the *gate*, Hoon's version of a lambda. A gate is a core with a single formula, and whose payload is a cell `[arguments context]`. To call the function, replace the formal arguments with your actual argument, then apply. The context contains any data or code the function may need.

(A core is not exactly an object, nor a map of names to formulas a vtable. The formulas are computed attributes. Only a formula that produces a gate is the equivalent of a method.)

Syntax, text

Hoon's front end (`vast`) is, at 1100 lines, quite a gnarly grammar. It's probably the most inelegant section of Urbit.

Hoon is a keyword-free, ideographic language. All alphabetic strings are user text; all punctuation is combinator syntax. The *content* of your code is always alphabetic; the *structure* is always punctuation. And there are only 80 columns per line.

A programming language is a UI for programmers. UIs should be actually usable, not apparently usable. Some UI tasks are harder than they appear; others are easier. Forming associative memories is easier than it looks.

Languages do need to be read out loud, and the conventional names for punctuation are clumsy. So Hoon replaces them:
`ace [1 space] fas / ran > bar | gap [>1 space, nl] rep ' bis \ hax # rob & buc $ ket ^ sac ; cab _ lep ( sig ~ cen % lit < tar * com , lus +`
For example, `%=` sounds like "centis" rather than "percent equals." Since even a silent reader will subvocalize, the length and complexity of the sound is a tax on reading the code.

A few digraphs also have irregular sounds: `== stet -- shed ++ slus -> dart -< dusk +> lark +< lush`

Hoon defines over 100 digraph ideograms (like `|=`). Ideograms (or *runes*) have consistent internal structure; for instance, every rune with the `|` prefix produces a core. There are no user-defined runes. These facts considerably mitigate the memorization task.

Finally, although it's not syntax, style merits a few words. While names in Hoon code can have any internal convention the programmer wants (as long as it's lowercase ASCII plus hyphen), one convention we use a lot: face names which are random three-letter syllables, arm names which are four-letter random Scrabble words.

Again, forming associative memories is easier than it looks. Random names quickly associate to their meanings, even without any mnemonic. A good example is the use of Greek letters in math. As with math variables, consistency helps too - use the same name for the same concept everywhere.

This "lapidary" convention is appropriate for code that's nontrivial, but still clean and relatively short. But code can't be clean unless it's solving a clean problem. For dirty problems, long informative names and/or kebab-case are better. For trivial problems (like defining `add`), we use an even more austere "hyperlapidary" style with one-letter names.

Twig structure

A Hoon expression compiles to a `twig` , or AST node. A twig is always a cell.

Like Nock formulas, Hoon twigs "autocons." A twig whose head is a cell is a pair of twigs, which compiles to a pair of formulas, which is a formula producing a pair. ("autocons" is an homage to Lisp, whose `(cons a (cons b c))` becomes Urbit's `[a b c]` .)

Otherwise, a twig is a tagged union. Its tag is always a rune, stored by its phonetic name as a cord. Because it's normal for 31-bit numbers to be direct (thus, somewhat more efficient), we drop the vowels. So `=+` is not `%tislus` , but `%tsls` .

There are too many kinds of twig to enumerate here. Most are implemented as macros; only about 25 are directly translated to Nock. There is nothing sophisticated about twigs - given the syntax and type system, probably anyone would design the same twig system.

Syntax geometry

Just visually, one nice feature of imperative languages is the distinction between statements, which want to grow vertically down the page, and expressions, which prefer to grow horizontally.

Most functional languages, inherently lacking the concept of statements, have a visual problem: their expressions grow best horizontally, and often besiege the right margin.

Hoon has two syntax modes: "tall," which grows vertically, and "wide," which grows horizontally. Tall code can contain wide code, but not vice versa. In general, any twig can be written in either mode.

Consider the twig `[%tsls a b]` , where `a` and `b` are twigs. `=+` defines a variables, roughly equivalent to Lisp's `let` . `a` defines the variable for use in `b` . In wide mode it looks like an expression: `=+(a b)` . In tall mode, the separator is two spaces or a newline, without parentheses:

```
=+  a
b
```

which is the same code, but looks like a "statement." The deindentation is reasonable because even though `b` is, according to the AST, "inside" the `=+`, it's more natural to see it separated out.

For twigs with a constant number of components, like `[%tsls a b]`, tall mode relies on the parser to stop itself. With a variable fanout we need a `==` terminator:

```
:*  a
    b
==
```

The indentation rules of constant-fanout twigs are unusual. Consider the stem `?:`, which happens to have the exact same semantics as C `?:` (if-then-else). Where in wide form we'd write `?:(a b c)`, in tall form the convention is

```
?:  a
  b
c
```

This is *backstep* indentation. Its motivation: defend the right margin. Ideally, `c` above is a longer code block than `a` or `b`. And in `c`, we have not lost any margin at all. Ideally, we can write an arbitrarily tall and complex twig that always grows vertically and never has a margin problem. Backstep indentation is tail call optimization for syntax.

Finally, the wide form with rune and parentheses (like `=+(a b)`) is the *normal* wide form. Hoon has a healthy variety of *irregular* wide forms, for which no principles at all apply. But using normal forms everywhere would get quite cumbersome.

`I4` : Arvo

`I4` is the source code for Arvo: about 600 lines of Hoon. This is excessive for what Arvo does and can probably be tightened. Of course, it does not include the kernel modules (or *vanes*).

`I4` formally produces the kernel step function, `step` for `I0`. But in a practical computer, we don't run `I0`, and there are other things we want to do besides `step` it. Inside `step` is the Arvo core (leg 7, context), which does the actual work.

Arvo's arms are `keep`, `peek`, `poke`, `wish`, `load`.

`wish` compiles a Hoon string. `keep` asks Arvo when it wants to be woken up next. `peek` exposes the Arvo namespace. These three are formally unnecessary, but useful in practice for the Unix process that runs Arvo.

The essential Arvo arm is `poke`, which produces a gate which takes as arguments the current time and an event. (Urbit time is a 128-bit atom, starting before the universe and ending after it, with 64 bits for subseconds and 64 bits for seconds, ignoring any post-2015 leap seconds.)

The product of the `poke` gate is a cell `[action-list Arvo]`. So Arvo, poked with the time and an event, produces a list of actions and a new Arvo state.

Finally, `load` is used to replace the Arvo core. When we want to change Hoon or Arvo proper, we build a new core with our new tools, then start it by passing `load` the old core's state. (State structures in the new core need not match the old, so the new core may need adapter functions.) `load` produces its new core wrapped in the canonical outer `step` function.

Since the language certainly needs to be able to change, the calling convention to the next generation is a Nock convention. `load` is leg `54` of the core battery, producing a gate whose formal argument at leg `6` is replaced with the Arvo state, then applied to the formula at leg `2`. Of course, future cores can use a different upgrade convention. Nothing in Arvo or Urbit is technically frozen, except for Nock.

State

Arvo has three pieces of dynamic state: its network address, or *plot*; an entropy pool; the kernel modules, or *vanes*.

Configuration overview

The first main-sequence event, `I5`, is `[%init plot]`, with some unique Urbit address (*plot*) whose secrets this urbit controls. The plot, an atom under 2^128, is both a routing address and a cryptographic name. Every Urbit event log starts by setting its own unique and permanent plot.

From `I6` we begin to install the vanes: kernel modules. Vanes are installed (or reloaded) by a `[%veer label code]` event. Vane labels by convention are cords with zero or one letter.

The vane named `%$` (zero, the empty string) is the standard library. This contains library functions that aren't needed in the Hoon kernel, but are commonly useful at a higher level.

The standard library is compiled with the Hoon kernel (`I2`) as subject. The other vanes are compiled with the standard library (which now includes the kernel) as subject. Vane `%a` handles networking; `%c`, storage; `%f`, build logic; `%g`, application lifecycle; `%e`, http interface; `%d`, terminal interface; `%b`, timers; and `%j`, storage of secrets.

Mechanics

Vanes are stored as a cell of a noun with its span. At least in Urbit, there is no such thing as a dynamic type system - only a static type system executed at runtime.

Storing vanes with their spans has two benefits. One, it lets us type-check internal event dispatch. Two, it lets us do typed upgrades when we replace a vane. It makes routine dispatch operations incur compiler cost, but this caches well.

Input and output

From the Unix process's perspective, Arvo consumes events (which Unix generates) and produces actions (which Unix executes). The most common event is hearing a UDP packet; the most common action is sending a UDP packet. But Arvo also interacts (still in an event-driven pattern) with other Unix services, including HTTP, the console, and the filesystem.

Events and actions share the `ovum` structure. An ovum is a cell `[wire card]`. A wire is a path - a list of cords, like `[%foo %bar %baz ~]`, usually written with the irregular syntax `/foo/bar/baz`. A card is a tagged union of all the possible types of events and effects.

A wire is a *cause*. For example, if the event is an HTTP request `%thus`, the wire will contain (printed to cords) the server port that received the request, the IP and port of the caller, and the socket file descriptor.

The data in the wire is opaque to Urbit. But an Urbit `poke`, in response to this event or to a later one, can answer the request with an action `%this` - an HTTP response. Unix parses the action wire it sent originally and responds on the right socket file descriptor.

Moves and ducts

Event systems are renowned for "callback hell". As a purely functional environment, Arvo can't use callbacks with shared state mutation, a la node. But this is not the only common pitfall in the event landscape.

A single-threaded, nonpreemptive event dispatcher, like node or Arvo, is analogous to a multithreaded preemptive scheduler in many ways. In particular, there's a well-known duality between event flow and control flow.

One disadvantage of many event systems is unstructured event flow, often amounting to "event spaghetti". Indeed, the control-flow dual of an unstructured event system is `goto`.

Arvo is a structured event system whose dual is a call stack. An internal Arvo event is called a "move". A move has a duct, which is a list a wires, each of which is analagous to a stack frame. Moves come in two kinds: a `%pass` move calls upward, pushing a frame to the duct, while a `%give` move returns a result downward, popping a frame off the duct.

`%pass` contains a target vane name; a wire, which is a return pointer that defines this move's cause within the source vane; and a card, the event data. `%give` contains just the card.

The product of a vane event handler is a cell `[moves new-state]`, a list of moves and the vane's new state.

On receiving a Unix event, Arvo turns it into a `%pass` by picking a vane from the Unix wire (which is otherwise opaque), then pushes it on a move stack.

The Arvo main loop pops a move off the move stack, dispatches it, replaces the result vane state, and pushes the new moves on the stack. The Unix event terminates when the stack is empty.

To dispatch a `%pass` move sent by vane `%x`, to vane `%y`, with wire `/foo/bar`, duct `old-duct`, and card `new-card`, we pass `[[/x/foo/bar old-duct] new-card]`, a `[duct card]` cell, to the `call` method on vane `%y`. In other words, we push the given wire (adding the vane name onto the front) on to the old duct to create the new duct, and then pass that along with the card to the other vane.

To dispatch a `%give` move returned by vane `%x`, we check if the duct is only one wire deep. If so, return the card as an action to Unix. If not, pull the original calling vane from the top wire on the duct (by destructuring the duct as `[[vane wire] plot-duct]`), and call the `take` method on `vane` with `[plot-duct wire card]`.

Intuitively, a pass is a service request and a give is a service response. The wire contains the information (normalized to a list of strings) that the caller needs to route and process the service result. The effect always gets back to its cause.

A good example of this mechanism is any internal service which is asynchronous, responding to a request in a different *system* event than its cause - perhaps many seconds later. For instance, the `%c` vane can subscribe to future versions of a file.

When `%c` gets its service request, it saves it with the duct in a table keyed by the subscription path. When the future version is saved on this path, the response is sent on the duct it was received from. Again, the effect gets back to its cause. Depending on the request, there may even be multiple responses to the same request, on the same duct.

In practice, the structures above are slightly simplified - Arvo manages both vanes and moves as vases, `[span noun]` cells. Every dispatch is type-checked.

One card structure that Arvo detects and automatically unwraps is `[%meta vase]` - where the vase is the vase of a card. `%meta` can be stacked up indefinitely. The result is that vanes themselves can operate internally at the vase level - dynamically executing code just as Arvo itself does.

The `%g` vane uses `%meta` to expose the vane mechanism to user-level applications. The same pattern, a core which is an event transceiver, is repeated across four layers: Unix, Arvo, the `%g` vane, and the `:dojo` shell application.

Event security

Arvo is a "single-homed" OS which defines one plot, usually with the label `our`, as its identity for life. All vanes are fully trusted by `our`. But internal security still becomes an issue when we execute user-level code which is not fully trusted, or work with data which is not trusted.

Our security model is causal, like our event system. Every event is a cause and an effect. Event security is all about *who/whom*: *who* (other than us) caused this event; *whom* (other than us) it will affect. `our` is always authorized to do everything and hear anything, so strangers alone are tracked.

Every event has a security `mask` with two fields `who` and `hum`, each a `(unit (set plot))`. (A `unit` is the equivalent of Haskell's `Maybe` - `(unit x)` is either `[~ x]` or `~`, where `~` is nil.)

If `who` is `~`, nil, anyone else could have caused this move -- in other words, it's completely untrusted. If `who` is `[~ ~]`, the empty set, no one else caused this move -- it's completely trusted. Otherwise, the move is "tainted" by anyone in the set.

If `hum` is `~`, nil, anyone else can be affected by this move -- in other words, it's completely unfiltered. If `hum` is `[~ ~]`, the empty set, no one else can hear the move -- it's completely private. Otherwise, the move is permitted to "leak" to anyone in the set.

Obviously, in most moves the security mask is propagated from cause to effect without changes. It's the exceptions that keep security exciting.

Namespace

Besides `call` and `take`, each vane exports a `scry` gate whose argument is a `path` - a list of strings, like a wire.

`scry` implements a global monotonic namespace - one that (a) never changes its mind, for the lifetime of the urbit; and (b) never conflicts across well-behaved urbits.

This invariant is semantic - it's not enforced by the type system. Getting it right is up to the vane's developer. Installing kernel modules is always a high-trust operation.

The product of the `scry` gate is `(unit (unit value))`.

So `scry` can produce `~`, meaning the path is not yet bound; `[~ ~]`, meaning the path is bound to empty; or `[~ ~ value]`, an actual value is bound. This value is of the form `[mark span noun]`, where `span` is the type of the noun and `mark` is a higher-level "type" label best compared to a MIME type or filename extension. Marks are discussed under the `%f` vane.

Vane activation and namespace invariant

The vane that Arvo stores is not the core that exports `call` etc, but a gate that produces the core. The argument to this gate is a cell `[@da $+(path (unit (unit value)))]`. Note that `$+(path (unit (unit value)))` is just the function signature of the `scry` namespace.

So, the head of the argument is the current time; the tail is the system namespace. The general namespace is constructed from the `scry` method of the vanes. The first letter of the head of the path identifies the vane; the rest of the head, with the rest of the path, is passed to the vane. The vane core is activated just this way for all its methods, including `scry` itself.

This answers the question of how to expose dynamic state without violating our referential transparency invariants. If we require the query to include exactly the current time (which is guaranteed to change between events) in the path, then we are able to respond with the current state of the dynamic data. If the path doesn't contain the current time, then fail with `[~ ~]`. This technique is only needed, of course, when you don't know what the state was in the past.

Additionally, since each vane knows the plot it's on, a `scry` query with the plot in the path is guaranteed to be globally unique.

But vanes do not have to rely on these safe methods to maintain monotonicity - for instance, the `%c` revision-control vane binds paths around the network and across history.

A tour of the vanes

The vanes are separately documented, because that's the entire point of vanes. Let's take a quick tour through the system as it currently stands, however.

`%a` `%ames`: networking

`%a` is the network vane, currently `%ames` (2000 lines). `%ames` implements an encrypted P2P network over UDP packets.

As a vane, `%ames` provides a simple `%pass` service: send a message to another plot. The message is an arbitrary `[wire noun]` cell: a message channel and a message body. `%ames` will respond with a `%give` that returns either success or failure and an error dump.

`%ames` messages maintain causal integrity across the network. The sender does not send the actual duct that caused the message, of course, but an opaque atom mapped to it. This opaque `bone`, plus the channel itself, are the "socket" in the RFC sense.

Messages are ordered within this socket and delivered exactly once. (Yes, as normally defined this is impossible. Urbit can do exactly-once because `%ames` messaging is not a tool to build a consistency layer, but a tool built on the consistency layer. Thus, peers can have sequence numbers that are never reset.)

More subtly, local and remote causes treated are exactly the same way, improving the sense of network abstraction. CORBA taught us that it's not possible to abstract RPC over local function calls without producing a leaky abstraction. The Arvo structured event model is designed to abstract over messages.

One unusual feature is that `%ames` sends end-to-end acks. Acknowledging the last packet received in a message acknowledges that the message itself has been fully accepted. There is no separate message-level result code. Like its vane interface, `%ames` acks are binary: a positive ack has no data, a negative ack has a dump. Thus, Arvo is transactional at the message level as well as the packet/event level.

Another `%ames` technique is dropping packets. For instance, it is always a mistake to respond to a packet with bad encryption: it enables timing attacks. Any packet the receiver doesn't understand should be dropped. Either the sender is deranged or malicious, or the receiver will receive a future upgrade that makes a later retransmission of this packet make sense.

Messages are encrypted in a simple PKI (currently RSA, soon curve/ed25519) and AES keys are exchanged. Certificate transfer and key exchange are aggressively piggybacked, so the first packet to a stranger is always signed but not encrypted. The idea is that if you've never talked to someone before, probably the first thing you say is "hello." Which is not interesting for any realistic attacker. In the rare cases where this isn't true, it's trivial for the application to work around. (See the network architecture section for more about the PKI approach.)

(For packet geeks only: `%ames` implements both "STUN" and "TURN" styles of peer-to-peer NAT traversal, using parent plots (see below) as supernodes. If the sending plot lacks a current IP/port (Urbit uses random ports) for the destination, it forwards across the parent hierarchy. As we forward a packet, we attach the sender's address, in case we have a full-cone NAT that can do STUN.

When these indirect addresses are received, they are regarded with suspicion, and duplicate packets are sent - one forwarded, one direct. Once a direct packet is received, the indirect address is dropped. Forwarding up the hierarchy always succeeds, because galaxies (again, see below) are bound into the DNS at `galaxy.urbit.org`.)

`%c` `%clay`: filesystem

`%c` is the filesystem, currently `%clay` (3000 lines). `%clay` is a sort of simplified, reactive, typed `git`.

The `%clay` filesystem uses a uniform inode structure, `ankh`. An ankh contains: a Merkle hash of this subtree, a set of named children, and possibly file data. If present, the file data is typed and marked (again, see the `%f` vane for more about marked data).

A path within the tree always contains the plot and desk (lightweight branch) where the file is located, which version of the file to use, and what path within the desk the file is at.

Paths can be versioned in three ways: by change number (for changes within this desk); by date; or by label.

Where `git` has multiple parallel states, `%clay` uses separate desks. For instance, where `git` creates a special merge state, `%clay` just uses a normal scratch desk. Don't detach your head, merge an old version to a scratch desk. Don't have explicit commit messages, edit a log file. `%clay` is a RISC `git`.

`%clay` is a typed filesystem; not only does it save a type with each file, but it uses `mark` services from `%f` to perform typed diff and patch operations that match the mark types. Obviously, Hunt-McIlroy line diff only works for text files. `%clay` can store and revise arbitrary custom data structures, not a traditional revision-control feature.

The `scry` namespace exported by clay uses the mode feature to export different views of the filesystem. For instance, `[%cx plot desk version path]`, which passes `%x` as the mode to clay's `scry`, produces the file (if any) at that path. `%y` produces the directory, essentially - the whole ankh with its subtrees trimmed. `%z` produces the whole ankh.

`%clay` is reactive; it exports a subscription interface, with both local and network access. A simple use is waiting for a file or version which has not yet been committed.

A more complex use of `%clay` subscription is synchronization, which is actually managed at the user level within a `%gall` application (`:hood`). It's straightforward to set up complex, multistep or even cyclical chains of change propagation. Intuitively, `git` only pulls; `%clay` both pulls and pushes.

Finally, the easiest way to use `%clay` is to edit files directly from Unix. `%clay` can mount and synchronize subtrees in the Urbit home directory. There are two kinds of mount: export and sync. While an export mount marks the Unix files read-only, sync mounts support "dropbox" style direct manipulation. The Unix process watches the subtree with `inotify()` or equivalent, and generates filesystem change events automagically. A desk which is synced with Unix is like your `git` working tree; a merge from this working desk to a more stable desk is like a `git` commit.

Access control in `%clay` is an attribute of the desk, and is either a blacklist or whitelist of plots. Creating and synchronizing desks is easy enough that the right way to create complex access patterns is to make a desk for the access pattern, and sync only what you want to share into it.

`%f` `%ford`: builder

`%f` is the functional build system, currently `%ford` (1800 lines). It might be compared to `make` et al, but loosely.

`%ford` builds things. It builds applications, resources, content type conversions, filter pipelines, more or less any functional computation specified at a high level. Also, it caches repeated computations and exports a dependency tracker, so that `%ford` users know when they need to rebuild.

`%ford` does all its building and execution in the virtual Nock interpreter, `mock`. `mock` is a Nock interpreter written in Hoon, and of course executed in Nock. (See the implementation issues section for how this is practical.)

`mock` gives us two extra affordances. One, when code compiled with tracing hints crashes deterministically, we get a deterministic stack trace. Two, we add a magic operator `11` (`.^` in Hoon) which dereferences the global namespace. This is of course referentially transparent, and `mock` remains a functional superset of Nock.

`%ford` is passed one card, `[%exec silk]`, which specifies a computation - essentially, a makefile as a noun. The `silk` structure is too large to enumerate here, but some

highlights:

The simplest task of `%ford` is building code, by loading sources and resources from `%clay`. For kernel components, a source file is just parsed directly into a `twig`. `%ford` can do a lot of work before it gets to the `twig`.

The Hoon parser for a `%ford` build actually parses an extended build language wrapped around Hoon. Keeping the build instructions and the source in one file precludes a variety of exciting build mishaps.

For any interesting piece of code, we want to include structures, libraries, and data resources. Some of these requests should be in the same `beak` (plot/desk/version) as the code we're building; some are external code, locked to an external version.

`%ford` loads code from `%clay` with a *role* string that specifies its use, and becomes the head of the spur. (Ie, the first path item after plot, desk and version.) Roles: structures are in `/sur`, libraries in `/lib`, marks in `/mar`, applications in `/app`, generators in `/gen`, fabricators in `/fab`, filters in `/tip`, and anything not a Hoon file in `/doc`.

Data resources are especially interesting, because we often want to compile whole directory trees of resources into a single noun, a typed constant from the point of view of the programmer. Also, requests to `%clay` may block - `%ford` will have to delay giving its result, and wait until a subscription for the result returns.

Moreover, for any interesting build, the builder needs to track dependencies. With every build, `%ford` exports a dependency key that its customers can subscribe to, so that they get a notification when a dependency changes. For example, `%gall` uses this feature to auto-reload applications.

Finally, both for loading internal resources in a build and as a vane service (notably used by the web vane `%eyre`), `%ford` defines a functional namespace which is mapped over the static document space in `/doc`.

If we request a resource from `%ford` that does not correspond to a file in `/doc`, we search in `/fab` for the longest matching prefix of the path. The fabricator receives as its sole argument that part of the path not part of the matching prefix.

Thus, if we search for `/a/b/c/d`, and there's no `/doc/a/b/c/d`, then we first check for `/fab/a/b/c/d`, then `/fab/a/b/c`, and so on. If we find, for example, a `/fab/a/b`, then we run that fabricator with the argument `/c/d`. Thus, a fabricator can present an arbitrary virtual document tree.

The `%ford` vane also exports its namespace through `scry`, but `scry` has no way to block if a computation waits. It will just produce `~`, binding unknown.

Another major `%ford` feature is the mark system. While the Hoon type system works very well, it does not fill the niche that MIME types fill on the Internets. Marks are like MIME types, if MIME types came with with executable specifications which defined how to validate, convert, diff, and patch content objects.

The mark `%foo` is simply the core built from `/mar/foo`. If there is no `/mar/foo`, all semantics are defaulted - `%foo` is treated as `%noun`. If there is a core, it can tell `%ford` how to validate/convert from other marks (validation is always equivalent to a conversion from `%noun`, and seldom difficult, since every Hoon structure is defined as a fixpoint normalizer); convert *to* other marks; patch, diff, and merge. Obviously `%clay` uses these revision control operations, but anyone can.

For complex conversions, `%ford` has a simple graph analyzer that can convert, or at least try to convert, any source mark to any target mark.

There are two ford request modes, `%c` and `%r` - cooked and raw. A cooked request, `%fc` in `scry`, has a target mark and converts the result to it. A raw request, `%fr`, returns whatever `%ford` finds easiest to make.

Generators and filters, `/gen` and `/tip`, are actually not used by any other vane at present, but only by the user-level `:dojo` app, which constructs dataflow calculations in Unix pipe style. Applications, `/app`, are used only by the `%g` vane.

`%g` `%gall`: application driver

`%g` is the application system, currently `%gall` (1300 lines) `%gall` is half process table and half systemd, sort of.

`%gall` runs user-level applications much as Arvo runs vanes. Why this extra layer? Vanes are kernel components written by kernel hackers, for whom expressiveness matters more than convenience. A bad kernel module can disable the OS; a bad application has to be firewalled.

But the basic design of an application and a vane is the same: a stateful core that's called with events and produces moves. Only the details differ, and there are too many to cover here.

One example is that `%gall` apps don't work with Arvo ducts directly, but opaque integers that `%gall` maps to ducts; the consequences of a bad app making a bad duct would be odd and potentially debilitating, so we don't let them touch ducts directly. Also, `%gall` does not like to crash, so to execute user-level code it uses the virtual interpeter `mock`. At the user level, the right way to signal an error is to crash and let `%gall` pick up the pieces - in a message handler, for instance, this returns the crash dump in a negative ack.

`%gall` uses `%ford` to build cores and track dependencies. Like vanes, `%gall` applications update when new code is available, sometimes using adapter functions to translate old state types to new ones. Unlike Arvo, `%gall` triggers updates automatically when a dependency changes.

Other than convenience and sanity checks, the principal value add of `%gall` is its inter-application messaging protocol. `%gall` messaging supports two communication patterns: a one-way message with a success/error result (`%poke`), and classic publish and subscribe (`%peer`).

`%peer` is designed for synchronizing state. The subscriber specifies a path which defines an application resource, and receives a stream of `%full` and `%diff` cards, total and incremental updates. (A simple "get" request is a special case in which there is a single full and no diffs.) Backpressure breaks the subscription if the queue of undelivered updates grows too deep. Broken subscriptions should not cause user-level errors; the user should see an error only if the subscription can't be reopened.

Recall that `%ames` routes messages which are arbitrary nouns between vanes on different urbits. `%gall` uses these `%ames` messages for remote urbits, and direct moves for

local IPC. Thus, from the app's perspective, messaging an app on another Urbit is the same as messaging another app on the same Urbit. The abstraction does not leak.

Messages are untyped at the `%ames` level, but all `%gall` messages carry a mark and are validated by the receiver. Marks are not absolute and timeless - `%ford` needs a `beak` (plot, desk, version) to load the mark source.

When a message fails to validate, the packet is dropped silently. This puts the sender into exponential backoff retransmission. The general cause of a message that doesn't validate is that the sender's mark has received a backward-compatible update (mark updates that aren't backward compatible are a bad idea - use a new name), and the receiver doesn't have this update yet. The retransmitted packet will be processed correctly once the update has propagated.

With this mechanism, Urbit can update a distributed system (such as our own `:talk` network), upgrading both applications and protocols, silently without user intervention or notification. The general pattern of application distribution is that the app executes from a local desk autosynced to a remote urbit, which performs the function of an app store or distro. As this server fills its subscriptions, a period of network heterogeneity is inevitable; and so is transient unavailability, as new versions try to talk to old ones. But it resolves without hard errors as all clients are updated.

`%e` `%eyre` : web server/client

`%e` is the web system, currently `%eyre` (1600 lines).

`%eyre` has three purposes. First, it is an HTTP and HTTPS client - or rather, it interfaces via actions/events to HTTP client handlers in the Unix layer.

Second, `%eyre` serves the `%ford` namespace over HTTP. The URL extension is parsed as a mark, but the default mark for requests is `urb`, which creates HTML and injects an autoupdate script that long-polls on the dependencies, reloading the page in the background when they change.

The normal way of publishing static or functional content in Urbit is to rely on `%ford` for format translation. Most static content is in markdown, the `%md` mark. Dynamic content is best generated with the `sail` syntax subsystem in Hoon, which is is essentially an integrated template language that reduces to XML.

Third, `%eyre` acts as a client for `%gall` apps, translating the `%gall` message flow into JSON over HTTP. `%poke` requests become POST requests, `%peer` becomes a long-poll stream. Our `urb.js` framework is a client-side wrapper for these requests.

The abstraction does not leak. On the server side, all it takes to support web clients is ensuring that outbound marks print to `%json` and inbound marks parse from `%json`. (The standard library includes JSON tools.) The `%gall` application does not even know it's dealing with a web client, all it sees are messages and subscriptions, just like it would receive from another Urbit app.

The dataflow pattern of `%gall` subscriptions is ideal for React and similar "one-way data binding" client frameworks.

Of course, `%gall` apps expect to be talking to an urbit plot, so web clients need to (a) identify themselves and (b) talk over HTTPS. `%eyre` contains a single sign-on (SSO) flow that authenticates an urbit user either to her own server or her friends'.

Ideally, an urbit named `~plot` is DNSed to `plot.urbit.org`. If you use your urbit through the web, you'll have an insecure cookie on `*.urbit.org`. Other urbits read this and drive the SSO flow; if you're `~tasfyn-partyv`, you can log in as yourself to `~talsur-todres.urbit.org`. The SSO confirmation message is sent directly as an `%ames` message between the `%eyre` vanes on the respective urbits.

Urbit also has an nginx configuration and node cache server, which (a) let a relatively slow Urbit server drive reasonably high request bandwidth, and (b) serve HTTPS by proxy.

Note that while external caching helps `%ford` style functional publishing, it does not help `%gall` style clients, which use server resources directly. Urbit is a personal server; it can handle an HN avalanche on your blog, but it's not designed to power your new viral startup.

`%j` `%jael` : secret storage

`%j`, currently `%jael` (200 lines), saves secrets in a tree. Types of secrets that belong in `%jael`: Urbit private keys, Urbit symmetric keys, web API user keys and/or passwords, web API consumer (application) keys.

`%jael` has no fixed schema and is simply a classic tree registry. Besides a simpler security and type model, the main difference between secrets and ordinary `%clay` data is that secrets expire - sometimes automatically, sometimes manually. When another vane uses a `%jael` secret, it must register to receive an expiration notice.

`%d` `%dill` : console and Unix

`%d`, currently `%dill` (450 lines) handles the terminal and miscellaneous Unix interfaces, mainly for initialization and debugging.

The console terminal abstraction, implemented directly with `terminfo` in raw mode, gives Urbit random-access control over the input line. Keystrokes are not echoed automatically. Output lines are write-only and appear above the input line.

`%dill` also starts default applications, measures system memory consumption, dumps error traces, etc.

`%b` `%behn` : timers

`%b`, currently `%behn` (250 lines), is a timer vane that provides a simple subscription service to other vanes.

Base applications

Arvo ships with three major default applications: `:hood` (1600 lines), `:dojo` (700 lines), and `:talk` (1600 lines).

`:dojo` : a shell

`:dojo` is a shell for ad-hoc computation. A dojo command is a classic filter pipeline, with sources, filters, and sinks.

Sources can be shell variables, `%clay` files, immediate expressions, source files in `/gen`, or uploads (Unix files in `~/urbit/$plot/.urb/get`). There are three kinds of generator: simple expressions, dialog cores, and web scrapers. Generators are executed by `%ford` through `mock`, and can read the Urbit namespace via virtual operator `11`.

Urbit does not use short-lived applications like Unix commands. A `%gall` application is a Unix daemon. A generator is not like a Unix process at all; it cannot send moves. A dialog generator, waiting for user input, does not talk to the console; it tells `:dojo` what to say to the console. A web scraper does not talk to `%eyre`; it tells `:dojo` what resources it needs (GET only). This is POLA (principle of least authority); it makes the command line less powerful and thus less scary.

`:dojo` filters are immediate expressions or `/tip` source files. Sinks are console output, shell variables, `%clay`, or downloads (Unix files in `~/urbit/$plot/.urb/put`). (The upload/download mechanism is a console feature, not related to `%clay` sync - think of browser uploading and downloading.)

`:talk` : a communication bus

`:talk` is a distributed application for sharing *telegrams*, or typed social messages. Currently we use `:talk` as a simple chat service, but telegram distribution is independent of type.

Every urbit running `:talk` can create any number of "stations." Every telegram has a unique id and a target audience, to which its sender uploads it. But `:talk` stations also subscribe to each other to construct feeds.

Mutually subscribing stations will mirror, though not preserving message order. A `:talk` station is not a decentralized entity; but urbits can use subscription patterns to federate into "global" stations - very much as in NNTP (Usenet).

Stations have a flexible admission control model, with a blanket policy mode and an exception list, that lets them serve as "mailboxes" (public write), "journals" (public read), "channels" (public read/write), or "chambers" (private read/write).

Subscribers are also notified of presence and configuration changes, typing indicator, etc. Also, telegrams contain a "flavor", which senders can use to indicate categories of content that other readers may prefer not to see.

`:talk` is usable both a command-line application and a reactive web client.

`:hood` : a system daemon

`:hood` is a classic userspace system daemon, like Unix `init`. It's a system component, but it's in userspace because it can be.

`:hood` is actually a compound application made of three libraries, `/helm`, `/drum`, and `/kiln`. `/helm` manages the PKI; `/drum` multiplexes the console; `/kiln` controls `%clay` sync.

`/drum` routes console activity over `%gall` messages, and can connect to both local and foreign command-line interfaces - ssh, essentially.

One pattern in Urbit console input is that an application doesn't just parse its input after a line is is entered, but on each character. This way, we can reject or correct syntax errors as they happen. It's a much more pleasant user experience.

But since both sides of a conversation (the user and the application) are making changes to a single shared state (the input line), we have a reconciliation problem. The Urbit console protocol uses operational transformation (like Google Wave or `git replot`) for eventual consistency.

`/helm` manages public keys and (in future) hosted urbits. See the network architecture section below.

`/kiln` implements mirroring and synchronization. Any desk can mirror any other desk (given permission). Mirrors can form cycles -- a two-way mirror is synchronization.

Implementation issues

We've left a couple of knotty implementation issues unresolved up until now. Let's resolve them.

Jets

How can an interpreter whose only arithmetic operator is increment compute efficiently? For instance, the only way to decrement `n` is to count up to `n - 1`, which is O(n).

Obviously, the solution is: a sufficiently smart optimizer.

A sufficiently smart optimizer doesn't need to optimize every Nock formula that could calculate a decrement function. It only needs to optimize one: the one we actually run.

The only one we run is the one compiled from the decrement function `dec` in the Hoon standard library. So there's no sense in which our sufficiently smart optimizer needs to *analyze* Nock formulas to see if they're decrement formulas. It only needs to *recognize* the standard `dec`.

The easiest way to do this is for the standard `dec` to declare, with a hint (Nock `10`), in some formalized way, that it is a decrement formula. The interpreter implementation can check this assertion by simple comparison - it knows what formula the standard `dec` compiles to. Our sufficiently smart optimizer isn't very smart at all!

The C module that implements the efficient decrement is called a "jet." The jet system should not be confused with an FFI: a jet has *no* reason to make system calls, and should never be used to produce side effects. Additionally, a jet is incorrect unless it accurately duplicates an executable specification (the Hoon code). Achieving jet correctness is difficult, but we can spot-check it easily by running both soft and hard versions.

Jets separate mechanism and policy in Nock execution. Except for perceived performance, neither programmer nor user has any control over whether any formula is jet-propelled. A jet can be seen as a sort of "software device driver," although invisible integration of exotic hardware (like FPGAs) is another use case. And jets do not have to be correlated with built-in or low-level functionality; for instance, Urbit has a markdown parser jet.

Jets are of course the main fail vector for both computational correctness and security intrusion. Fortunately, jets don't make system calls, so sandboxing policy issues are trivial, but the sandbox transition needs to be very low-latency. Another approach would be a safe systems language, such as Rust.

The correctness issue is more interesting, because errors happen. They are especially likely to happen early in Urbit's history. A common scenario will be that the host audits an urbit by re-executing all the events, and produces a different state. In this case, the urbit must become a "bastard" - logically instantiated at the current state. The event log is logically discarded as meaningless. Hosting a bastard urbit is not a huge problem, but if you have one you want to know.

In the long run, jet correctness is an excellent target problem for fuzz testers. A sophisticated implementation might even put 10% of runtime into automatic jet testing. While it's always hard for implementors to match a specification perfectly, it's much easier with an executable specification that's only one or two orders of magnitude slower.

Event planning

How do we actually process Urbit events? If Urbit is a database, how does our database execute and maintain consistency in practice, either on a personal computer or a normal cloud server? How does orthogonal persistence work? Can we use multiple cores?

An event is a transaction. A transaction either completes or doesn't, and we can't execute its side effects until we have committed it. For instance, if an incoming packet causes an outgoing packet, we can't send the outgoing packet until we've committed the incoming one to durable storage.

Unfortunately, saving even a kilobyte of durable storage on a modern PC, with full write-through sync, can take 50 to 100ms. Solid-state storage improves this, but the whole machine is just not designed for low-latency persistence.

In the cloud the situation is better. We can treat consensus on a nontrivial Raft cluster in a data center as persistence, even though the data never leaves RAM. Urbit is highly intolerant of computation error, for obvious reasons, and should be run in an EMP shielded data center on ECC memory.

There are a number of open-source reliable log processors that work quite well. We use Apache Kafka.

With either logging approach, the physical architecture of an Urbit implementation is clear. The urbit is stored in two forms: an event log, and a checkpoint image (saved periodically, always between events). The log can be pruned at the last reliably recorded checkpoint, or not. This design (sometimes called "prevalence") is widely used and supported by common tools.

The checkpointing process is much easier because Urbit has no cycles and needs no tracing garbage collector. Without careful tuning, a tracing GC tends to turn all memory available to it into randomly allocated memory soup. The Urbit interpreter uses a region system with a copy step to deallocate the whole region used for processing each event.

Events don't always succeed. How does a functional operating system deal with an infinite loop? The loop has to be interrupted, as always. How this happens depends on the event. If the user causes an infinite loop from a local console event, it's up to the user to interrupt it with ^C. Network packets have a timeout, currently a minute.

When execution fails, we want a stack trace. But Urbit is a deterministic computer and the stack trace of an interrupt is inherently nondeterministic. How do we square this circle?

Urbit is deterministic, but it's a function of its input. If an event crashes, we don't record the event in `I`. Instead, we record a `%crud` card that contains the stack trace and the failed event. To Urbit, this is simply another external event.

Processing of `%crud` depends on the event that caused it. For keyboard input, we print the error to the screen. For a packet, we send a negative ack on the packet, with the trace. For instance, if you're at the console of urbit `A` and logged in over the network to `B`, and you run an infinite loop, the `B` event loop will time out; the network console message that `A` sent to `B` will return a negative ack; the console application on `A` will learn that its message failed, and print the trace.

Finally, the possibilities of aggressive optimization in event execution haven't been explored. Formally, Urbit is a serial computer - but it's probable that a sophisticated, mature implementation would find a lot of ways to cheat. As always, a logically simple system is the easiest system to hack for speed.

Network and PKI architecture

Two more questions we've left unanswered: how you get your urbit plot, and how packets get from one plot to another.

Again, a plot is both a digital identity and a routing address. Imagine IPv4 if you owned your own address, converted it to a string that sounded like a foreign name, and used it as your Internet handle.

Bases are parsed and printed with the `%p` aura, which is designed to make them as human-memorable as possible. `%p` uses phonemic plot-256 and renders smaller plots as shorter strings:

```
8 bits    galaxy  ~syd
16 bits   star    ~delsym
32 bits   planet  ~mighex-forfem
64 bits   moon    ~dabnev-nisseb-nomlec-sormug
128 bits  comet   ~satnet-rinsyr-silsec-navhut--bacnec-todmeb-sarseb-pagmul
```

Of course, not everyone who thinks he's Napoleon is. For the urbit to actually send and receive messages under the plot it assigns itself in the `%init` event (`I5`), later events must convey the secrets that authenticate it. In most cases this means a public key, though some urbits are booted with a symmetric session key that only works with the parent plot.

How do you get a plot? Comets are hashes of a random public key. "Civil" non-comets are issued by their parent plot - the numerical prefix in the next rank up: moons by planet, planets by star, stars by galaxy. The fingerprints of the initial galactic public keys (currently test keys) are hardcoded in `%ames` .

The comet network is fully decentralized and "free as in beer," so unlikely to be useful. Any network with disposable identities can only be an antisocial network, because disposable identities cannot be assigned default positive reputation. Perhaps comets with nontrivial hashcash in their plots could be an exception, but nonmemorable plots remain a serious handicap.

The civil network is "semi-decentralized" - a centralized system designed to evolve into a decentralized one. Urbit does not use a blockchain - address space is digital land, not digital money.

The initial public key of a civil plot is signed by its parent. But galaxies, stars and planets are independent once they've been created - they sign their own updates. (Moons are dependent; their planet signs updates.)

The certificate or `will` is a signing chain; only the last key is valid; longer wills displace their prefixes. Thus update is revocation, and revocation is a pinning race. To securely update a key is to ensure that the new will reaches any peer before any messages signed by an attacker who has stolen the old key.

The engineering details of this race depend on the actual threat model, which is hard to anticipate. If two competing parties have the same secret, no algorithm can tell who is in the right. Who pins first should win, and there will always be a time window during which the loser can cause problems. Aggressively flooding and expiring certificates reduces this window; caching and lazy distribution expands it. There is no tradeoff-free solution.

Broadly, the design difference between Urbit and a blockchain network is that blockchains are "trust superconductors" - they eliminate any dependency on social, political or economic trust. Urbit is a "trust conductor" - engineered to minimize, but not eliminate, dependencies on trust.

For instance, bitcoin prevents double-spending with global mining costs in multiple dollars per transaction (as of 2015). Trusted transaction intermediation is an easily implemented service whose cost is a tiny fraction of this. And the transaction velocity of money is high; transactions in land are far more rare.

Another trust engineering problem in Urbit is the relationship between a plot and its parent hierarchy. The hierarchy provides a variety of services, starting with peer-to-peer routing. But children need a way to escape from bad parents.

Urbit's escape principle: (a) any planet can depend on either its star's services, or its galaxy's; (b) any star can migrate to any galaxy; (c) stars and galaxies should be independently and transparently owned.

Intuitively, an ordinary user is a planet, whose governor is its star, and whose appeals court is its galaxy. Since a star or galaxy should have significant reputation capital, it has an economic incentive to follow the rules. The escape system is a backup. And the comet network is a backup to the backup.

From a privacy perspective, a planet is a personal server; a star or galaxy is a public service. Planet ownership should be private and secret; star or galaxy ownership should be public and transparent. Since, for the foreseeable future, individual planets have negligible economic value, Urbit is not a practical money-laundering tool. This is a feature, not a bug.

Finally, a general principle of both repositories and republics is that the easier it is (technically) to fork the system, the harder it is (politically) to fork. Anyone could copy Urbit and replace the galactic fingerprint block. Anyone can also fork the DNS. If the DNS was mismanaged badly enough, someone could and would; since it's competently managed, everyone can't and won't. Forkability is an automatic governance corrector.

From personal server to digital republic

Republics? Any global system needs a political design. Urbit is designed as a a *digital republic*.

The word *republic* is from Latin, "res publica" or "public thing." The essence of republican goverment is its *constitutional* quality - government by law, not people.

A decentralized network defined by a deterministic interpreter comes as close to a digital constitution as we can imagine. Urbit is not the only member of this set - bitcoin is another; ethereum is even a general-purpose computer.

The "law" of bitcoin and ethereum is self-enforcing; the "law" of Urbit is not. Urbit is not a blockchain, and no urbit can assume that any other urbit is computing the Nock function correctly.

But at least the distinction between correct and incorrect computing is defined. In this sense, Nock is Urbit's constitution. It's not self-enforcing like Ethereum, but it's exponentially more efficient.

Non-self-verifying rules are useful, too. Defining correctness is not enforcing correctness. But Urbit doesn't seek to eliminate its dependency on conventional trust. Correctness precisely defined is easily enforced with social tools: either the computation is tampered with or it isn't.

Conclusion

On the bottom, Urbit is an equation. In the middle it's an operating system. On the top it's a civilization -- or at least, a design for one.

When we step back and look at that civilization, what we see isn't that surprising. It's the present that the past expected. The Internet is what the Arpanet of 1985 became; Urbit is what the Arpanet of 1985 wanted to become.

In 1985 it seemed completely natural and inevitable that, by 2015, everyone in the world would have a network computer. Our files, our programs, our communication would all go through it.

When we got a video call, our computer would pick up. When we had to pay a bill, our computer would pay it. When we wanted to listen to a song, we'd play a music file on our computer. When we wanted to share a spreadsheet, our computer would talk to someone else's computer. What could be more obvious? How else would it work?

(We didn't anticipate that this computer would live in a data center, not on our desk. But we didn't appreciate how easily a fast network can separate the server from its UI.)

2015 has better chips, better wires, better screens. We know what the personal cloud appliance does with this infrastructure. We can imagine our 1985 future ported to it. But the most interesting thing about our planets: we don't know what the world will do with them.

There's a qualitative difference between a personal appliance and a personal server; it's the difference between a social "network" and a social network. A true planet needs to work very hard to make social programming easier. Still, distributed social applications and centralized social applications are just different. A car is not a horseless carriage.

We know one thing about the whole network: by default, a social "network" is a monarchy. It has one corporate dictator, its developer. By default, a true network is a republic; its users govern it. And more important: a distributed community cannot coerce its users. Perhaps there are cases where monarchy is more efficient and effective -- but freedom is a product people want.

But no product is inevitable. Will we even get there? Will Urbit, or any planet, or any personal cloud server, actually happen?

It depends. The future doesn't just happen. It happens, sometimes. When people work together to make it happen. Otherwise, it doesn't.

The Internet didn't scale into an open, high-trust network of personal servers. It scaled into a low-trust network that we use as a better modem -- to talk to walled-garden servers that are better AOLs. We wish it wasn't this way. It is this way.

If we want the network of the future, even the future of 1985, someone has to build it. Once someone's built it, someone else has to move there. Otherwise, it won't happen.

And for those early explorers, the facilities will be rustic. Not at all what you're used to. More 1985 than 2015, even. These "better AOLs", the modern online services that are our personal appliances, are pretty plush in the feature department.

Could they just win? Easily. Quite easily. It's a little painful herding multiple appliances, but the problem is easily solved by a few more corporate mergers. We could all just have one big appliance. If nothing changes, we will.

To build a new world, we need the right equation and the right pioneers. Is Urbit the right equation? We think so. Check our work, please. Are you the right pioneer? History isn't over -- it's barely gotten started.