What Agreement Hath Mars With Earth? An Exploration of Urbit's IPC Interfaces

~mopfel-winrux
Native Planet

Abstract

The idealistic Maxwellian principles of a system built on Nock are somewhat at odds with the evolved and tangled nature of legacy software ecosystems. In order to facilitate practical computing, Urbit presents several interfaces for communication with external ("Earth") systems. Through a comprehensive examination of Urbit's Arvo kernel and Vere runtime, alongside its high-level and low-level IPC interfaces (`%khan` and `%lick`, respectively), we elucidate the architectural mechanisms that facilitate communication and control between these disparate computing realms. This exposition aims to demystify Urbit's architecture and underscore its potential to bridge the gap between the familiar digital environments of Earth and the untapped possibilities of Martian computing.

Contents

Urbit Systems Technical Journal I:1 (2024): 1–3.

Address author correspondence to `~mopfel-winrux`.

1 Introduction

In Urbit's conception of software, two worlds exist in stark contrast: Earth, the familiar domain of established software paradigms, bustling with complex and varied digital ecosystems; and Mars, the enigmatic realm of Urbit, a system as pristine and archaic as the Martian landscape itself. This article will explore how Earth software establishes communication with Martian code.

Unlike Earth, where code is a tapestry of evolution and patchwork, Mars aims to present a regime of functional-only idealism built on Nock, the "Maxwell's laws of software". Urbit envisions a world in which software exists as a Maxwellian construct defined by a small number of axioms (`~sorreg-namtyv`, 2013). Urbit code is purely functional-as-in-language, and should be untouched and unaltered by the non-Maxwellian intricacies that characterize Earth's digital environments. Underlying this sequestration lies a profound awareness by the "Martians" that there exist diverse forms of computing with which their system must nevertheless engage. Mars accommodates these external entities by allowing them to interact in the Martian *lingua franca* of Nock nouns.

In the original public explanation of Urbit, `~sorreg-namtyv` explicitly charges,

> The general structure of cross-planet computation is that Earth always calls Mars; Mars never calls Earth. The latter would be quite impossible, since Earth code is non-Maxwellian. There is only one way for a Maxwellian computer to run non-Maxwellian code: in a Maxwellian emulator.

However, this pristine discipline has never been strictly adhered to for any real Urbit implementation. Urbit has long provided two kernel-level vanes (which one may think of as microkernel modules) for interfacing with Earth's HTTP Web services, `%eyre` (the server) and `%iris` (the client). One can even make a reasonably strong argument that Urbit running as Nock on a virtual machine specifically breaks the convention to some extent.

This article briefly explains Urbit's architecture, focusing on Arvo (Urbit's kernel) and Vere (Urbit's runtime VM). Following this, we explain three core parts of the Urbit OS interface: `%khan`, `conn.c` and `%lick`. These serve as distinct conduits between the terrestrial and the Martian, facilitating not just interaction but a nuanced form of command and control of the entire system. This exposition aims to demystify the architecture of Urbit and bridge the gap between Earth's established software paradigms and the Martian vision of computation.

2 Urbit's Unique Landscape

2.1 Arvo: The Heart of Martian Computing

The Arvo kernel (also referred to as "Urbit OS") represents a paradigm shift in operating system design. Unlike traditional operating systems that operate on preemptive multitasking and complex event networks, Arvo is a purely functional operating system, characterized by its deterministic nature and compact size. The entire Urbit stack is about 80,000 lines of code, with Arvo itself comprising around 2,000 lines. This small codebase is intentional, reflecting a philosophy that system administration complexity is directly proportional to code size.

Arvo's unique architecture avoids what is referred to as "event spaghetti" by maintaining a clear causal chain for every computation (`~sorreg-namtyv` et al., 2016). Each chain begins with a Unix I/O event and progresses through a series of steps until the computation either completes successfully or fails. This deterministic nature allows for a high level of predictability and control, a stark contrast to most Earth-based

operating systems.

The kernel's design as a "purely functional operating system"—or, more accurately if idiosyncratically, an "operating function"—underscores its uniqueness. Arvo operates on the principle that the current state is a pure function of its event log, a record of every action ever performed. This determinism is a significant deviation from the normal situation that obtains on Earth, wherein operating systems allow for arbitrary programmatic alteration of global variables affecting other programs.

Arvo innovatively handles non-determinism by refusing to log non-completed events. The system, in essence, behaves like a stateful packet transceiver—events are processed, but there is of course no guarantee of any particular event successfully completing. This pattern mirrors the behavior of packet dropping in networking. Arvo's approach to handling non-determinism is unique and aligns with Urbit's overall philosophy of simplicity and determinism. Additionally, because Arvo runs on a VM (Vere), it can obtain non-deterministic information, such as stack traces from infinite loops, through the interpreter beneath it injecting events.

2.2 Vere: The Substrate of Martian Technology

Vere, the runtime of Urbit, plays a critical role in actualizing the Martian computing model on Earth-based hardware. As the Nock interpreter written in C, Vere is intricately optimized to run Arvo, ensuring seamless translation of its deterministic operations into practical execution on conventional systems.

Vere is architecturally split into two distinct parts: the `king` (the Earth interface, more or less) and the `serf` (the Mars substrate for Nock and Arvo). The `king` component is responsible for handling vane I/O and managing the effects on and from Unix. The `serf` provisions the Arvo virtual machine (VM) and related services. This partition allows for a clear delineation of tasks within the Vere runtime, enhancing its efficiency and robustness.[1]

[1]The Ares project, in development, seeks to cleanly replace the `serf` pro-

Communication between the `king` and `serf` is achieved via an inter-process communication (IPC) port. This IPC mechanism facilitates a continuous dialogue between the two components, ensuring that the `king` can effectively manage external interactions while the `serf` maintains the integrity of the Arvo VM.

Formally, Nock (and thus Urbit) is functional-as-in-language and free of side effects. In practice, the `hint` opcode 11 in Nock is used to pass some noun to the interpreter. The interpreter can then choose a higher order action to take based on this value, whether I/O, memory management, or other side effects. This allows for the `king` to pass hints to the `serf` which produces effects on Earth (send a network packet, get a keyboard input, etc.). This bridging of the Earth/Mars gap allows for vanes to interact outside of Mars.

Event processing is key to Vere's functionality. Events from the Unix environment are passed to Vere, which then translates and injects them into Arvo. This mechanism demonstrates a profound integration between Vere and Arvo, where Vere possesses direct knowledge of Arvo's state. Notably, Vere can "scry" or query Arvo's state without altering it, retrieving information as needed. This capability allows Vere to maintain the integrity of Arvo's deterministic model while enabling dynamic interaction with the outside world.

Urbit's runtime operation is effected by using Unix as its BIOS. By leveraging the robust, widely-used Unix system, Vere ensures that Urbit can run on a broad range of hardware platforms, effectively making Urbit's Martian technology accessible and operable in Earth's diverse computing environments. This strategic use of Unix not only provides the necessary I/O and optimizations for Arvo but also ensures that Urbit's advanced and unique software architecture can function in tandem with the existing, established hardware and software ecosystems of Earth.

Moreover, Vere's design allows it to inject events into Arvo, a feature crucial for maintaining the fluid communication and interaction between the Urbit system and its underlying hard-

cess, a contingency enabled by this division of responsibility.

ware and software infrastructure. This bidirectional communication channel ensures that Arvo can respond to external stimuli while remaining true to its functional and deterministic nature. Such events, from Arvo's perspective, are simply incidental injections to the event loop.

Thus, Vere serves as more than just a bridge between Martian technology and Earthly hardware; it is a finely tuned conduit, optimized to uphold and facilitate the unique computational paradigm that Arvo embodies.

3 %khan: The High-Level Thread Interface

The %khan vane functions as a high-level thread interface for managing both internal and external communications within the Urbit ecosystem. %khan is designed to efficiently handle complex I/O operations, employing an I/O monad coupled with an exception monad. This approach allows %khan to adeptly manage complex I/O tasks while effectively handling potential failures.

%khan's core design revolves around the use of 'threads', which are monadic functions designed to handle arguments and produce results. Threads are optimized for I/O operations, leveraging the I/O monad for structured and efficient task handling. The exception monad further ensures robust management of potential failures and unexpected events. An Arvo thread is impermanent and may fail unexpectedly; it usually relies on some process external to Arvo with unknown existence or reliability. In most of its intermediate states, it expects only a small number of events (usually one), so if the thread receives anything it does not expect, it fails. Running threads are also not upgradable and fail across code upgrades. Threads are fragile but well-suited for their use case in managing such tenuous I/O operations.

The %khan vane was conceived as a way to control Urbit ships from the exterior using threads.[2] %khan's conception evolved a fair bit from proposal to implementation. In practice,

[2] %khan was originally called the "control plane" after the network routing distinction between the control plane and the data plane.

`%khan` is essentially an interface wrapper for `%spider`-based threads,[3] which produces the topsy-turvy situation in which a vane relies on a piece of userspace infrastructure to function correctly. `%khan` allows for pre-written threads that allow for easy hosting and maintenance to be bundled and distributed, helping both hosting companies and self-hosters to manage Urbit instances efficiently.

4 `conn.c`

The `conn.c` driver in Vere[4] forms part of the `king` process. It exposes a Unix domain socket at `/path/to/pier/.urb/conn.sock` for sending and receiving data from external processes.

`conn.c`'s functionality, particularly its ability to dispatch messages and handle various types of requests, is critical for the command of Urbit's control plane. It serves as a way for Unix processes to receive insights about what is happening on Urbit. `conn.c` also allows for `%khan` and Vere generally to inject raw kernel moves (events, if successful) into the system.

From a technical perspective, `conn.c` accepts newt-encoded `++jam`med nouns (defined below) which take the form

```
[request-id command arguments]
```

Newt encoding is a way of communicating nouns as single data objects in an unambiguous format.

```
V.BBBB.JJJJ.JJJJ...
```

where `V` is the version (currently `0`); `B` is the size of the `++jam`med noun in bytes (little-endian unary); and `J` is the `++jam`med noun (little-endian). This structure allows for a variety of commands to be executed, including:

1. `%ovum`, the injection of raw kernel moves;
2. `%fyrd`, a direct shortcut to `%khan` commands;

[3] `%spider` is an agent for managing transient thread-level operations, i.e. in `%gall`.

[4] `vere/io/conn.c`

3. `%urth`, runtime subcommands like `%pack` or `%meld`;
4. `%peek`, namespace scry requests into Arvo; and
5. `%peel`, emulated namespace scry requests into Vere.

A valid `conn.c` command produces a newt-encoded `++jammed` noun with type `[request-id output]`, where:

- `request-id` matches the input `request-id`; and
- `output` depends on the `command`.

An invalid `conn.c` command produces a newt-encoded `++jammed` noun with type `[0 %bail error-code error-string]`.

5 `%lick`: The Low-Level IPC Interface

Although also dealing with interprocess communication, the `%lick` vane (`~mopfel-winrux`, 2023) was designed for a very different use-case than `%khan`: to allow external processes, in particular hardware drivers, to intercommunicate with Urbit. `%lick` focuses on creating a generic noun interface over Unix domain sockets.

`%lick` manages IPC ports and the communication between Urbit applications and POSIX applications via these ports. Other vanes and applications request `%lick` to open an IPC port, notify it when something is connected or disconnected, and transfer data between itself and the Unix application. `%lick` works by opening a Unix socket for a particular process, which allows serialized IPC communications. These involve a `++jammed` noun so the receiving process needs to know how to communicate in nouns. The IPC ports texttt%lick creates are Unix domain sockets (`AF_UNIX` address family) of type `SOCK_STREAM`. Connections are made via files in a directory under the pier, `.urb/dev`.

The process on the host OS must therefore strip the first 5 bytes, `++cue` (unjam) the `++jammed` noun, check the mark, and (most likely) convert the noun into a native data structure.

Listing 1: Usage of `%lick` cards.

```
++  init  [[%pass / %arvo %l %spin /control]~ this]
++  on-arvo
  |=  [=wire =sign-arvo]
  ?+    sign-arvo  (on-arvo:def wire sign-arvo)
      [%lick %soak *]
    ?+    mark.sign-arvo  [~ this]
        %connect
      :_  this
      :~  :*  %pass  /spit  %arvo  %l
              %spit  /control  %init  area.state
          ==  ==
  ==  ==
++  send-state
  |=  =state
  ^-  card:agent:gall
  :*  %pass  /spit  %arvo  %l
      %spit  /control  %state
      [slick:state face.state food.state live.state]
  ==
```

To understand what `%lick` is doing, we need to briefly examine Unix's IPC model. IPC ("interprocess communication") describes any way that two processes in an operating system's shared context have to communicate with each other. `%lick` focuses on Unix domain sockets, which are merely file-like communication endpoints.

For instance, a valid use of `%lick` would use cards to communicate (Listing 1). Outbound moves are specified as `%spit`.

`%lick`'s vane definition is even simpler than `%khan`'s: it has no `++abet` nested-core pattern and primarily communicates to a `unix-duct` provided in its state. The `+$owner` is a `+$duct` to handle the return `%soak`.

Arvo will send three types of `%soaks` to an open `%lick` port. These `%soaks` may be one of:

1. `%connect` when the first connection is established on the Earth-side IPC port;

2. `%disconnect` when the last connection is broken on the Earth-side IPC port; or

3. `%error` when an error occurs.

`%lick` also will send a `%disconnect %soak` to every agent when Vere is started.

Gall needs to wrap `%soak` and `%spit` to route properly (see e.g. `++ap-generic-take`). This permits multiple agents to share sockets with the same name while still organizing an agent's sockets by a unique path.

6 Conclusion

This exploration into Urbit's IPC interfaces, in particular `%khan`, `conn.c`, and `%lick`, has sought to illuminate the architectural innovations that distinguish Urbit from traditional computing paradigms. These mechanisms permit Vere to balance Arvo's demand for pristine determinism against the host Unix OS's many competing software interfaces and tools. While Urbit itself obviously relies deeply on these, following this pattern in other systems could lead to a redefinition of how digital ecosystems interact. ⛆

References

~mopfel-winrux (2023) "UIP-0101: %lick. An IPC Vane". URL: https://github.com/urbit/UIPs/blob/main/UIPS/UIP-0101.md (visited on ~2024.1.25).

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

— (2013) "Nock 4K". URL: https://docs.urbit.org/language/nock/reference/definition (visited on ~2024.2.20).

~sorreg-namtyv, Curtis Yarvin et al. (2016). *Urbit: A Solid-State Interpreter*. Whitepaper. Tlon Corporation.