What Hast Earth to do With Mars: An explanation of Urbit's IPC interfaces

~mopfel-winrux
Native Planet

Abstract

In the evolving landscape of digital computation, the interaction between conventional software paradigms and innovative systems like Urbit presents a unique challenge. This paper delves into the architectural intricacies and interfaces of Urbit, contrasting its pristine, Maxwellian principles against the complex, evolved nature of Earth's software ecosystems. Through a comprehensive examination of Urbit's kernel (Arvo) and runtime (Vere), alongside its high-level and low-level IPC interfaces (`%khan` and `%lick`, respectively), we elucidate the mechanisms that facilitate communication and control between these disparate computing realms. Our analysis 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

Manuscript submitted for review.

Address author correspondence to `~mopfel-winrux`.

1 Introduction

In the realm 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 establish a communication with the Martian code.

Unlike Earth, where software is a tapestry of evolution and patchwork, Mars presents a realm of purity. It is a world where software exists as a Maxwellian construct, it is defined by a small set of axioms (**Nock4k**). Urbit code is untouched and unaltered by the non-Maxwellian intricacies that characterize Earth's digital environments. Yet, in this isolation lies a profound awareness - an understanding by the Martians that there exist diverse forms of computing with which they must engage. Mars, accommodates these external entities by allowing them to interact in the Martian language of nouns.

In the original explanation of of Urbit, **Yarvin2010**, explicitly says "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." This however has never been true for Urbit as it was implemented. Urbit has two vanes (best thought of as kernel modules) `%eyre` and `%iris` that are written as a specific interface to Earth (a web server and client, respectively). One can even make the argument that Urbit running on a virtual machine specifically breaks this convention.

This paper begins with a brief explanation of Urbit's architecture, focusing on Arvo (Urbit's kernel) and Vere (Urbit's Runtime). Following this, is an explanation of three core parts of the Urbit OS: `%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. Through this exploration, we aim 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

Arvo, also known 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 being only 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. 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, 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 norm, where operating systems allow for programmatic alteration of global variables affecting other programs.

Arvo handles non-determinism in an innovative way. The system, in essence, behaves like a stateful packet transceiver – events are processed, but there's no guarantee of the event successfully completing. This mirrors the behavior of packet dropping in networking. This approach to handling non-determinism is unique and aligns with Arvo'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.

The architecture of Vere is split into two distinct parts: the `king` and the `serf`. The `king` component is responsible for handling vane I/O and managing the effects on Earth. In contrast, the `serf` operates as the process running the Arvo virtual machine (VM). This division of responsibilities allows for a clear delineation of tasks within the Vere runtime, enhancing its efficiency and robustness.

Communication between the `king` and `serf` is achieved through 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.

A core opcode in Urbit's assembly language (Nock) is the `hint` opcode (opcode 11). The `hint` opcode passes some value to the interpreter. The interpreter can then choose what higher order action to take based on this value. For example, the simplest use of the `hint` opcode is the ordinary "debug printf". A more complex use of the opcode allows Urbit to pass hints to Vere which produces effects on Earth (send a network packet, get a keyboard input, etc.); bridging the Earth/Mars gap and allowing for vanes to interact outside of Mars.

Key to Vere's functionality is how it processes events. 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 in-

teraction with the outside world.

Urbit's use of Unix as its BIOS is an important aspect of its runtime operation. 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 hardware and software infrastructure. This bidirectional communication channel ensures that Arvo can respond to external stimuli while remaining true to its functional and deterministic nature.

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 of Urbit

`%khan`, as a crucial component of Urbit's architecture, functions as a high-level thread interface, pivotal for managing both internal and external communications within the Urbit ecosystem. It 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.

At its core, `%khan`'s design revolves around the use of 'threads', which are monadic functions designed to handle arguments and produce results. These 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. The weakness of threads is that it's impermanent and may fail unexpectedly. Since it is usually relying on some external In most of its intermediate states, it expects only a small number of events (usually one), so if it receives anything it didn't expect, it fails. When code is upgraded, it's impossible to upgrade a running thread, so it fails.

`%khan` was conceived as a way to control Urbit ships from the exterior using threads. The concept evolved a fair bit from proposal to implementation. In practice, `%khan` is essentially an interface wrapper for Spider-based threads, which produces a somewhat strange (but not unprecedented) situation in which a vane relies on a piece of userspace infrastructure to function correctly. `%spider` is an agent for transient thread-level operations.

`%khan` allows for pre-written threads that allow for easy hosting and maintenance to be bundled and distributed; helping hosting companies and self-hosted ships to easily run efficiently.

4 conn.c

conn.c[1] is a driver in Vere. It is a part of the "King" process. It exposes a Unix domain socket[2] at /path/to/pier/.urb/conn.sock for sending/receiving data from external processes.

The functionality of conn.c, 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. It also allows for pass commands either %khan and the injection of raw kernel moves (events) into the system.

From a technical perspective, conn.c accepts newt-encoded ++jamed nouns (defined below) which take the form [request-id command arguments]. The newt-encoded format is:

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

Where

- V is the version (currently 0)
- B is the size of the ++jamed noun in bytes (little endian)
- J is the ++jamed 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. %urth - runtime subcommands like %pack or %meld
4. %peek - namespace scry requests into Arvo
5. %peel - emulated namespace scry requests into Vere

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

- request-id matches the input request-id
- output depends on the command

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

[1] https://github.com/urbit/vere/blob/develop/pkg/vere/io/conn.c
[2] https://en.wikipedia.org/wiki/Unix_domain_socket

5 %lick - The Low Level IPC Interface

Although also dealing with interprocess communication, `%lick` (**UIP-101**) 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 domain sockets.

`%lick` manages IPC ports, and the communication between Urbit applications and POSIX applications via these ports. Other vanes and applications ask `%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 the `SOCK_STREAM` type. The connections are made via file path in `.urb/dev` of the pier.

The process on the host OS must therefore strip the first 5 bytes, `++cue`[3] the `++jamed` noun, check the mark and (most likely) convert the noun into a native data structure.

To understand what `%lick` is doing, we need to look at Unix's IPC model briefly. 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[4], which are just communication endpoints[5].

For instance, a valid use of `%lick` would use cards that look like this:

```
++  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
      ~&  >  "connect"
      :_  this
          [%pass /spit %arvo %l %spit /control %init area.state]~
      ==
  ==
::
++  send-state
  |=  =state
  ^-  card:agent:gall
  :*  %pass
```

[3] https://docs.urbit.org/language/hoon/reference/stdlib/2p#cue
[4] https://en.wikipedia.org/wiki/Unix_domain_socket
[5] https://man7.org/linux/man-pages/man7/unix.7.html

```
      /spit
      %arvo %l
      %spit
      /control
      %state
      [slick:state face.state food.state live.state]
  ==
```

The vane definition of `%lick` is even simpler than `%khan`: it has no `++abet` core and primarily communicates to the `unix-duct` 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` are `%connect` when the first connection is established on the Earth-side IPC port, `%disconnect` when the last connection is broken on the Earth-side IPC port, and `%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 lets multiple agents share sockets with the same name, while placing an agents sockets in a unique file path.

6 Conclusion

This exploration into Urbit's IPC interfaces, specifically `%khan` and `%lick`, has illuminated the profound architectural innovations that distinguish Urbit from traditional computing paradigms. By delving into the inner workings of Arvo and Vere, as well as the mechanisms facilitating communication between these components, we've uncovered a realm where simplicity and determinism reign supreme. This study not only demystifies the complex underpinnings of Urbit but also highlights its potential to redefine our interaction with digital ecosystems.

Urbit's design philosophy, rooted in the principles of a more principled and coherent digital universe, challenges us to reconsider the foundations upon which our current software ecosystems are built. The contrast between the evolved, patchwork nature of Earth's software and the Maxwellian purity of Urbit's architecture serves as a poignant reminder of the potential for innovation in software design. As we stand at the crossroads of Earth's complexity and Mars' simplicity, the lessons drawn from Urbit's IPC interfaces beckon us towards a future where software is not only a tool but a reflection of a more streamlined and purposeful digital existence.

The journey through Urbit's landscape offers a vision of what digital computation could become: a domain where efficiency, clarity, and integrity are not aspirational but foundational. As we continue to navigate the evolving landscape of software development, the insights garnered from this analysis of Urbit will undoubtedly inspire further exploration and innovation. In embracing the principles elucidated by Urbit's design, we pave the way for a future where the digital and human experiences are more intimately aligned, fostering a computing environment that is both revolutionary and deeply resonant with the principles of simplicity and purpose that guide our

exploration of the digital frontier.