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

~mopfel-winrux
Native Planet

Abstract

In this groundbreaking paper, we embark on a cosmic journey with the renowned character Ernest P. Worrell as he ventures into the unexplored realm of Martian computing. Drawing inspiration from Ernest's comically ingenious encounters with everyday challenges, we investigate the foundations of what we term "Artificial Stupidity." As Ernest grapples with Martian technology, we delve into the intricacies of programming errors, algorithmic missteps, and the curious phenomena that arise when human-like intelligence meets extraterrestrial computing systems. Our analysis sheds light on the unexpected intersections between humor, artificial intelligence, and the cosmic absurdity of Martian software. Join us in this interplanetary exploration as we unravel the mysteries of Artificial Stupidity through the lens of Ernest's interstellar escapades.

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 sets forth on an expedition to unravel the mysteries of this Martian software landscape, exploring how the intricate processes of terrestrial technology can establish a dialogue 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, 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, in its wisdom, accommodates these external entities, allowing them to interact in a dialect familiar to its inhabitants: the language of nouns.

Our journey into this uncharted territory begins with a deep dive into the architecture of Urbit, focusing on the foundational layer upon which Arvo, Urbit's kernel, operates. This exploration will illuminate the operational dynamics and philosophical underpinnings that make Urbit a universe apart. Following this, we delve into the intricacies of two pivotal components of Urbit's ecosystem: `%khan` and `%lick`. These vanes serve as distinct conduits between the terrestrial and the Martian, facilitating not just interaction but a nuanced form of control and collaboration with Arvo. Through this exploration, we aim to demystify the enigma of Urbit and bridge the cosmic gap between Earth's established software paradigms and the Martian vision of computational purity and simplicity.

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 30,000 lines of code, with Arvo itself being only around 1,000 lines. This small codebase is intentional, reflecting a philosophy that system administration complexity is directly proportional to code size.

Arvo's design philosophy positions it not just as an operating system but as a new frontier in the peer-to-peer internet space. Unlike traditional operating systems like Windows, macOS, or Linux, Arvo doesn't aim to replace them; instead, it offers a unique user experience akin to a web browser, yet it is a fully-fledged OS in its own

right. This distinctive approach enables Arvo to function within a virtual machine, theoretically capable of running on bare metal.

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's terminal cause. This deterministic nature allows for a high level of predictability and control, a stark contrast to the non-deterministic nature of 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 nondeterminism in an innovative way. The system, in essence, behaves like a stateful packet transceiver – events are processed, but there's no guarantee of completion, mirroring the behavior of packet dropping in networking. This approach to handling nondeterminism is unique and aligns with Arvo's overall philosophy of simplicity and determinism.

Arvo's determinism is a key feature, stacking it atop a frozen instruction set known as Nock. This concept, though new to operating systems, is not foreign to computing. Similar to how CPU instruction sets like x86-64 are frozen at the chip level, Arvo freezes the instruction set at a higher level, enabling deterministic computation. This high-level determinism contrasts sharply with the non-determinism typically found in Earth's operating systems.

Handling nondeterminism in Arvo is akin to a heuristic decision-making process similar to dropping a packet in networking. This approach views Arvo as a stateful packet transceiver, where the completion of events is not guaranteed, a stark difference from traditional computing models. Additionally, because Arvo runs on a VM, it can obtain nondeterministic information, such as stack traces from infinite loops, through the interpreter beneath it, further enhancing its operational capabilities.

The vision of Urbit, with Arvo at its core, is to transition from developer-hosted web services on multiple foreign servers to self-hosted applications on a personal server. Each architectural decision in Arvo aligns with this vision, emphasizing user ownership and management of data. This approach sets it apart from the conventional model of cloud-based services and centralized data management prevalent in Earth-based systems.

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.

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 interaction with the outside world.

Urbit's use of Unix as its BIOS is a pivotal 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 IO monad coupled with an exception monad. This dual-mechanism approach allows Khan to adeptly manage complex IO tasks while effectively handling potential failures.

Khan's role in Urbit can be likened to that of a control plane. Its primary function is to execute threads via a Unix Socket and relay the results back. This setup supports various thread-running modes, enhancing the flexibility of Urbit's computing environment and allowing for diverse interaction methods with both internal and external processes.

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 IO monad for structured and efficient task handling. The exception monad further ensures robust management of potential failures and unexpected events.

Khan's implementation ensures that Urbit serves as a personal server, replacing multiple developer-hosted web services on foreign servers with self-hosted applications on a single personal server. Its design underscores Urbit's commitment to user ownership and data management, setting it apart from the conventional model of cloud-based services and centralized data management prevalent in Earth-based systems.

4 conn.c

conn.c[1] is a driver in Vere. It is a part of the "King" (a.k.a. "Urth") 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 into the system. The implication of this means that whoever is running your urbit runtime controls your urbit.

From a technical perspective, conn.c accepts newt-encoded ++jammed nouns which take the form [request-id command arguments]. 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 jammed 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].

4.1 %ovum

The argument to an %ovum command is a raw kernel move which is injected directly into the Arvo event loop. This is a very powerful - and potentially dangerous - tool. For example, if a ship somehow got into a state where Clay was no longer working properly (meaning new files could not be compiled to fix the state of the kernel), the source code for a new, working Clay could be directly injected into the ship using an %ovum.

The output of an %ovum command is:

- [%news %done] if the move completed successfully
- [%news %drop] if the move was dropped
- [%bail goof] if an error occurred

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

4.2 %fyrd

`%fyrd` is a direct shortcut to the Khan vane. The arguments to a `%fyrd` command are (in order):

1. The name of the desk in which the thread lives (e.g. `%base`) or `beak` for the thread (e.g. `[%zod %base %10]`)
2. The name of the thread (e.g. `%hi`)
3. Mark to which the output should be cast (e.g. `%tape`)
4. Mark for how to interpret the input argument to thread (e.g. `%ship`)
5. Input argument to thread (e.g. `zõd`)

The output of a `%fyrd` command is `[%avow (each page goof)]`, the value of `each` depending on whether the thread succeeded or not.

4.3 %urth

The argument to the `%urth` command is a subcommand for the action to perform. Currently, the only valid commands are `%pack` and `%meld`.

`%urth` will return `%&` if given a valid command as input, otherwise it will return `[0 %bail 0xffffffff %urth-bad]`. No other output is emitted.

4.4 %peek

The `%peek` command is used to perform a namespace read request (a.k.a. scry) using Arvo's external peek interface `arm +22 in arvo.hoon`[3]. The argument to `%peek` is the `nom` input to `+peek` in `arvo.hoon` (`lyc` is auto-filled as `[ ]`, i.e., "request from self"). That is to say that the argument to `%peek` must have type:

```
$+  each  path
$%  [%once vis=view syd=desk tyl=spur]
    [%beam vis=view bem=beam]
==
```

Practically speaking, this means that the input will look like one of these three examples:

```
[%& p=path]
[%| p=[%once vis=view syd=desk tyl=spur]]
[%| p=[%beam vis=view bem=beam]]
```

Where:

- `path` is a `[view beam]`, with the `view` passed in as a `coin`

[3] `https://github.com/urbit/urbit/blob/develop/pkg/arvo/sys/arvo.hoon#L1774`

- `view` is the vane code for the scry, as well as an optional care, possibly appended to the vane (e.g. `%j`, `%gx`, etc.)
- `beam` is a `[beak spur]`
- `desk` is used to auto-generate a `beak`: `[our desk now]`
- `spur` is the scry endpoint for the agent or vane

The output of a `%peek` command is `[%peek (unit (unit scry-output))]`, where means that the scry endpoint is invalid, and `[ ]` means that the scry resolved to nothing.

4.5 %peel

`%peel` attempts to emulate a scry-like namespace, like the one used by Arvo and accessed by `%peek`. The argument to `%peel` should be a path. Valid paths result in a non-null `unit` containing the result of the scry. Invalid paths result in null (i.e.). The valid paths and the data they return are:

```
/help   (unit (list path))  Supported %peel paths
/live   (unit ~)            Pier health check; succeeds if pier is running
/khan   (unit ~)            Khan health check; succeeds if Khan vane is running
/info   (unit mass)         Pier info as a mass
/v      (unit @t)           Returns version of the Vere binary as a cord
/who    (unit @)            Returns the Azimuth identity of the ship as an atom
```

Note that the pier info above is returned as a `mass` report, i.e. type `(pair cord (each * (list mass)))`. This is not the same as the `|mass` memory report. `/mass` is meant to be a valid `%peel` path which returns the `|mass` memory report, but it is currently unimplemented.

5 %lick

Altho also dealing with interprocess communication, Lick was designed for a very different scenario than Khan: to allow external processes, in particular hardware drivers, to intercommunicate with Urbit. (This breached the Earth/Mars divide.) Thus `/sys/vane/lick` focuses on instrumenting a low-level noun interfaces 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 Lick creates are Unix domain sockets (`AF_UNIX` address family) of the `SOCK_STREAM` type.

The connexions are made via filepaths in `.urb/dev` of the pier.

The format is:

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

- `V` is the version
- `B` is the jam size in bytes (little endian)
- `J` is the jammed noun (little endian)

The process on the host OS must therefore strip the first 5 bytes, `++cue`[4] the jamfile, check the mark and (most likely) convert the noun into a native data structure.

5.1 `/sys/lull` Definition

```
::                                                  ::::
::::                    ++lick                        ::  (1j) IPC
  ::                                                ::::
++  lick  ^?
  |%
  +$  gift                                          ::  out result <-$
    $%  [%spin =name]                               ::  open an IPC port
        [%shut =name]                               ::  close an IPC port
        [%spit =name =mark =noun]                   ::  spit a noun to the IPC port
        [%soak =name =mark =noun]                   ::  soak a noun from the IPC port
    ==
  +$  task                                          ::  in request ->$
    $~  [%vega ~]                                   ::
    $%  $>(%born vane-task)                         ::  new unix process
        $>(%trim vane-task)                         ::  trim state
        $>(%vega vane-task)                         ::  report upgrade
        [%spin =name]                               ::  open an IPC port
        [%shut =name]                               ::  close an IPC port
        [%spit =name =mark =noun]                   ::  spit a noun to the IPC port
        [%soak =name =mark =noun]                   ::  soak a noun from the IPC port
    ==
  ::
  +$  name  path
  --  ::lick
```

To evaluate what `/sys/vane/lick` is doing, we need to look at Unix's IPC model briefly. IPC ("interprocess communication") describes any way that two processes in

[4] `https://docs.urbit.org/language/hoon/reference/stdlib/2p#cue`

an operating system's shared context have to communicate with each other. `%lick` focuses on Unix domain sockets[5], which are just communication endpoints[6].

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

The vane definition of `/sys/vane/lick` is even simpler than `/sys/vane/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`.

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, and each agent can have its own folder.

6 Conclusion

To summarize, why did you write it? Why do we care? What impact should it have on Urbit development?

Your bibliography is a separate BibTeX file. We use the `plainnat` bibliography style. You can use `natbib` citation commands like `\citep{wikipedia}` for parenthesized references. Use `\citet{wikipedia}` for inline references. You can also use `\citeauthor{wikipedia}` for the principal author's name.

"You can use traditional TeX—or LaTeX—representations" [Varney, 1987].

"Or you can use fancy quotes—and symbols."

[5] `https://en.wikipedia.org/wiki/Unix_domain_socket`
[6] `https://man7.org/linux/man-pages/man7/unix.7.html`

References

Jim Varney. Ernest goes to camp. Film, 1987. URL `https://www.imdb.com/title/tt0092974/`. Directed by John R. Cherry III. Produced by Stacy Williams. Written by John R. Cherry III and Coke Sams. Starring Jim Varney, Victoria Racimo, John Vernon, Iron Eyes Cody, Lyle Alzado, Gailard Sartain, Daniel Butler, Patrick Day, Scott Menville, Jacob Vargas, and Todd Loyd. Buena Vista Pictures.