Hyperware: A General-Purpose Sovereign Cloud Computer

Benjamin McCormick, Nicholas B Ludwig, Markus Vaas, and James Foley

Sybil Technologies AG

Abstract. Hyperware is a software platform designed to integrate all facets of modern crypto application development. Users can run their own services at both the interface and backend level. Corporations or other entities can provide services in a permissionless, protocolized manner. This node-based cloud computing model resolves the impedance mismatch between onchain protocols and web services. Developers can write apps in any programming language that compiles to Wasm and easily distribute them to nodes. The Hyperware platform includes several components that will be described in this whitepaper: a virtual machine, an onchain global namespace, a utility token for expanding and assigning a value-topology to that namespace, a PKI (Public-Key Infrastructure), a peer-to-peer networking protocol, a modular smart contract account system, and finally a governance apparatus to distribute onchain assets and continue development of the platform. All of these components work in lockstep to solve the problems that have heretofore discouraged developers from embracing peer-to-peer computing.

July 16 2025
Revision 3

Table of Contents

1 Overview . . . 3
2 Hypermap . . . 4
2.1 Example Hypermap Entries . . . 6
3 HNS: Hyperware Name System . . . 7
3.1 Specification . . . 7
3.2 Indexing . . . 8
3.3 Adding Other Onchain Identity Primitives . . . 8
4 Hyperware OS . . . 9
4.1 WIT . . . 10
4.2 Microkernel . . . 13
4.3 Message Passing . . . 13
4.4 Capabilities-Based Security . . . 15
4.5 System Primitives . . . 17
4.6 Example Process . . . 17
4.7 Selected Runtime Modules . . . 18
4.8 Runtime Extensions . . . 22
4.9 Backwards Compatibility . . . 22
5 Package Manager . . . 23
5.1 Specification . . . 24
5.2 Package Metadata . . . 25
5.3 Package Manifest . . . 26
5.4 Hyperdrive App: App Store . . . 27
6 Kit . . . 28
7 Hypermap Advanced . . . 29
7.1 Top-Level Zones . . . 29
7.2 Name-Keys . . . 30
7.3 Data-Keys . . . 30
7.4 Extensibility . . . 31
7.5 Counterfactual Addresses For Hyperware Smart Accounts . . . 31
7.6 ERC-6551 Token-Bound Accounts . . . 32
7.7 Scaling . . . 32
7.8 Review . . . 33
8 HYPR Token . . . 33
8.1 Registration . . . 33
8.2 Discussion . . . 34
8.3 Current and Future Uses . . . 36
9 Hyperware Governance . . . 36
9.1 Voting . . . 37
9.2 TLZ Management . . . 37
9.3 Progressive Decentralization . . . 38
9.4 Default-distro App: Governance Portal . . . 39

9.5 Other Duties 39
10 A Hyperware Future 40
11 Appendix: 3 Ways to Use Hyperware 41

1 Overview

Cryptocurrency, specifically smart contract blockchains, triggered a nascent revolution in permissionless protocols: software that allows participation by all, yet can be shut down by none. But progress towards a fully decentralized, permissionless Internet has stagnated even as specific niches like decentralized finance flourish. We believe that this progress is constrained less by blockchain speed and throughput as it is by an underdeveloped offchain computing substrate.

Recall the singular problem blockchains are designed to solve: preventing the double-spending of a cryptographically-owned asset.[1] The mechanism to achieve this, now well-proven, expanded to turing-complete VMs, and replicated dozens of times, is simple: signed transactions confirmed by a decentralized validator set and deposited into an append-only distributed ledger.

But what operations *actually benefit* from an onchain transaction?

Permissionless finance obviously requires blockchains, at least at the moment of settlement, as do operations that mutate ownership of an asset. Smart contracts have proven that double-spend prevention can productively be applied to any digital asset that requires guaranteed global consensus on the order and provenance of operations.

Which operations do not benefit from such guarantees? For one, any action that only requires a signature from a single public key and does not need to be ordered: a signed message from an individual, or an API published by an entity acting as the single source of truth. Blockchain transactions are similarly unnecessary for actions undertaken between trusted parties, which, in fact, comprise a large portion of online communication. It turns out that the category of networked operations that *do not* require global consensus is much larger than the category of those that do benefit from being transactions.

Some protocols, like many in decentralized finance, function perfectly fine with no user interaction outside their onchain transaction protocol. The user interface for such a protocol is merely a wrapper over the smart contract deployed onchain. There is a vast landscape of possible protocols, however, that do not fit entirely into the purely-onchain paradigm. Forcing these protocols into this model has led to countless failures.

Some believe that merely increasing transaction throughput by a few orders of magnitude (no small task) can resolve the issue by allowing transactions to be cheaply verified, even when they are not strictly necessary. We do not. There will always be significant relative costs to placing transactions in a globally-distributed ledger—not only monetary, but also in terms of latency, data storage,

[1] The Bitcoin whitepaper describes using a distributed ledger to prevent double-spend of Bitcoin, and subsequent cryptocurrency projects generalized this to a blockchain preventing double-spend of arbitrary digital assets.

and compute overhead. Transactions will *always* present an impedance mismatch and be an inferior technical solution for operations that do not require global consensus. Until we provide a proper platform for such operations, "Web3" will simply never outcompete "Web2".

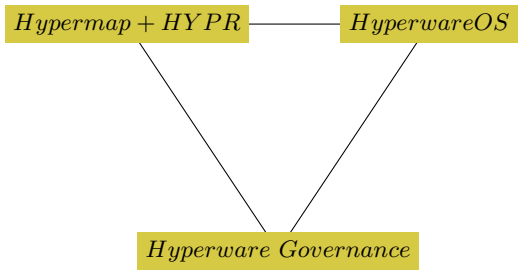

Fig. 1. Components of Hyperware.

The goal of the Hyperware Hyperstructure[2] is to present a permissionless substrate for computing. Smart contract blockchains provide access to global consensus state for provenance over digital assets, while Hyperware does "everything else". In this paper, we present an operating system, an onchain global namespace, a value assignment and ranking mechanism, and the long-term governance structure for these components.

2 Hypermap

Historically, discoverability of both *peers* and *content* has been a major barrier for peer-to-peer developers. Discoverability can present both social barriers (finding a new user on a game or chat) and technical obstacles (automatically acquiring networking information for a particular username). Many solutions have been designed to address this problem, but so far, the "devex" (developer experience) of deploying centralized services has continued to outcompete the peer-to-peer discoverability options available. Why is this?

1. Libraries such as `libp2p`, while effective at their goal of providing peer-to-peer primitives, do not provide the "batteries included" identity, discoverability, and network-effect-potential of more traditional centralized alternatives, and can also be difficult to approach for new developers.
2. "Pure" peer-to-peer protocols still rely on hardcoded lists to bootstrap new entrants.
3. Constructs such as distributed hash tables and CRDTs, frequently used in peer-to-peer protocols, are complex to properly implement.

[2] https://jacob.energy/hyperstructures.html

4. In order for a full, up-to-date snapshot of some globally-shared data to be easy to aquire, it should be stored in a single place.
   – For Hyperware, that "one place" is on a public blockchain inside a single smart contract.
   – Multiple map contracts across multiple chains can be used to scale horizontally in the future while still providing a consistent interface to the global state.
   – All data necessary to bootstrap peer-to-peer interaction must be available within this globally-shared map.
   – Any "missing piece" required to complete handshakes or source peers will result in unreliability and re-centralization.

Hypermap is an onchain key-value store inspired by dmap[3] , a minimalist onchain path-formatted key-value store. It serves as the base-level shared global state that all nodes use to share critical signaling data with the entire network. Like dmap, Hypermap is organized as a hierarchical path system and has mutable and immutable keys. Several aspects of the minimal map implementation are customized for the "namespace" use case.

A brief description:

1. All keys are strings containing exclusively characters 0–9, a–z (lowercase), - (hyphen).
2. A key may be one of two types, a name-key or a data-key.
3. Every name-key may create sub-entries directly beneath it.
4. Every name-key is an ERC-721[4] NFT (non-fungible token), with a connected token-bound account with a counterfactual address, implemented according to the ERC-6551[5] standard.
5. The implementation of a token-bound account may be set when a name–key is created (unless parent name-key has set a "gene": see below).
6. A name-key may optionally specify a "gene": a token-bound account implementation that all of its child name-keys will inherit. Children of a "gene"-specifying parent will have both their "gene' and their token-bound account implementation set to the implementation specified by their parent's "gene".
7. Every name-key may inscribe data in data-keys directly beneath it.
8. A data-key may be mutable (a "note", prepended with `~`) or immutable (a "fact", prepended with `!`)

For a complete specification, see Advanced Hypermap in Section 7, which goes into detail regarding token-bound accounts, sub-entry management, the use of data-keys, and protocol extensibility. For a description of the value-assignment mechanism that overlaps Hypermap, see Section 8 for the HYPR utility token.

[3] https://github.com/dapphub/dmap
[4] https://eips.ethereum.org/EIPS/eip-721
[5] https://ercs.ethereum.org/ERCS/erc-6551

2.1 Example Hypermap Entries

```
os
    foo
        ~ip
        ~ws-port
        ~tcp-port
        ~net-key
    bar
        ~routers
        ~net-key
hypr
    baz
        package
            ~metadata-hash
            ~metadata-uri
        !this-is-permanent
        ~this-is-mutable
eth
    alice
        ~routers
        ~net-key
    bob
        ~routers
        ~net-key
```

Fig. 2. Example Hypermap.

Fig. 2 shows an example with three top-level "zones", `eth`, `hypr`, and `os`. Below those are a number of namespace entries: `foo`, `bar`, and `baz`. The full path for `foo`'s `~ip` sub-entry would be `~ip.foo.os`.

In this paper, we will sometimes use the term "domain" interchangeably with what we refer to here as a "namespace entry". This is a useful shorthand, and in many ways, Hypermap does mirror the role of DNS in the worldwide web. However, the domain analogy is inaccurate if applied directly to all namespace entries because not all namespace entries resolve to a networking protocol target.

E.g., only entries containing `~net-key` are used by the HNS (Hyperware Name System) which runs *on top* of Hypermap. Since entries `baz.hypr` and `package.baz.hypr` have no data-keys to describe their status in the HNS, the "domain" analogy breaks down for them. The design of Hypermap is generic in the sense that many protocols are expected to share this global namespace for different purposes. The specification of HNS itself, as a protocol operating on Hypermap, is described in Section 3, as is the specification of the Hyperware Package Manager, Section 5, both of which make an appearance in this example.

Entries `hypr`, `baz.hypr`, and `package.baz.hypr` are *all* NFTs, and *all* have associated token-bound accounts (so are all name-keys: '`os`', etc.). The owner address of a namespace entry (usually) has singular control of its token-bound account. However, a given account implementation may contain arbitrary logic, such as the ability for anyone to mint a sub-entry, or edit a "note" key underneath. A name-key may set its "gene" to a token-bound account implementation which causes *all the name-key's subsequently minted children to use that implementation for their token-bound account implementation and "gene".*

3 HNS: Hyperware Name System

Hyperware Name System is a protocol built on top of Hypermap that acts as a PKI for the network. HNS transforms an entry in the Hypermap namespace into a *node identity* for use in the Hyperware network, where a node is an instance of Hyperware OS, able to communicate peer-to-peer with other such nodes. Node identities are central to the programming model of Hyperware OS. Usually manipulated as strings in a process, a node identity is the first component of an `address`, which uniquely identifies a specific process running on that node, part of a package, published by some node. See Section 4 for definitions and discussion.

3.1 Specification

The definition of a node identity in the HNS protocol is any Hypermap entry that has:

1. A `~net-key` note AND
2. (a) A `~routers` note OR
   (b) An `~ip` note AND at least one of:
      i. `~tcp-port` note
      ii. `~udp-port` note
      iii. `~ws-port` note
      iv. `~wt-port` note

A sample of this protocol can be seen in Fig. 2. Two classes of nodes are defined: *direct* and *indirect*. Direct nodes are those that publish an `~ip` and one or more of the `port` notes. Indirect nodes are those that publish `~routers`. The nature of direct and indirect nodes in networking is described in Section 4.7.

The data stored at `~net-key` must be 32 bytes corresponding to an Ed25519 public key. This is a node's signing key which is used across a variety of domains to verify ownership, including in the end-to-end encrypted networking protocol between nodes. The owner of a namespace entry/node identity may rotate this key at any time by posting a transaction to Hypermap mutating the data stored at `~net-key`.

The bytes at a `~routers` entry must parse to an array of 32-byte values. Each 32-byte value should be a keccak256 namehash that resolves to a node identity.

These strings should be node identities. Each node in the array is treated by other participants in the networking protocol as a router for the parent entry. Routers should themselves be direct nodes. If a string in the array is not a valid node identity, or it is a valid node identity but not a direct one, that router will not be used by the networking protocol. Further discussion of the networking protocol specification is presented in the Section 4.7.

The bytes at an `~ip` entry must be either 4 or 16 big-endian bytes. A 4-byte entry represents a 32-bit unsigned integer and is interpreted as an IPv4 address. A 16-byte entry represents a 128-bit unsigned integer and is interpreted as an IPv6 address.

Lastly, the bytes at any of the following port entries must be 2 big-endian bytes corresponding to a 16-bit unsigned integer:

1. `~tcp-port` note
2. `~udp-port` note
3. `~ws-port` note
4. `~wt-port` note

These integers are translated to port numbers. In practice, port numbers used are between 9000 and 65535. Ports between 8000-8999 are usually saved for HTTP server use.

3.2 Indexing

Events emitted by Hypermap are used to index map data. Hyperware OS provides all the primitives required to index effectively in a userspace application, and the default distribution of the OS includes a process that indexes Hypermap for the purpose of reporting HNS data to the networking protocol module.

3.3 Adding Other Onchain Identity Primitives

The HNS is not an attempt at replacing or competing with existing onchain identity primitives such as ENS[6] and Lens[7]. Rather, it is designed to satisfy the public key infrastructure needs of the Hyperware network. It is of paramount importance that nodes can initiate secure communication with one another without the use of any data other than what is available publicly onchain. Peer-discovery middlemen induce centralization and complicate networking protocols.

The structure of Hypermap means that HNS avoids competition with other identity primitives by seamlessly integrating them. As of this paper, this has already been done for ENS protocol. Here is a brief description of the procedure to do so:

1. Create a contract to allow owners, and only owners, of a given identity primitive to mint their corresponding name in a Hypermap namespace controlled by this contract.

[6] https://ens.domains
[7] https://lens.xyz

2. Mint and transfer the top-level namespace entry corresponding to an outside identity primitive, `lens` for example.
3. If necessary, configure LayerZero, or another such cross-chain messaging protocol, to allow owners of an identity primitive on another chain to verify their ownership on the chain that Hypermap is deployed on.
4. The final result is that the owner of, for example, `myname.lens` can exclusively register `myname.lens` in Hypermap, add a `~net-key` sub-entry, and use it as their PKI entry for Hyperware.

4 Hyperware OS

This section discusses the architecture of Hyperware OS. For a more "hands-on" description of the OS, including detailed programming examples and documentation, go to book.hyperware.ai.

Hyperware OS is a process virtual machine run to operate a "node" on the Hyperware network. At its core, the virtual machine wraps around a Wasm runtime[8] that executes all userspace code. After a node identity is registered onchain in the HNS, the operator should boot the OS using the private key matching the public `~net-key` posted in the Hypermap during registration. Once this has been done, if the networking details (routers, IP, etc) are properly read from the Hypermap and matched by the runtime, that node is now "online". Other nodes can interact with the booted node through the Hyperware networking protocol by reading its HNS node identity and using the data stored there.

The runtime is as simple as possible, with a maximal amount of logic ejected to userspace. In the future, Hyperware will benefit from "client diversity" as does a traditional blockchain: many implementations of the virtual machine[9] will make the network more resilient to potential bugs and decentralize the development process, leading to productive ossification of core features, stability, and long-term strength. Sybil Technologies distributes the reference runtime: Hyperdrive.

While all packaged into a single executable, the OS can productively be described in 3 parts: a *runtime*, a set of *runtime modules*, and *userspace*. The "kernel" frequently referred to in this paper is in fact just a runtime module.

The runtime is a "native" (to whatever architecture it targets, e.g., Unix, a browser, hardware, ...) program that manages node booting (including onchain registration) and (generally asynchronously or in parallel) executes the runtime modules. Runtime modules are blocks of code written at the same level of abstraction as the runtime itself, but designed to resemble userspace processes. These modules are registered in the kernel *as* processes, meaning that they

[8] Wasm is specified at https://webassembly.github.io/spec. Hyperware uses Wasm for processes because it is a highly performant, language independent, portable, and sandboxable compilation target.

[9] The Wasm runtime is by a wide margin the most complex aspect of the OS, and at least a dozen such runtimes exist today, written in multiple languages. This bodes well for future Hyperware client diversity.

can be messaged by userspace in the system-wide request-response protocol and secured via capabilities. Finally, userspace consists of all non-runtime-module processes executed virtually by the kernel. Userspace processes are always compiled to Wasm and comport to the Wasm Component Model[10]. Userspace Wasm code must be compiled against a Wasm Interface Type (WIT) file as used in the Wasm Component Model which defines the common set of "system calls" and types afforded to userspace processes by the kernel.

Hyperdrive, the reference implementation, is written in Rust as are all the runtime modules. It also comes with a number of pre-installed userspace packages that perform critical tasks. In the future, other entities will likely seek to distribute their own implementations that may contain different pre-installed packages or even different runtime modules.

```
eth:distro:sys
http-client:distro:sys
http-server:distro:sys
kernel:distro:sys
kv:distro:sys
net:distro:sys
state:distro:sys
terminal:distro:sys
timer:distro:sys
sqlite:distro:sys
vfs:distro:sys
```

Fig. 3. The full list of runtime modules in Hyperdrive (the reference implementation) as of this writing.

4.1 WIT

Wasm Interface Type[11], or WIT, is a language to describe types and function definitions that can be used in a Wasm component. Hyperware OS uses a single WIT file[12] to define the types shared across all processes and provide a number of functions. The functions fall into three categories:

1. Self-configuration
2. Capabilities management
3. Message I/O

[10] https://component-model.bytecodealliance.org
[11] https://component-model.bytecodealliance.org/design/wit.html
[12] https://github.com/hyperware-ai/hyperware-wit

WIT files are organized into "worlds". All types and functions provided to Hyperware processes are currently stored in one world labeled `lib`. In a separate world, `process-v1`, a single function named `init` is exported, which means that all Wasm apps that use the `process-v1` world must implement that function. The Hyperware kernel starts executing a process by calling the process's `init`.

WIT Types Discussion of the types presented here will occur throughout the rest of the OS description. Some types in hyperware.wit are omitted for brevity or because they are discussed later.

```
// JSON is passed over Wasm boundary as a string.
type json = string;

type node-id = string;

type context = list<u8>;

record process-id {
    process-name: string,
    package-name: string,
    publisher-node: node-id,
}
record address {
    node: node-id,
    process: process-id,
}
record lazy-load-blob {
    mime: option<string>,
    bytes: list<u8>,
}
```

Fig. 4. Basic types in hyperware.wit

An `address` globally identifies a process running on a particular node.

A `process-id` identifies a particular process by its publisher, package name, and process name.

WIT Host Functions WIT host functions are implemented by the kernel. The Wasm Component model allows these functions to be called by processes.

```
// self-configuration
print-to-terminal()
set-on-exit()
get-on-exit()
get-state()
set-state()
clear-state()
spawn()

// capabilities management
save-capabilities()
drop-capabilities()
our-capabilities()

// message I/O
receive()
get-blob()
send-request()
send-requests()
send-response()
send-and-await-response()
```

Fig. 5. Host functions in hyperware.wit

WIT Process Format The process format enforced by hyperware.wit is remarkably simple: it imports the types and functions defined in the main library world, and requires processes to implement a single function: `init`.[13]

```
world process {
    include lib;
    export init: func(our: string);
}
```

Fig. 6. Process world in hyperware.wit

`init` serves as the entry point for a process. The kernel begins execution of a process by calling `init`. When `init` returns, the process will cease execution.[14]

[13] This does not preclude processes from implementing other functions.

[14] All processes are single-threaded. To perform parallel computation, one can spawn child processes.

4.2 Microkernel

Every aspect of the operating system, including the kernel itself, comports to a set of messaging rules defined by the microkernel[15] which is responsible for five things:

1. Using a Wasm runtime[16] to execute compiled processes that implement the Hyperware WIT standard, where execution includes managing their memory usage.
2. Implementing the host functions, exposed to all processes, defined in Hyperware WIT standard.
3. Implementing the kernel API that allows processes with kernel-messaging capabilities to perform aspects of process management.
4. Passing messages between all processes including to/from the kernel itself.
5. Enforcing messaging capabilities.

Messaging capabilities are a subset of the capabilities security model defined by the OS, issued by the kernel process, `kernel:distro:sys`. Each process can mark itself as either `public` or `private` at instantiation. Public processes can be messaged by any other process. Private processes, as enforced by the kernel, require that the message source holds their messaging capability. See the discussion of capabilities in Section 4.4 for details on their use and how capabilities apply to processes running a remote node.

As of this writing, the kernel runtime module in Hyperdrive fits into about 2,500 lines of Rust code.

The kernel is responsible for maintaining backwards compatibility. If a process was written for an older version of the kernel (which is determined by the version number of the WIT file it implements), newer kernels must store that WIT version and match it to the process. If data structures in the WIT file change between versions, the kernel is responsible for translating between formats. The current version of the WIT file is 1.0.0. Backwards compatibility will be permanently maintained for all subsequent versions.

4.3 Message Passing

A message between two Hyperware processes is either a request or a response. A message has a single source `address` and a single target `address`.

[15] A microkernel architecture is ideal for a "Wasm OS" because it allows a maximal amount of system logic to live in Wasm itself, allowing system code to experience all the safety and conveniences afforded to userspace processes. See https://wiki.osdev.org/Microkernel

[16] Hyperdrive uses Wasmtime.

```
record request {
    inherit: bool,
    expects-response: option<u64>,
    body: list<u8>,
    metadata: option<json>,
    capabilities: list<capability>,
}
record response {
    inherit: bool,
    body: list<u8>,
    metadata: option<json>,
    capabilities: list<capability>,
}
variant message {
    request(request),
    response(tuple<response, option<context>>),
}
```

Fig. 7. Message type in hyperware.wit

Messages are produced and consumed by Hyperware processes.

If the node identity indicated in the target `address` matches that of the local kernel or is simply the string `our`, the message is routed directly through the kernel to the target (assuming the source has the capability to message the target or the target is public). Otherwise, the message is routed through the networking runtime module, `net:distro:sys`, to the remote node indicated.

Ordering of messages between a given source and a given target is enforced by both kernel and networking protocol. Messages are not otherwise ordered, meaning that if process A sends messages 1, 2, 3 to process B, and process B sends messages 4, 5, 6 to A, no guarantees are enforced other than that process B will receive messages 1, 2, 3 in that order and process A will receive 4, 5, 6 in that order. If process C sends message 7 to A, it may be received before 1, after 3, or somewhere in between.

Message delivery is not guaranteed. If a message targets a local process, the target may crash or the kernel may suspend execution between message creation and delivery. Far more treacherous is the delivery of messages to remote processes. Nodes may go offline, experience network congestion, or otherwise drop incoming messages.[17] To this end, two error modes are baked into Hyperware message passing: offline and timeout.

Requests can be sent at any time, while responses must target a process that has a matching outstanding request. A request is outstanding if:

[17] Computer networking, being a fundamentally physical process, is impossible to effectively abstract over without failure modes because the physical world imposes them.

1. It expects a response
2. Its timeout has not expired

Every request that expects a response must set a timeout value, measured in seconds. The kernel is responsible for returning a timeout error to a request that expects a response and does not get matched to one within the number of seconds declared.[18]

The offline error type is only returned by `net:distro:sys`. It may be returned if the node that a request targets is definitively unreachable. This may occur if a direct node's networking information in HNS is invalid, an indirect node has no routers, a node refuses all networking protocol connections / does not comport to protocol, or any other such immediate error. In practice, "offline" and "timeout" can usually be treated the same way: by a combination of alerting the program user and retrying the message.

Protocols for retrying a message, particularly to a remote process, are left to userspace. Different applications are best served by different retry strategies: a one-off message may be awaited in a blocking fashion, surfacing an error to the sender's UI. A message that is part of a large data transfer may not expect a response at all, instead relying on the final message in the transfer to await a response. The networking protocol's role as a general-purpose messaging system means that it must support all of these use cases and more.

4.4 Capabilities-Based Security

Hyperware OS uses capabilities[19] to enable sensible security between both userspace processes and runtime modules. Security between programs is directly related to the sovereignty goals of the OS: a user must be able to install a program without needing to evaluate its source code or blindly trust its developer. Wasm programs are sandboxed, but have access to powerful tools including networking, memory, CPU, and disk space—not to mention the possible secrets they contain (consider a wallet program). Not only must these tools be granted to programs on a case-by-case basis, but without some form of control between sandboxed programs, the sandbox becomes pointless, as any power or secret knowledge granted to a given program could be accessed by other programs!

[18] Timeouts must be viewed as a lower bound, as in, the kernel will not return a timeout error for at least X seconds. The upper bound cannot be guaranteed.

[19] Capabilities are "unforgeable tokens of authority", validated by the kernel, that allow processes to acquire privileges both at the runtime and userspace level. See the paper "Capability Myths Demolished" for a good introduction to the topic.

```
record capability {
    issuer: address,
    params: json,
}
```

Fig. 8. Capability type in hyperware.wit

Capabilities are signed by the local kernel's `~net-key` to convert them into unforgeable tokens of authority. Processes don't need to concern themselves with verifying signatures. Instead, the kernel filters out local capabilities that are not properly signed. If a process is in possession of a capability, it may send it to another process. Remote capabilities—capabilities created by a different kernel—are not verified. Why not? If an invalid remote capability is created and passed in a message, the holder will be alerted to its invalid nature if/when the holder tries to use it.

Software written on Hyperware OS will often benefit from declaring a set of capabilities desired at the time of install (see further discussion in Section 5). Many of the built-in runtime modules distributed with the OS, including the kernel itself, have a capabilities protocol. The kernel's capabilities protocol is part of the OS specification because it applies to every process and is the bedrock security model of the OS. It is also very simple:

1. Upon instantiation, every process is given its own *messaging capability.*
2. Every process may mark itself as `public`.
3. A messaging capability is defined as a capability with the `issuer` field set to the process in question, and the `params` field set to the string `"messaging"`[20]
4. If a process is public, the kernel will pass any message to it. If not, the kernel will check that local processes sending requests to this process are in possession of the messaging capability.

Note that remote processes are not filtered by messaging capabilities. Because other nodes could run modified runtimes that spoof such information as process names, it does not make sense to filter by process name for a remote message. Instead of using messaging capabilities to filter remote processes, a process instead may decide whether or not it accepts messages from remote sources in general, and then filter by the identity of the source node which cannot be spoofed. The situation may change with the advent of trusted computing, since remote nodes can be guaranteed to be running an unmodified runtime.

[20] Quotation marks included here to produce valid JSON, as is best practice for the params field.

4.5 System Primitives

Hyperware OS manages four primitives via runtime modules:

1. Networking: sending encrypted messages between nodes using permanent cryptographic identities.
2. Data Persistence: writing to disk with the option to use remote backup systems.
3. Global Consensus State: integrating with blockchains to read data and write transactions.
4. Web: HTTP client and server.

The commonality between these four items is the requirement for I/O. Therefore, they cannot be built as userspace Wasm processes. Instead they are written as runtime modules: chunks of code at the same native level of the runtime and specially registered as processes in the kernel, which is itself a runtime module.

Networking, data persistence, blockchain access, and HTTP read/write are all presented to userspace processes as a request-response API between a runtime module and the process using the primitive. See Fig. 3 for a full list of the process IDs that present these primitives. The API for a given runtime module included in the `distro` package is part of the set of interfaces grouped within the Hyperware OS versioning system. The OS uses a single semantic versioning number to indicate breaking and non-breaking changes to these APIs, the kernel, the HNS/Hypermap onchain protocols, and the networking protocol. This is covered further in the discussion of backwards compatibility in Section 4.9.

A few not-strictly-necessary but useful I/O primitives are also presented as APIs via runtime modules. These are: a terminal, a timer, and the advanced data persistence options of SQLite, a key-value store, and a virtual filesystem.

Hyperware OS can expose new primitives at the runtime level via *extensions*, covered in Section 4.8.

4.6 Example Process

Now that the OS has been described in the abstract, and before we dive in to the specific designs of various runtime modules, it may be helpful to provide a code sample showing what a process actually looks like.

```
wit_bindgen::generate!({
    path: "wit",
    world: "process-v1",
});

struct Component;
export!(Component);

use crate::hyperware::process::standard::print_to_terminal;

impl Guest for Component {
    fn init(our: String) {
        print_to_terminal(0, "hello from a process");
        print_to_terminal(
            0,
            &format!("our process-id: {our}")
        );
    }
}
```

Fig. 9. A process implemented in Rust.

By generating bindings from `hyperware.wit`, a process acquires a set of types and functions from the langauge in which it is written. The types and functions generated are often cumbersome to use directly due to their basic nature—in practice nearly all processes will use a library written for their particular langauge that smooths over the WIT interface and provides helper functions, type implementations, and so on.[21] To generate WIT bindings, it is merely required to import `hyperware.wit` and use the guest language's tooling, in the case of this example wit-bindgen[22] for Rust.

4.7 Selected Runtime Modules

This section describes a number of runtime modules of critical importance.

Virtual Filesystem: The operating system ships with `vfs:distro:sys`, a module that presents a standard filesystem API accessible to all processes with the capability to message it. Directories and files created in the VFS are saved on the host machine's filesystem. All I/O is mediated by the VFS, allowing processes to abstract away management of filesystem resources.

[21] As of this writing, most processes have been written in Rust and an extensive library of this description has already been written, available at https://github.com/hyperware-ai/process_lib

[22] https://github.com/bytecodealliance/wit-bindgen

All processes that wish to persist data locally between boots will use either the VFS or another runtime module that writes to disk, which may be an extension or one of the default-distribution's SQLite or key-value store modules.

Networking: This runtime module is the part of the OS that implements the Hyperware networking protocol. This module is somewhat special: in the kernel, messages with a `target` that contains a node identity other than that of the kernel are all routed to `net:distro:sys`. Once a message is passed to the networking module, it is routed to the target node using the information available in the Hyperware Name System. For this reason, the networking module *must* be made aware of the current onchain state of the HNS.

HNS updates are given to `net:distro:sys` using a request API made available to processes that have messaging capabilities to it. Note that messaging capability to `net` is only required to send configurational messages, and is not required to simply send networked messages: those are handled through the kernel. Note also that as the HNS state grows, it will become prudent to not load the entire state into the networking module, but rather to dynamically query the state as networking information is required for accessing new nodes or updating stale data from known ones.

The networking protocol itself will not be fully specified here[23]. However, some key aspects:

- The protocol exclusively uses information available onchain, including IP addresses, ports, and router nodes, to facilitate message-passing. See the description of HNS in Section 3.
- Direct nodes publish their routing information onchain. Indirect nodes publish a set of routers who facilitate message-passing for them. This is analogous to STUN+TURN in WebRTC. A router may be able to facilitate a direct connection for indirect nodes in a STUN-like manner.
- Networking may occur across many underlying transport protocols. The specification of HNS in the Hypermap allows for a node identity to publicize the port to be used for each transport protocol that node supports. The runtime is responsible for implementing each protocol that a node broadcasts. In practice, nodes will mostly use TCP for direct communication and routers will support a variety of protocols used by special-purpose nodes (such as mobile devices or nodes running in-browser).
- Messages are end-to-end encrypted using a Noise protocol[24] where each node has a static public key in their Ed25519 key published onchain, the cipher function is ChaChaPoly, and hash function is BLAKE2s. The XX pattern is used for handshakes.[25]
- Message-passing in a networked context aims to be as similar as possible to the local context. However, the offline and timeout error-cases together cover the inescapable realities of networking.

[23] See https://book.hyperware.ai/system/networking_protocol.html
[24] http://www.noiseprotocol.org/noise.html
[25] This is described in Noise as "Noise_XX_25519_ChaChaPoly_BLAKE2s".

– There are no ACKs at the Hyperware protocol level: if the underlying transport protocol confirms delivery, failure to do so can become an offline error. Otherwise, the request-response pattern must be used to confirm message delivery.

HTTP Client & Server: Web access is a critical part of many programs that run on Hyperware. A built-in HTTP client module allows processes to ingest data from the web. The HTTP server module allows processes to serve data to the web, either statically by sending a payload to be served for a given path, or dynamically by requesting the server module to forward incoming HTTP requests to the process. Both modules present a request-response API (documented elsewhere).

Most programs that present an interface to the node user do so through a combination of static and dynamic HTTP server path bindings. All paths bound by a process are prefixed with the `process-id`, in order to remove the possibility of collisions or imposter resources. Further, paths may be bound as "authenticated", meaning that access will require a JSON Web Token (JWT) given via password login to the node. Once the token is acquired via login, a user may access any authenticated path served by processes on the node.

This raises a subtle security issue: if a process serves an API allowing the user to perform various actions over HTTP (as is quite common), other processes running on the same node can easily access the API by serving a frontend JavaScript file of their own, which can fetch content from any authenticated path on the node. This circumvents the capability-based security model applied to inter-process communication and suddenly shifts the trust assumptions for all software installed on a node that acquires the capability to message the HTTP modules.

Rather than allow this escalation of trust assumptions, the HTTP server provides a special mechanism to serve authenticated paths that are only accessible at a *subdomain*. The security model of the browser is thus leveraged to generate a new JWT for the subdomain URL, required for access to the paths at the subdomain and disallowing client access from the base domain. Users may then login at the subdomain with the same password as the base node server and generate a new token in their browser. We call this model "Secure Subdomains". Processes that serve sensitive HTTP APIs for user interfaces are encouraged to use Secure Subdomains to properly sandbox these operations away from other software running on the user's node.

ETH RPC: The OS includes an Ethereum (and EVM-compatible chains) indexing runtime module, `eth:distro:sys`, which provides read and write access to blockchain data. This module can connect directly to WebSocket RPC endpoints or relay through other Hyperware nodes, forming a potential chain of relays.

The module implements standard Ethereum JSON-RPC API methods, supporting operations such as querying block data, retrieving account balances, esti-

mating gas costs, and sending transactions. Processes interact with this module through a request-response API, typically using a `Provider` struct that encapsulates chain-specific details and request handling.

Optional `.eth_providers` and `.eth_access_settings` JSON files in the node's home folder may be used to configure the module. The former allows users to specify their preferred RPC endpoints, relay nodes, and chain-specific settings, and the latter controls whether other nodes may use this node as a provider with potential allow/deny lists. The configuration can also be modified at runtime through the module's API, enabling flexible provider management.

The module supports both one-time requests and subscriptions, particularly useful for monitoring real-time events specific log entries or Hypermap note keys. Subscriptions are managed through unique identifiers, allowing processes to filter and unsubscribe from event streams as needed.

`eth:distro:sys` also integrates with Hyperware-specific functionalities, such as querying the Hypermap contract, which is central to Hyperware's naming and identity system. This allows processes to interact with Hyperware-specific on-chain data seamlessly.

```
let node = namehash("node.foo.os");
let (tba, owner, note) = provider.hypermap_get(&node)?;
```

The module acts as a relay and subscription manager for Ethereum JSON-RPC requests and responses. Processes may use helper functions and structs to format requests according to the Ethereum JSON-RPC specification[26].

`eth:distro:sys` handles maintenance of subscriptions, managing provider connections, and facilitating the routing of Ethereum data between Hyperware nodes, allowing processes to have direct control over their Ethereum interactions while benefiting from the module's network management capabilities, including the ability to relay requests through other nodes when direct RPC access is unavailable or not yet configured.

This design facilitates the development of decentralized applications that can efficiently interact with global blockchain networks, even in constrained peer-to-peer environments. It allows for unique scaling possibilities:

- Applications can default to public endpoints while allowing users to easily switch to their own nodes.
- Nodes without direct RPC access can relay through peers, distributing network load.
- Multi-chain applications can be built with a unified interface, simplifying development across different EVM-compatible networks.

By providing this flexible and powerful interface to Ethereum and other EVM chains, Hyperware OS enables developers to create robust blockchain-integrated applications while giving users control over their blockchain access points.

[26] https://ethereum.github.io/execution-apis/api-documentation/

SQLite, KV-Store: In addition to the VFS, the OS provides SQLite and key-value store runtime modules. These modules serve as high-performance disk storage options for processes that require persistence. Each module has a request-response API (documented elsewhere) exposing their respective operations. Both SQLite and KV-store structure requests such that processes with the default messaging capability may create and access only their own tables. However, the capability to read or write a database can be shared from one process to another, enabling extensibility. Of course, processes may use other storage options designed in userspace or as runtime extensions.

4.8 Runtime Extensions

Wasm is an excellent compilation target for processes. Processes are naturally sandboxed and cross-platform. However, there are also costs associated with Wasm. For example, not all libraries can be compiled to Wasm and hardware support for accelerators like GPUs is still bleeding edge. Extensions supplement and compliment Hyperware processes, removing these constraints, while maintaining the advantages associated with kernel-provided services, e.g., the request/response system.

Extensions are simply WebSocket clients, written in any language, that run natively alongside Hyperware OS and connect to a paired process. The paired process serves as the interface between the extension and the rest of the Hyperware system.

Extensions can be written in any language and can use any library, since an extension is just a native program that can connect to Hyperware as a WebSocket client and that implements a certain protocol.

The downside of extensions is that they are not as easy for users to install and use. Since they are native, rather than Wasm, they will not run on arbitrary systems. They are also not as easy to distribute as packages. Therefore only sophisticated users should be expected to run extensions, since they will either need to compile them themselves or set up and maintain an additional program running next to Hyperware.

4.9 Backwards Compatibility

No backwards-incompatible change will be allowed in a subsequent version. The surface area presented by the OS for the purpose of backwards-compatibility is defined as:

- The networking protocol
- The request-response API for each runtime module listed in Figure 3
- hyperware.wit and the kernel-level implementation of the host functions
- The on-disk footprint of runtime modules that use disk, along with the encrypted keyfile used by the OS to store the networking key and JSON Web Token used for authenticating of node-served frontends.

A number of userspace packages included in the reference distribution must also be backwards-compatible in practice due to the inconvenience created by breaking changes. This includes (but is not limited to) `app-store`, `hns-indexer`, `homepage`, and `terminal`.

5 Package Manager

Like HNS, the Hyperware Package Manager is a protocol deployed on Hypermap. It is another protocol critical to the operation of Hyperware OS. As described in Section 4, the userspace presented by the OS is comprised of processes, which are bundled into packages. There is no kernel-level method for managing the packages installed in a node. Rather, userspace programs with the required capabilities must save packages in the virtual filesystem and prompt the kernel to start running certain processes.

If a process has the necessary capabilities, it may create requests to and receive responses from `kernel:distro:sys` like any other process. The kernel specifies a request type that includes commands relevant to managing processes. Programs that wish to "install" and "uninstall" processes merely submit these requests. These programs must also have capabilities to access the virtual filesystem, such that they can create new top-level directories formatted in such a way that the kernel can access compiled `.wasm` files that contain a single process.

By convention, packages are stored in a `.zip` file with the full name of the package `<package-name>:<publisher-node-id>`, e.g., for a chat app `chat` published by `template.os`, the full name of the package is `chat:template.os`. The top level of the zipped directory contains a `pkg` directory and optionally a directory for the source code of each process defined in the `pkg` directory. The `pkg` directory defines processes by:

1. Containing a `.wasm` file, the name of which matches the process name
2. Optionally declaring the process in a file named `manifest.json`, which defines the processes in the package that should be run upon installation and when a node with this package installed is first booted.

`pkg` may also contain a `scripts.json` file which defines a list of processes that can be run as scripts. Scripts are merely processes that, by convention, can be executed from the system terminal, run for some period of time, and, before exiting, optionally return a final response, which the terminal may print.

It is important to note that all of the above logic exists outside of the kernel and runtime. A package's directory, metadata, and manifest are all interpreted by userspace code and boiled down to a series of kernel commands including `InitializeProcess`, `RunProcess`, and `KillProcess`. Of course, Hyperware OS would not be very useful without this logic, and so Hyperdrive comes with a combination app store and package manager called `app-store:sys`. Note that the publisher name, `sys`, is not a node identity. The publisher value in a package name is not enforced by the kernel. It is accepted uncritically, and it is again the responsibility of the userspace package manager to assert a valid publisher if desired.

5.1 Specification

The userspace app store/package manager uses an onchain protocol running on Hypermap to enable app discoverability and ranking. The definition of a package (interchangeably called an "app") in this protocol is any Hypermap entry that has both of the following sub-entries:

`~metadata-hash`, `~metadata-uri`.

The publisher name of a package is the parent-parent entry. The package name is the last path item in the parent entry.

A `~metadata-hash` entry must contain 32 big-endian bytes corresponding to a SHA-256 hash of the `metadata.json` file used to install a package.

A `metadata-uri` entry must contain a UTF-8 string: a Uniform Resource Identifier (URI) indicating where the metadata file can be found (which, when hashed, matches `~metadata-hash`).

```
os
    foo
        ~ip
        ~ws-port
        ~net-key
        foos-app
            ~metadata-hash
            ~metadata-uri
hypr
    bar
        baz
            ~metadata-hash
            ~metadata-uri
        bam
            ~net-key
            ~routers
            ~metadata-hash
            ~metadata-uri
            boozle
                ~metadata-hash
                ~metadata-uri
```

Fig. 10. Example Hypermap with multiple packages present.

In Fig. 10 there are 4 packages present: `foos-app:foo.os`, `baz:bar.hypr`, `bam:bar.hypr`, and `boozle:bam:bar.hypr`. Note that the parent path from a valid package sub-entry contains the entire package name including publisher. Note also that a publisher does not need to be a valid node identity as defined in the HNS protocol, though in practice it likely will be. A single publisher providing multiple packages can do so by minting sub-entries corresponding to

those packages' names. Lastly, a package may also itself be a node identity as demonstrated in `bam:bar.hypr`. This is a good example of how protocols in Hypermap's global namespace interact and overlap with one another.

5.2 Package Metadata

The value of a package's `~metadata-uri` must be some kind of resource serving `metadata.json`, a file that must hash to `~metadata-hash`. If these requirements are met, a user may use `metadata.json` to gather information about a package.

```
{
    "name": "template",
    "description": "a description of the package",
    "image": "a URL to an image file",
    "properties": {
        "package_name": "template",
        "current_version": "0.1.0",
        "publisher": "template.os",
        "mirrors": [
            "template.os",
            "mirror-node-1.os",
            "mirror-node-2.os",
            "https://my-site.com/my-package.zip"
        ],
        "code_hashes": {
            "0.1.0": "abc"
        },
        "wit_version": 1,
        "dependencies": []
    },
    "external_url": "a URL to a project website",
    "animation_url": "a URL to an animation file"
}
```

Fig. 11. A metadata file.

The structure of `metadata.json` is designed to match that of the ERC-721 metadata spec such that packages can be ERC-721 NFTs.

The `mirrors` field is an array of strings that should either be HNS node identities or URIs that resolve to the zipped package. Mirror nodes must configure themselves to host a package using their App Store program.

Note that the top-level `name` field and `package_name` in `properties` need not match. The former may be a descritive user-facing name while the latter must match the actual package name to be used by the OS.

All fields are required but may be left empty other than `package_name` and `publisher`, which are required to have values.

If `current_version`, `code_hashes`, or `mirrors` are left empty, users will likely be unable to download the package, because a downloaded package is verified by hashing the zipped file and comparing it to the desired version's entry in `code_hashes`.

5.3 Package Manifest

In the `pkg` directory of a package, a developer may write a manifest to programmatically define how a package should be installed and save it as `manifest.json`. If this file is not present, the package will not be installable. The manifest is a JSON array where each element is a description of a process that must be instantiated upon install and subsequent boots of the node.

```
[
    {
        "process_name": "chess",
        "process_wasm_path": "/chess.wasm",
        "on_exit": "Restart",
        "request_networking": true,
        "request_capabilities": [
            "homepage:homepage:sys",
            "http-server:distro:sys",
            "net:distro:sys",
            "vfs:distro:sys"
        ],
        "grant_capabilities": [
            "http-server:distro:sys"
        ],
        "public": true
    }
]
```

Fig. 12. A manifest file.

All fields are required.

A package manifest is interpreted in userspace by a program such as (but not limited to) the default package manager in order to instantiate any and all processes within a package that are intended by the developer to start upon package install. The manifest contains an array of objects, each of which corresponds to a process in the package.

`process_name` sets the name of the process and `process_wasm_path` allows the developer to specify the path to the WebAssembly binary file for the process, relative to the `pkg` directory.

The `on_exit` field sets the behavior of the process when it exits. There are three possible behaviors:

1. `"None"` - The process is not restarted and nothing happens.
2. `"Restart"` - The process is restarted immediately.
3. `"Requests"` - The process is not restarted, and a list of requests set by the process are fired off. These requests have the `source` and `capabilities` of the exiting process.

Documentation and examples of this behavior, along with some subtleties regarding process crashes, can be found in the Hyperware Book[27].

The `request_networking`, `request_capabilities`, `grant_capabilities`, and `public` fields control the process's networking and capabilities, and whether or not the process should be publicly visible. `request_networking` and `public` are booleans that set, respectively, whether the process may communicate with other nodes and whether the process may be communicated with by other processes *whether or not they have a messaging capability object for the process*. The messaging capability object refers to the kernel's capabilities protocol described in Section 4.4.

Finally, `request_capabilities` and `grant_capabilities` are arrays of capability objects serialized in JSON that the process being installed *expects to receive and grant*. The userspace program that interprets a manifest must itself own a capability in order to honor the capabilities in the manifest. Manifests should only request capabilities that are necessary for program execution. In the case of the package manager, a user will be notified and expected to manually approve the capabilities given to a newly installed package's processes. Granted capabilities are generated from the process being instantiated. The shorthand version of a kernel-mediated messaging capability is simply the string version of the `process-id` for which messaging is being requested or granted, seen in Fig. 12.

5.4 Hyperdrive App: App Store

As noted, Hyperdrive comes with the App Store. This package is named `app-store:sys`. The `main` process, `main:app-store:sys`, indexes Hypermap to identify packages onchain, manages the installation of packages using kernel commands, and presents a web UI for a node operator to browse, install, and manage packages (generally labeled as "apps" in the frontend).

The App Store will also use HYPR to enable ranking and filtering of available apps (see discussion of HYPR in Section 8). A common failure mode of distributed networks is that content becomes saturated and global curation is impossible without re-centralization. In the case of an app store, this manifests

[27] https://book.hyperware.ai

as copycat apps, low-effort scams, and a lack of discoverability for even very popular and widely-installed apps. Registering HYPR (see Section 8.1) with namespace entries that are packages published in Hypermap allows users to:

1. Only display apps that have a certain amount of value assigned to them, filtering out discarded tests and spam, and
2. Have a metric to compare apps against one another, allowing one to compare two similarly-named apps and easily see which is more widely adopted in the peer-to-peer network.

This basic filter-and-sort mechanism repeats itself across all protocols in Hypermap. Since each node in the network can directly index and apply algorithms to the namespace, different implementations of software can filter-and-sort using different paramters customized to user preference and use-case.

6 Kit

Kit is the CLI (command-line interface) development toolkit for Hyperware. It provides a variety of tools, including but not limited to:

- `new`: Creates packages from templates
- `build`: Builds packages by compiling processes within
- `start-package`: Installs and starts a built package on a node
- `boot-fake-node`: Runs "fake" temporary development nodes
- `boot-real-node`: Runs a node with a given "home" directory
- `run-tests`: Runs user-defined tests on a network of "fake" nodes
- `publish`: Publish a package on Hypermap

Kit aims to provide developers with every tool required to go from idea to finished project. For example, a developer can use `kit new` to create a template, iteratively develop, build, and test using a combination of an IDE, `kit build`, `kit start-package`, and `kit boot-fake-node`. Finally, use `run-tests` to ensure stability of the project going forward and `publish` to distribute it!

Kit's built-in templates help newcomers learn by giving them working examples so they can quickly begin experimenting. They also demonstrate best-practices.

Kit makes use of Hyperware's HTTP server RPC endpoint to interact with Hyperware. As such, kit is restricted to localhost only. However, `kit connect` allows remote development as long as developers have `ssh` access to the remote machine. `kit connect` showcases a strategy employed throughout `kit`: when tools already exist, make use of them by wrapping them in an easy-to-use way. Here, `connect` uses `ssh` tunneling to send the `kit` request securely to the remote machine. As such, developers can work on remote Hyperware nodes with the same ease that `kit` affords for local development, but security is not compromised for users.

7 Hypermap Advanced

As described in Section 2, Hypermap is an onchain hierarchical key-value store. Keys in the map come in three varieties: "name", "note", and "fact".

Each name entry is an ERC-721 NFT. The entry NFT is minted with an ERC-6551[28] token-bound account. This token-bound account, or TBA, is the only address which is permitted to create sub-entries beneath the associated name-key.

Note and fact entries within Hypermap may store data in their memory slot within the map. Notes are mutable, while facts are immutable once set. They may not have sub-entries.

Combining these properties allows for the creation of advanced permissioning systems within Hypermap, allowing the namespace as a whole to become a tapestry of sub-namespaces, each with unique properties. For example, the process by which HNS integrates with existing onchain identity primitives (described in Section 3.3): an entry is deployed with custom logic (using cross-chain messaging protocols if necessary) that restricts sub-entry creation to only wallets with the matching onchain asset for the sub-entry they are posting a transaction to mint.

7.1 Top-Level Zones

In data structure terminology, Hypermap is a tree, and thus has a "root" node. The root node is not particularly interesting, and sits immutable, with no custom logic, upon protocol deployment. And, given that a tree can be recursively defined, any entry deeper in the tree can be arbitrarily treated as a "root" node for the tree beneath it. This intuition of recursive trees is important for understanding both "how" and "why" entries "govern" the entries beneath them using custom contract logic. Once understood, it's easy to see how any protocol on top of Hypermap can thrive within even a deeply-nested namespace entry and create an entire namespace of its own within.

However, there's a gap that must first be crossed: how do entries immediately beneath the root node come into existence, such that the first layers of custom logic, and potentially infinite subsequent layers, can be applied?

These entries immediately beneath the root node are called Top-Level Zones (TLZs). A fair analogy can be made to Top-Level Domains in the Internet's Domain Name System. And like DNS TLDs, the creation of new TLZs in Hypermap must be permissioned in some way. Uncontrolled proliferation of entries at the top level would lead to name-squatting, a lack of reasonable schelling points, and disincentivize the desired composition of rulesets stacked at various levels of depth within the map: all issues demonstrated by the history of similar namespaces. Unlike DNS, though, TLZs in Hypermap must be distributed in a fair manner and in such a way that overall control of namespace is totally decentralized. Of course, ownership of a TLZ is also permissionless once acquired,

[28] https://eips.ethereum.org/EIPS/eip-6551

leaving no central authority in control of namespace operation. The existence of immutable and relatively ungoverned TLZs also guarantees free expression for protocols within at least those areas of the namespace.

See Section 9.2 for a description of the multi-phase TLZ distribution process.

A given TLZ can be owned by a contract that implements rent logic, requiring regular payments for control over a sub-entry. Or, an owner contract could dynamically re-allocate sub-entries as temporary or permanent rewards for auctions, gameplay, or other onchain activities. We anticipate and welcome these experimental outcomes. A number of TLZs will be transferred to immutable contracts at launch to perform various system roles and integrate with existing onchain identity primitives. The `os` TLZ, for example, will be controlled by a contract that allows any sub-entry to be minted freely, by anyone, and owned forever. To prevent name squatting and generally dilute the value of this "namespace of last resort", minted sub-entries are required to be 9 or more characters long. This character minimum is an example of custom logic that may be implemented at the TBA level for a given name entry.

7.2 Name-Keys

Name-keys determine ownership of entries in the namespace and the ability to both create sub-entries and inscribe data into data-keys. Sub-entries are just name-keys directly beneath the parent name-key, so, unless contract logic in a higher parent entry has disallowed it, the name-key `hello.os` may freely create sub-entries like `sub1.hello.os` and `sub2.hello.os`. Name-keys are tokenized as ERC-721 NFTs and bound to TBAs at counterfactual addresses. The usual properties of NFTs apply to name-keys: they may be transferred, wrapped, and composed with onchain protocols that operate on ERC-721s.

7.3 Data-Keys

Data-keys store content in the Hypermap. There are two flavors: mutable "notes", and immutable "facts".

Entries of this variety **may not** mint sub-entries, hence the prefix: one can use `~my-data.hello.os` to store data while minting `my-data.hello.os` in order to mint sub-entries beneath it, should one desire to do so.

The content of a note or fact is stored as bytes inside the contract map. The owner of the parent name-key is the only address that can set/modify the data stored at that slot. The interpretation of stored bytes is the responsibility of the protocol reading and writing from that entry.

All data is public. Protocols that wish to operate on private data may store hashes at namespace entries, operate offchain within the end-to-end encrypted Hyperware networking protocol, or ideally use a combination of both: Hypermap for public signaling and dispersion of schelling points, peer-to-peer messaging for data exchange.

7.4 Extensibility

Hypermap is designed to be extensible. Protocols such as the Hyperware Name System and the package manager extend Hypermap by interpreting the data stored at certain keys in a particular way. A specific description of how these protocols atop Hypermap specify themselves may be seen in Section 3 and Section 5, respectively.

In the general case, a protocol specifies itself on Hypermap by declaring a set of data-keys that are interpreted a certain way and endow certain properties to their parent name-key. For example, a simple motd ("message of the day") protocol might specify that the bytes stored at any `~motd` key will be interpreted as a UTF-8 string message from the parent key, which could be a node identity in the HNS. If the owner of the key `howdy.hypr` wishes to participate in this protocol, it simply mints the key (mutably, one would hope) `~motd.howdy.hypr` and stores bytes there, perhaps `[68 65 6c 6c 6f 20 77 6f 72 6c 64]`.

Extension of Hypermap is totally permissionless: any protocol can operate on the keys and data stored in the map. Note that if two protocols use the same entry or entries to store data, key owners may be forced to choose between participating in one protocol or the other. If an entry label is already in use by a popular protocol, developers creating a new protocol would be advised to either match the data format in current use for that entry label, or ensure non-overlap by prefixing/postfixing the entry label with a custom value. For example, if the key `howdy.hypr` is the entry-of-interest, and the motd protocol described above is in common use, a different protocol that wishes to use the `~motd` entry label could specify that it instead reads that label from `~my-protocol-motd.howdy.hypr`.

Another strategy for avoiding conflicts is to subdivide the namespace by storing a protocol's data entries at a nested path beneath the relevant entry. A different protocol that wishes to use the `~motd` entry label could specify that it reads that label from `~motd.my-protocol.howdy.hypr` rather than directly below.

7.5 Counterfactual Addresses For Hyperware Smart Accounts

A counterfactual address is a smart contract address that is known before code is deployed into its storage on the blockchain. Ethereum enables counterfactual address creation with the create2 opcode, which deploys code to an address deterministically generated given the contract address calling the opcode, the initialization bytecode for the contract to be created, and a salt. Since each of these factors is known ahead of time for any given entry, every name-key in Hypermap has a corresponding counterfactual address for its smart contract account. This property comes in handy when designing protocols that operate on token-bound accounts. A developer can instruct a user to create a sub-entry for their node identity and have already deposited assets in the wallet that will be created as a result.

7.6 ERC-6551 Token-Bound Accounts

The smart contract accounts that are deployed to these addresses are ERC-6551 compliant token-bound accounts. This enables the ownership of a given node on the network to be managed according to any logic that operates on ERC-721 tokens. Anything and everything associated with the account can be transferred to a new owner.

Token-bound accounts are fully programmable "smart accounts". As a result, they can implement arbitrary logic to govern the assets within. They can also be used to control the sub-entries endowed to them by Hypermap's logic. By default, the token-bound account associated with a name entry will have the sole ability to create sub-entries beneath. This can be modified to expose a public mint function with arbitrary requirements, such as a minimum amount of HYPR registered to the sub-entry. Such logic has already been implemented in various top-level zones to create public namespaces (e.g. `os`).

When a name entry is created, the minter sets an implementation for the token-bound account. A name entry in Hypermap may set its `gene` to a specific token-bound account implementation. This enforces that all subsequently created sub-entries will use the same implementation (and inherit the `gene` as well), overriding the implementation set by the minter.

7.7 Scaling

The scaling properties of a Hypermap instance are limited by the blockchain on which it operates. We do not forsee this being an immediate concern as Ethereum Layer-2s can handle enough transaction throughput to support basic Hypermap usage for a large number of users. However, a Hyperware-centric future will clearly require vastly more scale in terms of Hypermap operations per second–more than any single blockchain can support today. The scaling solution must therefore be horizontal: a network of independent Hypermap instances, each capable of handling a portion of the total state of the namespace. Thankfully this strategy dovetails with the single-chain Hypermap implementation. The Hypermap contract declares a single root node (see Section 7.1). The first instance of Hypermap simply declares the root node to be `0x0`. Subsequent deployments may scale horizontally across multiple blockchains by deploying with a root node set to an existing namespace entry *within* the "main" instance (or even a separate instance, in a nesting pattern).

A subsequent deployment, therefore, has a "pointer" within the primary namespace and all entries are nested under the entry in the primary namespace. The "pointer" entry will likely post immutable facts such as `!chain-id` and `!address` to direct to the subsequent deployment. This strategy will allow the namespace to easily scale across a variety of Ethereum Layer-2 blockchains, and possibly other EVM-compliant blockchains. Note that while the namespace information will work straightforwardly across deployments, the smart account functionality will require more implementation work to work across chain boundaries if at all. Smart account module inheritance may be limited to a single

chain. It is worth noting, however, that cross-chain messaging projects such as LayerZero already support functionality for executing transactions from an ERC-6551 token-bound account in a cross-chain fashion. This means that namespace entries may operate as wallets across supported chains.

7.8 Review

A quick review of Hypermap's architecture:

1. All keys are strings containing exclusively characters 0–9, a–z (lowercase), - (hyphen).
2. A key (also called an entry) may be one of two types, a name-key or a data-key.
3. Every name-key is an ERC-721 NFT with an ERC-6551 token-bound account.
   (a) Name-keys may create sub-entries directly beneath themselves
   (b) Name-keys may inscribe data in data-keys directly beneath themselves.
4. A data-key is controlled by its parent name-key and points to bytes stored in contract memory.
5. Data-keys are either "notes" (mutable) or "facts" (immutable).
6. An owner of a name-key can apply rules to the path structure beneath that key.
7. Various protocols will run on top of Hypermap by inspecting specific name–keys and their data entries, and parsing those entries in various ways.
8. The top level keys are called Top-Level Zones or TLZs.
   (a) TLZ minting will be governed in a decentralized manner.
   (b) Once created, a TLZ may define custom rules for its sub-entries.
   (c) TLZs will produce the tapestry of namespace governance schemes that allow Hypermap to be used for a wide variety of protocols.

All that remains for a full understanding of Hypermap's utility is the role of the HYPR token, which operates in lockstep with the Hypermap namespace.

8 HYPR Token

HYPR is a utility token designed to fill two roles in the Hyperware network: assignment of relative value in the global namespace and namespace/protocol governance.

8.1 Registration

A token holder may choose to *register* HYPR with an entry in Hypermap. To register HYPR, a token holder submits a transaction to the Hypermap registration contract specifying the amount of HYPR to register, the duration of the registration, and the target entry.

The minumum registration duration is four weeks (one "month").

Data-keys (prefixed with `~/!`) cannot be a target for registration. Any other entry is a valid target, from top-level entries to arbitrarily deeply nested entries. The target entry does not need to be owned by the address performing registration.

Registration of HYPR is performed in order to produce a value-weighted onchain directory of nodes, apps, and other content. Every protocol built on Hypermap can automatically benefit from the registration of HYPR to keys to create a listing data structure that can be sorted, filtered, and act as a market for user attention.

Note that multiple addresses can register tokens to the same Hypermap entry at any given time. The amount of HYPR registered to an entry is the sum of all active registrations. More details about registration can be found in the Hyperware Tokenomics document, published separately.

8.2 Discussion

As described in Section 2, Hypermap addresses the discoverability problem in peer-to-peer programming by allowing participants to claim paths and post data to a global hierarchical namespace. However, this mass of bytes is near-useless without a *weighting mechanism* that can be used programmatically or manually to evaluate content for relative value.

Consider the operation of a web search engine. First, content is crawled and indexed. In the indexing process, semantic and relative value is assigned to a given piece of content. These weights are then used during a given search to provide a ranked set of content objects that best match the search query. Hypermap combined with HYPR is not itself a search engine, but it does provide the substrate to operate such mechanisms in a decentralized way. The entries in the map are content and registered HYPR is a weight-primitive that applies a topology to that content.

This substrate offers a significant improvement over its centralized counterpart in that incentives are aligned between users and providers. A "provider" can be considered any party that places entries in the global namespace. Providers near-universally seek to optimize for visibility in a zero-sum competition with other providers. A "user" can be considered any "set of eyes" on the namespace (not necessarily human eyes), which providers compete over. Historically, both providers within and the operator of a centralized directory object have been incentivized to abuse the attention of users. Additionally, operators have been incentivized to unfairly extract from providers, devising schemes such as placing a competitor's entries above a provider unless a special fee is paid. Providers abuse attention in a similar manner by bribing operators to weight their content higher in areas where it's not actually relevant to the user.

These inefficiencies appear unavoidable in the modern web. The Hypermap architecture combined with a single weighting mechanism publicly shared between users and providers presents an alternative in which all parties are forced to compete fairly. Users, too, are empowered to reward entries in the global

namespace. Because any address can register on any namespace entry, the ability of a single provider to spend tokens on their irrelevant or otherwise spam-like entry is generally washed out by the broader ability of users to reward valued content by attaching to it.

Meanwhile, the "operator" role is neutralized. Hyperware's constrained governance mechanism is responsible for maintaining the namespace but has no control over the operation of a namespace entry held by another party or the registration operation. The protocol is naturally incentivized not to interfere with the utility of HYPR as a weighting mechanism: any "thumb on the scale" would be visible onchain and immediately impact the value of the neutral weighting mechanism that is HYPR.

The Hypermap+HYPR substrate does not include a built-in algorithm to execute "search" or any other ranking strategy on its weights and values. There is no single algorithm that will apply to the entire Hypermap. Algorithms will instead be written for specific protocols running on Hypermap. These will have access to HYPR "weights" as one tool in determining quality rankings, and many will also take into account other factors. At the time of this writing, it is impossible to predict the specifics of algorithms that will enter popular use for evaluating protocols on Hypermap.

The onchain primitives described in this paper are remarkably simple. Registering tokens does not require any game theory or MEV protection properties. Creating and mutating namespace entries is also not subject to adversarial conditions, since the ability to do so is only granted to an entry's owner.[29] As a result, we have very little to discuss regarding protocol risk, assuming the protocol is implemented properly[30].

It is possible for algorithms operating on Hypermap to use other weight systems, even including a registration system deployed by someone else that uses an entirely different token. Hypermap algorithms will undoubtedly, in many cases, take factors other than just the amount of HYPR registered into account when ranking and filtering entries. If users, providers, and the operator (the Hyperware protocol) are all incentivized via protocol ownership, HYPR will remain an extremely powerful Schelling point for its intended purpose. Protocols such as HNS and the package manager already integrate with HYPR and future development of protocols at the OS and core distribution level will use HYPR.

[29] That does not mean the same is true for contracts that utilize the protocol. Hypermap and HYPR are designed to be modular—as described at length elsewhere in this paper, much of the utility of the protocol will come from contracts deployed "on top" to manage a given namespace and other such things. These contracts must be designed carefully to avoid failure modes common to onchain protocols. For example, if one deploys a contract to manage a top-level namespace that wishes to allow for anyone to register a new sub-entry as their node ID, and exposes a function to claim any name, it would be trivially easy for someone else to front-run that transaction and "steal" the name. A simple solution is to have the user commit to a hash of their desired name as in ENS name minting.

[30] Audits pending at the time of this writing.

What inspires registering tokens to a namespace entry?

Registration of tokens offers utility to the owner of the attached namespace by enhancing their property's ranking in various algorithms running on the namespace. For this reason, owners of a namespace entry will be naturally incentivized to register their tokens on their own namespace. This will manifest itself differently across different protocols running on Hypermap. In the Hyperware Name System, tokens attached to a node identity enable spam-prevention algorithms, preferential routing algorithms, and many others that may be built in userspace or even directly into the OS. The mere social incentive to connect value to an identity will likely inspire registration. We expect this dynamic to play out not just for HNS entries, but also for a number of other social or social-adjacent protocols that naturally fit into the Hypermap architecture.

8.3 Current and Future Uses

HYPR is currently used by both HNS and the Hyperware package manager protocol to address spam, provide a ranking/sorting system, and assign status to nodes and apps in a global context.

One interesting option available to protocols on Hypermap not demonstrated by the protocols in this paper is the ability to discriminate between top-level zones. It is not required that a protocol apply itself to the entire Hypermap. Instead, a protocol may define itself as only being valid within a single top-level domain, a subset of top-level zones, or even a subset of entries at some arbitrary level of nesting. This may prove to be an ideal way to run protocols in a future where Hypermap is very large and indexing the entire map is more difficult than indexing a subset of it.

9 Hyperware Governance

In Fig. 1, Hyperware is presented as a triangle with Hypermap+HYPR and Hyperware OS above Hyperware Governance. Seated at the base of the protocol, Hyperware's governance mechanism is responsible for:

1. Initially distributing the Hypermap namespace and stewarding it towards full permissionlessness via ossification at the TLZ level.
2. Incentivizing use of the namespace until it becomes self-perpetuating.
3. Voting on proposals to improve Hyperware OS in a backwards-compatible way over time, while ensuring that adoption of the offchain software remains aligned with growth of the onchain protocol.

9.1 Voting

HYPR can be locked, producing voting power as a side effect, and allowing for registration to Hypermap entries as discussed in Section 8.1. Voting power

is related to the number of HYPR tokens locked and the duration they are locked for. Voting power can be delegated. More information about locking and registration can be found in the Hyperware Tokenomics document.

Governance is not an empty role in Hyperware—unlike purely onchain protocols, which often fail to benefit from active governance, Hyperware governance includes ongoing responsibilities over the namespace and protocol.

Decentralized finance protocols generally benefit from maximal immutability: once a stable and useful primitive exists, its value only tends to decay with changes. For this reason, "governance" as applied to purely onchain protocols has historically been somewhat weak. There is no governance necessary if a protocol is truly immutable and permissionless. **Hyperware is not a purely onchain protocol**, however, and its governance must be executed in a decentralized manner for the network to be stable, neutral, and permissionless.

While the governance protocol will be strongly incentivized by builders to remain permanently backwards-compatible (meaning that protocols launched on Hyperware will never be forced to apply an upgrade), additive non-breaking aspects can be integrated into the protocol to keep pace with the fast-moving world of software.

9.2 TLZ Management

The most important role of the governance mechanism is to steward the Hypermap namespace until it can stand on its own.[31]

At some point, ownership of the Hypermap namespace, and in particular the TLZs, will be distributed enough that no single entity could disrupt the operation of the network as a whole by abusing ownership rights. Since each TLZ can be the root of an entire namespace, the theoretical security requirements for continued operation of the network are 1 of N good actors. In practice, though, there should be many hundreds of TLZ owners. Reasonable behavior by owners will be modulated by two things:

1. The ability of sub-entry owners to move to other areas of the namespace.
2. A strong natural preference by users to use sub-entries controlled by immutable smart contracts.

Therefore, once enough TLZs are held by a diversity of immutable smart contracts, and possibly mutable contracts controlled by DAOs, the Hyperware governance mechanism will have succeeded in the first phase of its role. From that point on, core developers will continue to contribute to the development of all open-source components of the software.

[31] Note that each TLZ can create its own form of governance, and we hope to see a diversity of approaches. The Hyperware governance mechanism does not dictate how any portion of the namespace operates.

TLZ Auctions Initial namespace distribution will take the form of auctions, in which the governance mechanism executes a proposal to mint a single TLZ or bundle of TLZs and send them to an onchain auction contract, where the winner takes ownership. Auctions can take many forms, such as English or Dutch, and many smart contract implementations of various auction types exist.[32] TLZ auctions may sell ownership of a namespace entry or merely rent it by either transferring the NFT to the winner or approving the winner to use a smart contract, which owns the actual TLZ NFT, for a given period of time.

By auctioning off and otherwise selling namespace, particularly top-level namespace, the governance mechanism may generate revenue. Revenue earned this way may be directed anywhere, depending on the auction implementation approved in a new TLZ proposal, perhaps as further reward for governance participation or initiatives to further increase the utility of the namespace.

9.3 Progressive Decentralization

Through 3 phases, Hyperware governance will transition from a small team of core developers to a fully decentralized protocol.

In phase 1, a small set of TLZs will be selected for creation. These will be minted directly to contracts, including those already in use such as `os`, `dev`, and `hypr`, those associated with an existing onchain identity primitive such as `eth`, and anything else useful.

During phase 1, more TLZs may be created and distributed. The HYPR registration mechanism will be activated during this time.

In phase 2, voting is enabled, but proposal creation is permissioned. In order to achieve distributed ownership of the namespace, the first proposal will be to approve a list of TLZs to be auctioned off. Auctions will seed a treasury controlled in a trusted role during phase 1 and later delegated to an adminstrator or possibly burned, depending on governance decisions. The exact format of the auction is to-be-determined: it will be onchain with permissionless participation. Auctions during phase 2 will direct proceeds to a treasury funding core development.[33]

Phase 2 will last until the auctions have completed and the operating system is at a point in its development where future changes can be approved or denied via Improvement Proposals approved by voters. This means the operating system must be fully specified such that it can be altered by Proposals and confirmed by Specification votes. Development of the OS to the point of specification may take anywhere from 6 months to one year from the current implementation.

[32] https://a16zcrypto.com/posts/article/how-auction-theory-informs-implementations

[33] The auction contract used in phase 2 will sell *ownership* of a TLZ: the winner will have the asset transferred to their address. Phase 3 allows voters to approve any kind of auction contract, which may include styles of auction that do not transfer ownership, but rather implement some kind of rent mechanism. TLZs can also implement rent mechanisms of all sorts within their namespace, which is one of the many modes of TLZ governance we hope to see.

Phase 3 begins the fully decentralized operation of the protocol. Proposals are activated, which combined with voting completes the governance mechanism. Auctions in phase 3, rather than directing proceeds to a treasury, will direct HYPR tokens to a burn address. Auction participants will compete to burn the largest amount of tokens to win the auction. In a similar sense that EIP-1559 directs gas fees to a burn rather than funding a treasury, auctions burning tokens allow the protocol to maintain neutrality.[34]

The vote-proposal system includes a precise set of actions:

- Approve/Deny new TLZ + auction contract to be used for auction[35]
- Approve/Deny new Hyperware Improvement Proposal (HIP)
- Approve/Deny new Hyperware Specification

Proposals will be shared peer-to-peer in the Hyperware network using a protocol to-be-determined, which will involve a mechanism to filter for meaningful proposals (itself using Hypermap and HYPR for this purpose, naturally).

Since Hyperware Improvement Proposals (HIPs) affect (only) offchain software, they only exist to alter the agreed-upon specification of the operating system, which is written, hashed, and posted onchain. A successful HIP will result in a Specification vote. At the point of a successful specification change, all Hyperware users are expected to run an implementation of the operating system that comports to the new specification. This will act as an effective Schelling point for the network.

9.4 Default-distro App: Governance Portal

Hyperdrive will include a Governance Portal app, alongside the App Store and other utilites. The Governance Portal will serve as a central point where a node identity can create, vote on, and review proposals. Auctions and other onchain activities may also be accessible through the Governance Portal.

9.5 Other Duties

The governance protocol may do other work to increase the utility of the namespace and improve the functionality of the operating system, such as developing more Hypermap protocols like HNS and the App Store. It may also take actions to incentivize use of Hypermap and HYPR together.

One significant possibility for the onchain namespace protocol involves scaling and expansion. The initial deployment will take place on a single blockchain, but both volume of transactions and support for composition with other onchain protocols may inspire expansion to other chains. Horizontal scaling of this kind is achievable by minting a namespace entry (possibly a new TLZ) that points to a smart contract on another blockchain. The indicated deployment of Hypermap

[34] https://eips.ethereum.org/EIPS/eip-1559

[35] Note that the specific implementation of auction can be determined by voters, which decides the auction style.

would use that namespace entry as its "root node" (which is simply `0x0` on the primary deployment). Such decisions will be made by the governance protocol, which will be the owner of the root node.

With the ability to propose and ratify Hyperware Improvement Proposals, governance of Hyperware also has the ability to create community Schelling points around standards in software and data. The emergent behavior of Hyperware users will determine how this ability is used, but since each instance of the operating system is keyed to the decisions of the governance mechanism, one may expect that the standards around community discussion, including where it takes place, could one day be determined by the protocol itself. In traditional centralized software, network effects and switching costs make communities fragile to platform risk or the obsolescence of a key protocol. Hyperware's governance mechanism may be the solution to this fragility, allowing the Hyperware community at large to coordinate at a meta-level beyond individual apps and protocols.

10 A Hyperware Future

In a Hyperware future, the web as we know it today is replaced by a tapestry of permissionless protocols that combine the sovereignty of peer-to-peer with the power of industrial-scale computing. Via runtime extensions and controlled namespaces, today's first-party platforms will transform into modularized protocols where users tap into centrally-operated services à la carte. A new generation of protocols will be built peer-to-peer-first, allowing anyone to act as a provider of powerful services like AI image, text, and video producers or high-speed anonymous networking.

These protocols will not be developed by massive teams or require brigades of dev-ops workers to stay online. A small group of programmers will specify their design, create an implementation, and publish it onchain. They may choose to enable future upgrades by decentralizing the governance of their specification with a DAO and token, or simply allow its permissionless use forever in a final state. Switching between protocols will be a trivially easy process for users. The monopoly of ossified web protocols will be obliterated by Hyperware's user-node architecture, which allows for the operation of "transformer" protocols that enable vampire-attacks on any existing protocol with no technical know-how on the side of the user. Nodes can run anywhere: by a user in their home, in a data center, or on a mobile device. Nodes will execute the Hyperware OS specification, but be virtualized by a diversity of runtimes optimized for different environments.

Mainstream technology will continue to evolve rapidly at both the hardware and software level. New programming languages and new chips will emerge that continue to improve on performance and security. Hyperware is a beneficiary of this innovation, and at the same time, it unleashes the potential of new computing capabilities while maintaining the sovereignty of the end user.

11 Appendix: 3 Ways to Use Hyperware

Hyperware enables "sovereign computing" by allowing anyone to run their own node and communicate directly peer-to-peer using protocols deployed on the network. However, making this sufficiently approachable for non-technical users has been a stumbling block for past instances of similarly decentralized software. We address this issue by supporting three user-level entrypoints into the Hyperware network:

1. Hosted Nodes
2. Desktop App
3. Self-Hosted Nodes

These are presented in order of complexity for the end user, from least technical to most. We expect that most users will operate hosted nodes while developers and power users will self-host nodes. The desktop GUI version of Hyperware will allow anyone with a computer to run a node for free, but have fundamental limitations regarding remote access (such as through a mobile app) and consistent uptime, which will likely be important for some protocols.

These options are presented today, but as Hyperware matures, routes that require more technical investment will open up. This includes running Hyperware OS on a mobile device or distributing a version of Hyperware bundled as a Linux distribution for bare-metal servers.

1. Hosted Nodes: The absolute easiest way to join the Hyperware network is to access a node hosted by a professional service. We have already developed a framework for running such a service and intend to open-source the platform while partnering with ecosystem participants to offer a managed service. A user may access a hosted node via a web browser or a mobile app. Hosts may offer various forms of authentication and payment in exchange for access to the node.

While allowing a host to manage one's node sacrifices some degree of total ownership over one's identity and data, this strategy still offers critical advantages over centralized web services.

Firstly, the user only needs to trust a single entity—the host—rather than a separate entity for each service they use.

Secondly, node hosts are always engaged in a game-theoretic competition with other hosts with users as the benefactors. If a host abuses their power, it is a trivial matter for users to move to another host, because each host offers the same fundamental functionality, defined by the Hyperware OS specification. Even offering the ability to import and export an existing node is a feature enforced by this competition, in the sense that users will undoubtedly choose a host that offers this ability over one that does not. Game theory also ensures that hosts will be forced to compete on cost, driving the price of hosting to the minimum premium over raw compute resources.

In fact, historical precedent indicates that hosting for such a service will almost certainly be free. Why? Developers of various value-accretive protocols

will offer hosting as a loss-leader to attract users. In Hyperware's case, hosts can easily offer a node with their software pre-installed, while users can onboard for free and later migrate their identity to a new host or a fully self-hosted setup.

2. Desktop App: As a second entrypoint into the Hyperware network, we maintain a desktop application for easy install and execution like that of a regular program. All that it does is run a node and serve its web frontend as a standalone application. This is a simple way for users to run a node on their computer without needing to interact with the command line.

The ability to package Hyperware as a traditional app was a design goal from the beginning of the project and inspired features such as indirect nodes described in Section 3.1. Users do not need to perform any advanced system configuration to robustly run a node from their laptop or desktop computer.

Nodes run in this fashion will inherently lack support for remote access and regular uptime. Since using Hyperware-powered mobile apps will be a core user experience, most long-term users will be driven to either find a hosting solution or self-host. The same holds true for running protocols that require a node to regularly perform actions or receive messages on the network: the uptime characteristics of a desktop app are not appropriate for such tasks. However, the desktop app will be ideal for testing out Hyperware as a casual user.

3. Self-Host a Node: Running a node on a home server or VPS instance offers the most control over one's identity and data. It is also a somewhat technical process that involves navigating a command line interface and performing system administration tasks like installing a web server, managing a firewall, and configuring DNS. Only a small fraction of users will ever choose to self-host, but for developers and power users (who, for example, may want to provide services like RPC access or compute resources to other nodes) it will always be the best option. Self-hosting Hyperware is similar in complexity to operating a full node for a blockchain and has the same uptime considerations. However, the Hyperware network does not have a network-wide consensus mechanism, so the importance of distributing nodes across many different entities is significantly reduced. It is safe to have the vast majority of nodes distributed across a few hosting providers, with a relatively small number of self-hosted nodes. Self-hosting in the CLI and desktop app form will always be available as a fallback to maintain user sovereignty.