Project Wildland

The Why, What, and How

Joanna Rutkowska, Julian Zawistowski, and Andrzej Regulski

Golem Foundation

April 2020

Version: 1.0

Contents

Overview

This paper provides a technical overview for a new project, which attempts to address the following problems:[1]

1. How to store and manage our data efficiently and reliably?
2. How to effectively navigate our growing collection of information?
3. How to share data with others, using fine-grained access controls and without giving up control to 3rd-parties?
4. How to create an economic framework that ensures project's sustainability?

In order to solve these problems, the project introduces the concept of a self-defined logical data container, which is a fundamental unit of interest in the system (an impatient reader might go directly to 2.2 to get a glimpse of what these containers are).

This paper is divided into three main parts:

1. The **Why**, which discusses *why* we believe that solving the problems outlined above is so important (this part can be skipped with first reading),
2. The **What**, which attempts to paint a high-level picture of *what* we plan to build (a high-level declarative view),
3. The **How**, which sketches the initial technical architecture of the underlying infrastructure needed to "run" what we discussed in the "What" chapter. In this part we also discuss the organizational decomposition of the project.

[1] This work has already been funded by the Golem Foundation and no future fundraising is planned. This paper is thus *not* intended as a material for prospective investors, but rather as an initial technical document, primarily for those who will be building the system with us.

Chapter 1

Why

In this chapter we describe the problem we are attempting to solve and discuss *why* we believe it is of utmost importance to focus on this particular problem rather than dozens of others. We then discuss how the solution might look in the following chapter. Impatient readers might want to skip this highly philosophical chapter and go directly to Chapter 2 ("What") during the first reading.

1.1 Data, services and digital sovereignty

With the progress of our civilization, the amount of information we have accumulated has become horrendous. At the same time, technology has made it very easy for us to access large amounts of information within short periods of time. An increasing amount of work has been undertaken to automate various tasks through technologies, which interact with personal or public data. Today we organize our time with semi-automated, cloud-stored calendars; our information feeds with algorithms run on social media platforms; and our personal tasks with digital assistants like Google Assistant, Amazon's Alexa or Apple's Siri. While those technologies indeed make our lives easier, we have, at the same time, become increasingly dependent on them.

Evidence is gradually mounting that we have already lost our digital sovereignty to the service providers, who offer us all the miracles of today's IT technology while fully controlling[1] our data and applications to process them. This leads to some profound consequences, which range from frequent interferences of politics in Western democracies to making social interactions ever shallower and more focused on irrelevant digital junk.

[1] From the user's perspective, there are different levels of data control. The most basic level of control is the ability to access and copy the data. Next is the ability to modify data. And lastly is to then have control over other parties' access rights, including the ability to process the data and use it within applications without a need of involving a third party. At them moment, many online services do not provide even the most basic level of control.

1.2 What went wrong

The problems mentioned above are well known and have been widely discussed. Yet it is worth summarizing how we ended where we are.

It seems that there are three major reasons of our loss of digital sovereignty:

1. It is technically simpler to provide bundled, closed services.[2] The technical solutions offered by the service providers are oriented towards the best possible user experience and engagement. It turned out that this is easiest to achieve within closed platforms, with full control of the service provider over critical elements constituting the service. That resulted in the lack of control of the user over where their data is stored and how it is processed.

2. A closed service allows providers to monetize users, which in many cases has become the the dominant business model. This also creates an incentive to minimize users' control over their data and prohibits them from switching service providers. Even if a particular platform does not directly spy on its users and does not serve targeted ads, the platform owners are still incentivised to hold a tight grip over users' data (usually by controlling the underlying infrastructure), which is vital to keeping their tools and services competitive.

3. Users can either use the service or opt-out. In other words, users can use the service or stop using it, otherwise their impact over the development of the service they use is non-existent. This is problematic, especially if the service provider monetizes users in a way which is non-transparent and potentially harmful. Alternatives to such services and platforms, based on open protocols are scarce and usually isolated from each other, which makes them a hardly viable solutions to most users.

These three problems result in a broken model of how services we all use in our everyday life are provided to us and give service providers strong incentives to further close their services and compromise users' sovereignty. As a society, we have unknowingly given away governance over what should essentially always have remained a public infrastructure to private and profit-oriented entities, which have been able to build their robust business models by controlling the way we store, access and share our data.

1.3 Availability means sovereignty

But what does digital sovereignty really mean? Usually the term is related to: 1) the control over data, and 2) data privacy. While both of these notions are important components of digital sovereignty, it seems that what is really at stake is the availability of data and digital services. The most obvious situation that illustrates this challenge is the lack of viable alternatives for users who consciously decide to cease using a service because of compromised privacy or no control over their data.

[2] Bundled from the end-user's perspective. Technically services are not bundled, since usually service providers use a combination of in-house and external software and infrastructure.

The issue of availability becomes even more problematic when otherwise credible service providers deny users access to a service. There are multiple examples of groups of users forced to reveal sensitive information under the threat of denial of service, as well as examples of limiting or disabling users' access for reasons discretionary to the service operator without any possibility to appeal against the decision. Depending on the type of the service being denied, this may result in data loss, lack of access to the user's social graph and general impairment of user's digital life.

The availability cannot be secured in the current setup with service providers controlling the infrastructure required to provide the service, which has many features of a public good.

We believe that the first step towards true digital sovereignty is the availability: control of infrastructure we use to store and process our data.

1.4 Monetization and sustainability

Unbundling of services and infrastructure is surely technically possible. The question is: why it has not happened yet, or rather, why we have diverted from full control we enjoyed on our disconnected machines in the nineties to handing over our digital lives to a handful of corporations?

It seems that the root reason is rather the social and economic model of services embraced in the first decade of 21st century. Successful monetization of users has generated revenue streams, which made services even better or even more addictive, making the model sustainable. Open source, community-driven initiatives have been and still are being outcompeted with no chance for mainstream adoption because they are missing one key element: a sustainable economic model.

Surprisingly, also the blockchain space has not yet proposed a sustainable model of financing and bootstrapping open protocols. So far decentralised and/or blockchain projects seem to be either not actually sustainable (however, with substantial initial funding they can continue development for an extended period of time) or to replicate legacy monetization models.

We believe that a proposal of an economic framework for digital sovereignty technology is critical for success of such an initiative.

Chapter 2

What

We want Wildland to enable seamless management of data and information.[1] To achieve this, we introduce what we believe to be a new concept of a **self-defined and self-named logical data container**.

Below we are introducing Wildland containers and describe what we envision these containers to be and how they can be used and shared, both by the user, as well as from the technical perspective. In the next chapter ("How") we discuss different possible technical infrastructures to "run" the containers on and also discuss the relationship between the various parts of the project's infrastructure and applications.

2.1 Evolution of data management paradigms

The ability to access relevant data quickly and reliably is of vital importance.

In the pre-cloud era, we were forced to remember on which of our devices (back then called "machines") and on which of our many hard drives or other media the specific information we need had been stored. Not to mention all the overhead required to sync the data between different machines and disks.

The cloud promised to liberate us from all this hassle. And in many ways it has succeeded. This is one of the reasons why most of us embraced cloud-based storage for most of our data.

Yet, the cloud has forced us into a new paradigm: the service-focused - or more specifically user account-focused - paradigm. Today we think: "my photos are on Facebook", "my work

[1] We understand *data* as any form of stored *information*, typically encoded in various formally-defined formats and stored in files or transmitted over network. Information is a more generic term, not necessarily related to computers. Data is just one way of storing and distributing information. Folk legends, gossips and some forms of visual art might be another. *Knowledge* is *understanding* of information and is, thus, associated with a particular mind, not just with mere information. A mind might be given superb access to information on some topic, yet might not be able to gain any knowledge from it. Ability to make practical use of knowledge might be called *wisdom*.

spreadsheets are on Google Drive", "my (personal) calendar is bound to my Apple ID (my work calendar is on Google, of course)".

This new paradigm has quickly led to several unhealthy developments: from vendor lock-ins to generally giving up control over our data and, consequently, large parts of our digital lives. The 3rd-party companies, whose interests are often not aligned with that of their customers, can not only passively sniff various private data about our personal lives, but also actively moderate the view of the world for us. Last but not least, they can cut us off (either intentionally, or not) from access to our accounts at any time. This creates fear in some of us that large parts of our digital life might suddenly get wiped out. Such a situation might happen because an individual is judged as no longer satisfying the service agreement, or perhaps because of the service provider deciding to end operations in the particular jurisdiction. Even a technical failure or an attack on the centralized infrastructure of the provider could disrupt a person's digital life.

With Wildland, we would like to change this paradigm from service-oriented to data-oriented.

2.2 Wildland data containers

To allow us to change the paradigm from service-focus to data-focused, we introduce what we call a **data container**.

From the user's point of view, containers are like directories (or git repositories). In fact, the user can easily turn any of their existing directory into a Wildland container by creating a Wildland manifest file, like the one below:[2]

[2] In practice users would use some tool to create manifest files for them. This is similar to how one can turn any existing directory into a git repository by initializing it with `git init` command, which in turn creates a manifest and metadata directory called `.git/` within the target directory. Other users would only create and interact with git repositories using graphical tools, such as GitHub Web interface, which underneath would still end up creating these additional metadata files or directories next to the user files.

```yaml
paths:
  - /Photos/Missions/Solaris/heli-trip
  - /Photos/Nature/Oceans/Solaris/heli-trip
  - /Psychology/UberMinds/Solaris/heli-photos
  - /Work/Missions/Solaris/field-study-01
  - /Fun/Heli/Solaris/low-level-ocean-manouvers
  - /Timeline/years/2960s/2961/02/31/trip
  - /Places/Planets/Solaris/Ocean/trip-01
  # The following UUID is by convention only and might be unique (or not)
  # only within the namespace of the particular signer of this container!
  - /.uuid/A67D6A3E-AF50-49C8-82FF-3DE7A49B1765
signer: &admin 0x749e49279f59d9e03a5e31c7ce1051568846b024

# The Wildland Infrastructure (see chapter 3) will try to setup things in order
# to obey the following constrains:
constrains:
  access:
    - user: *admin
      rights:
        - full
    - user: 0x561629df72c1f89bd13280358d853595c3aa5dd6:\
          /users/emails/snaut@solaris.missions
      rights:
        - read
    - user: 0xe2ed22ebcfb898baab9c8716e996e8684b39af75:\
          /directories/SolarisStationRootDirectory:\
          /visitors/Rheya
      rights:
        - read
        - append

  storage:
    # Store on Write-Once Read-Many -type of storage
    type: WORM

backends:
  # listing of storage manifests for this container:
  storage:
    # specific storage manifest hosted on my NAS (but might point to some other backend):
    - webdav://mynas.mydomain.org/heli-trip-storage-manifest-01.yaml

content:
  - path: IMG_10001.jpeg
    type: file
    hash: 749494d6de6ce1d80ca9a2b507b3e1ba7609c680
  - path: IMG_10002.jpeg
    # ...
```

From now on, that (physical) directory, wherever it is located – on the user's local filesystem, home NAS, some cloud storage service - starts its life as a Wildland container. It can now be accessed, shared, and generally be taken care of by the various Wildland tools and apps which we discuss in the the reminder of this paper.

To facilitate unsticking of containers from infrastructure, services, and user accounts, Wildland tries to move as much meta-information about the container into the container itself. This includes the various properties (discussed below), the signer and access rights, and even naming of the container as depicted on the listing above.

2.3 Self-Naming and Multi-Categorization

A container can be identified by specifying its `signer` public key (typically coinciding with the owner/user who created the container), followed by giving one of its *self-defined* `paths` (or names), i.e.:

```
<pubkey>:/<path>
```

The rationale for this scheme is that it is the `signer` (see below), who manages and signs the container's manifest, and has the ultimately say on how the structure of the naming hierarchy for this key should look like.

The container from the example above could thus be referenced via e.g.:[3]

```
0x749e..6b024:/Photos/Missions/Solaris/heli-trip

# Direct access through the client's default directory ('@default'):
# (Directories are discussed later on in the text)
@default:/Photos/Missions/Solaris/heli-trip

# Direct accesses through the client's own key ('@me'):
@me:/Psychology/UberMinds/Solaris/heli-photos
@me:/Work/Missions/Solaris/field-study-01

# This UUID-based addressing is entirely based on convention,
# as there is not way to ensure UUID uniqueness across different
# key signers:
@me:/.uuid/A67D6A3E-AF50-49C8-82FF-3DE7A49B1765
```

This multi-categorization scheme ("multicat" for short, after [10]) offers many advantages in managing of our data:

[3] By convention, a special path `/@uuid/` will let access a container by its UUID. But note that Wildland does *not* enforce UUID to be actually unique crossing domains of different `signer` identities – indeed a malicious user can always generate duplicates of some other containers. For example: `0x749e4...24:/@uuid/A67D6A3E-AF50-49C8-82FF-3DE7A49B1765`.

1. Lack of need to decide on *the one and only* categorization hierarchy scheme is liberating and enables easier and quicker access to the information we need,

2. The scheme is helpful in escaping the service-oriented paradigm – no longer the user needs to bother themself on which service a given information is stored (Dropbox?, Google Drive?, Facebook?, Flicker?), but instead can think about the information addressing in more semantic-focused way.

```
$ cat multicats.ascii

     /^-^\     /^-^\   /^--^\
     \___/     \___/   \____/
     /   \     /   \  /      \
    |     |   |     ||        |
     \_ _/     \_ _/  \__  __/
   ####\\#######//#######\ \####
        \\     //         \ \
         ||   ((        |  | |
          \\   ))       \\_//
            \_//
```

2.3.1 Dealing with conflicting paths

What happens if two or more containers claim to have the same name (i.e. at least one of their defined paths coincide)?

Wildland, assuming decentralization on nearly every level, does not attempt to enforce any kind of anti-conflicting policy.[4]

Instead, Wildland accepts that there might be containers with conflicting paths *within* any single `signer` identity, but ensures there won't be conflicts on the global level. This is the very reason why each container in Wildland, even when referenced by its UUID, is always expected to be preceded by the `signer` pubkey. Assuming the public key cryptography used is strong enough (an implementation detail we are not planning to discuss in this paper), it should be practically impossible for having a collision on the global namespace level. And this is all we need.

Additionally, we assume that it might be a helpful convention (but neither operational nor security-critical) if the client software implements some *default policy* for resolving situations when a single user (signer) ends up having more than one container named with the same path.[5] A simple policy could e.g. always return the most recent container, or a randomly selected one, or all matching. This is up to the implementation of the client.

[4] Not even blockchain-based, as this would mean centralization on some higher level of abstraction still – namely the use of some logical *singleton*, i.e. the blockchain.

[5] Such a situation might occur e.g. if a user created the manifest file by hand or due to some implementation bug in the particular client software.

2.4 Key manifests, container ownership and administration

The manifest file is signed with a key, whose public portion is specified in the `signer` field of the manifest. By definition, the container's signer is the entity, which is able to generate signatures corresponding to the public key specified in that field.

Naturally, the signer can always grant themself full access to the container's content. But doing so requires modification and (re-)signing of the manifest, thus leaving an audit-able trace.[6]

In most situations we expect the signer to be the same entity as the actual *owner* of the container and its content. However, we can easily imagine other situations, such as when a company administrator might be a signer for containers used by other employees in that organization, but not be granted any access rights to the actual content of the containers.[7]

Wildland expects that there is an associated container for each signer identity, called **key manifest**. Below is an example of such a key-defining container for the signer of the container defined in the previous example above:

```
paths:
  # all these paths are by convention and up to directory admin to verify:
  - "/users/names/Kris Kelvin"
  - "/users/emails/krisk@lem.universe"
signer: 0x74663d50d18411a112b55dfe3d73ed964cca506a
pubkey: 0x74663d50d18411a112b55dfe3d73ed964cca506a
validity:
  not after: 2999-01-01
```

We can also imagine situations where one entity is using multiple keys - for example, an entity representing group of people or simply one person using different keys on different devices. In those cases, `pubkey` field in the key manifest will contain multiple entries (which can also points at other key manifests, possibly naming also multiple keys). In any case, the key manifest is signed with a key named in the `signer` field. Here is an example of a key-manifest naming multiple pubkeys:

[6] Assuming the underlying infrastructure has been configured to store each version of the manifest for later audits.

[7] We currently envision each container to always have exactly one signer entity. But it might be possible to extend this scheme in the future to allow each manifest to specify several (N) signer identities in order to allow interesting share-of-duty M-of-N access control in some organizations.

```yaml
paths:
  # all these paths are by convention and up to directory admin to verify:
  - "/users/names/Snaut"
  - "/users/names/Snow"
  - "/users/emails/snaut@solaris.missions"
signer: 0x32a14d70e5be862b8a15f3f5df7a7b46a906a598
pubkey:
  - 0x32a14d70e5be862b8a15f3f5df7a7b46a906a598
  - 0x26018c83004b1ca16b9a64f535b121731e3051e9
validity:
  not after: 2999-01-01
```

Maintaining multiple pubkeys in every (copy of) key manifest may be a highly non-trivial task since the owner may not directly control all those copies (like directories). For this reason, a key manifest can name a specific storage(s) that should be checked for canonical, up to date version of the key manifest. Since both key manifests are describing the same signer, the client can easily verify authenticity of the other key manifest because both are signed with the same key.

This storage can be also used to store manifests of other containers of the same signer, easing discovery of them (see chapter 2.8).

Here is another example of a key manifest used in further examples:

```yaml
paths:
  # all these paths are by convention and up to directory admin to verify:
  - "/users/Harey Kelvin"
  - "/users/Rheya Kelvin"
  - "/visitors/Rheya"
signer: 0xb6ddcc11f5818361b4ab7fc96ecfa72aa270e421
pubkey:
  - 0xb6ddcc11f5818361b4ab7fc96ecfa72aa270e421
  - 0x490ae7ffceb4edb7b3475ed7bd1278eec2c94084
storage:
  - webdav://mynas.mydomain.org/default-storage-manifest.yaml
validity:
  not after: 2999-01-01
comment: "Am I a real person?"
```

2.5 Container sharing and access control

When one user (i.e. one client instance, corresponding to one particular `signer` pubkey) shares a container to another user (a different `signer` id), a compliant Wildland client should expose (mount) the offered container under the tree rooted in a subdirectory named as the original signer pubkey, i.e. (assume the / of the path below is the root of where all Wildland containers are mounted by the client who received the container):

```
/<offerer_pubkey>/some/path/defined/by/offerer
```

But how does Wildland enforce access control? Naturally, if anyone who could merely *guess* a path to the original container (as discussed below in 2.8) could get automatic access to it, would not be satisfying, as neither user pubkeys nor the containers paths are expected to be confidential.

In general there are could be two kinds of access control:

1. infrastructure-enforced, e.g. storage-backed which denies access to files in question on which the container manifests or its content is stored,
2. cryptography-based access control, which makes the metadata and/or content unreadable (confidentiality) and also invalid in case any unauthorized modifications got introduced (integrity).

As discussed more in the next chapter, it is up to the client – or Wildland infrastructure in general – to find and set up suitable backend infrastructure to host users' containers on. Thus, the specific access control method employed for a particular container would depend on the specifics of the infrastructure it has been deployed on.

For example, if a container was to be backed by an AWS S3 storage service, then the Wildland software could take advantage of the AWS IAM access control mechanisms in addition to the client-side cryptography. But if the same container was to be stored on e.g. an IPFS filesystem, then cryptography would be the only available tool. Users, naturally, would be able to intervene (via client configuration and container manifest's `constrains` section) with regards to what kind of infrastructure is acceptable for hosting of their containers.

2.6 Delegated content

It is likely that users who would receive containers from others will want to be able to rename them and locate them within their own path hierarchies. With the original container (i.e. the one exposed through the `/<offerer_pubkey>` tree) this is, in general, not possible since these containers manifest's are signed by their corresponding owners and not by the receiving user. Thus, they are not modifiable by the receiving user.[8]

To address this usecase, we envision that a user will be able to create a stub-container, whose manifest will say that the actual content is to be taken from another Wildland container, e.g.:

[8] Of course, a software client running on the user-controlled system can do anything it – or the admin of that machine – wants. Yet such an "unauthorized" modification of a container's manifest would not be very useful for the user, as the modified container would not be valid (and thus e.g. could not be shared further) anywhere outside of this particular system, which runs the "non-compliant" client.

```
signer: 0x32a14d70e5be862b8a15f3f5df7a7b46a906a598
paths:
  - /Work/SolarisMission/staff/reports/2961/02/31/kris-report
  - /social/crew/kris/boring_staff_07
  - /.uuid/97C96490-9B67-475C-817C-DC911E8E6167
content:
  - type: remote
    url: wildland:/SolarisMission/RootDir:\
                  /users/Kris Kelvin:\
                  /Work/Missions/Solaris/field-study-01
    # The above Wildland URL means:
    # 1. Resolve first:
    #    "/SolarisMission/RootDir:/users/Kris Kelvin"
    # 2. ...then use as a pubkey/dir for resolving of:
    #    "/Work/Missions/Solaris/field-study-01":
    # More on the nested directories further in the text.
```

2.7 Collaboration and conflict resolution

Collaboration happens when more than one actor is given write-access to the same resource. Collaboration rises a problem of conflict resolution, which occurs after two or more actors have modified the same resource without being aware of simultaneous modifications introduced by other actors.

The simplest solutions to the problem of conflict resolution is to enforce exclusive access semantics, i.e. that at no time more than one actor gets write-access to the same resource. Older software version control systems worked that way. This is also a very inefficient approach.

Modern software, especially ones based on decentralized architecture, such as git [11], takes a different approach.[9] It attempts to detect and resolve potential conflicts after they have occurred. Git is particularly good at resolving many conflicts in a fully automatic way, yet at times it needs to give up and ask the user, what they want. In such a case the problem of deciding on the "right version" is moved outside of the system and offloaded to the persons operating git. Approaches such as those used by git are heavily file-format and application-dependent (in case of git they work well for text-based files representing software code, but they would be of little use to resolve e.g. two conflicting photo file edits).

A different approach is taken by Web applications with centralized logic, such as Web-based office tools (e.g. Google Docs). The client software smartly offloads conflict *avoidance* to the operator, by always interactively presenting the very latest view of the document. The (human) user, is expected to notice that some characters on the screen have changed places, perhaps just a fraction of a second, before typing their own letter. This works well for human-edit-able documents. It also requires centralized logic (even if not necessarily centralized infrastructure).

[9] A truly decentralized software could not enforce the exclusive access even if it wanted to disregard all its inefficiency problems.

As Wildland operates below the application level, and is also decentralized on many levels of the architecture, we think it is not possible for Wildland to implement any generic conflict avoidance or resolution policy.

Instead, Wildland hopes to provide enough building blocks for applications to provide effective domain-specific conflict avoidance & resolution solutions. One such building block are the container hooks for conditional code execution, discussed below in 2.9, together with some more real-life examples in 3.1.2.

2.8 Bottom-up process for identity and container discovery

How do users find each other's containers? How do they obtain each other's pubkeys in order to share containers in a trustworthy way?

Rather than building a top-down infrastructure for key and resource directories, similar to DNS, in Wildland we take an inverse, **bottom-up approach**.

Wildland infrastructure assumes that the owner pubkey can be specified indirectly or recursively. So, for example, an address like this:

```
<id1>:/<path1>:/<path2>
```

will be resolved in the following way:

1. First the `<id1>:/<path1>` will be treated as an address of a Wildland container with an expectation that it will resolve to a valid key manifest, defining a new identity: `<id2>`,
2. Next, Wildland client will attempt to retrieve the container using the just obtained `<id2>` as a pubkey for the container retrieval, i.e.: `<id2>:/<path2>`.

This process could involve more than just 2 steps of indirection and could be summarized with the following simple code in Python:

```
def wl_resolve_recursively (default_root, path):
    pubkey_token,path_token = path.rsplit (':',1)
    wlm_actor = None

    if len(pubkey_token.rsplit (':',1)) > 1:
        # the pubkey part still in the x:y form:
        wlm_actor = wl_resolve_recursively (default_root, pubkey_token)
    else:
        # pubkey now in single token form:
        wlm_actor = wl_resolve_single (default_root, pubkey_token)

    # Finally, use the newly obtained pubkey
    # to resolve the actual container:
    return wl_resolve_single (wlm_actor, path_token)
```

Note the pubkey alone does not tell *how* to get a container of a given signer. In a general case, Wildland assumes this information to be provided by the container owner when sharing the container, in form of storage manifest (see chapter 3.2.2). But in case of a public container (like a directory), Wildland gives ability to point at the right storage directly in a key manifest using the `storage:` key (see chapter 2.4). This allows seamless discovery of public containers, while preserving privacy of non-public ones.

This should allow to build identity and container directories at various levels of organizations. For example, a community might designate a local Wildland infrastructure, identified by some `signer` pubkey which will be disseminated to all the interested members of the community using means outside of the Wildland system (e.g. via writing it down on a chalkboard). Then there could be several such communities deciding to establish an inter-community directory, again in a form of a designated Wildland signer pubkey. Continuing upwards, there could be some organization with ambitions to maintain a planetary directory, yet implemented with traditional centralized cloud infrastructure. And there could be another ambitious project to implement such a global directly with a help of some smart contract, to assure name uniqueness. End users might always decide which one they want to subscribe to. Or might decided to use none of these global ones and continue with just local community ones if this is satisfying for their needs.

Here's an example:

```
# Direct access through the (client's) default directory ('@me'):
[@me:]/Photos/Missions/Solaris/heli-trip

# Get the user key manifest from the Solaris Space Ship,
# which has been configured as a default dir for this client:
[@default:]/SolarisMission/RootDir:/users/Kris Kelvin:\
# ... then resolve the final container address:
      /Photos/Missions/Solaris/heli-trip

# Even more levels of indirection:
lem=0x50726f664b61726b6172616e556e697665727365
$(lem):/Universes/Solaris/rootdir/:\
     /directories/SolarisStationRootDirectory:\
     /users/Kris Kelvin:\
     /Fun/Heli/Solaris/low-level-ocean-manouvers
```

The use of a colon to separate the directory part from the container part might seem superficial, yet the rationale for its use, we believe, comes from privacy. If we didn't use the explicit marker of which part should be passed to which directory, it's tempting to think we could just offload to the remote entity to give us a hint by answering either: "yes, I can still resolve the path further, please send in subsequent path tokens for resolve", or: "I have no more knowledge further down this path, please now contact the resultant directory". This should work, yet, a malicious directory (a higher-level one) might be cheating us that it has all knowledge about all our path fragments, only to trick us into giving in the full path of the resource we want to access. This would be a privacy attack on addressing metadata. We believe the use of explicit ':' in Wildland addresses mitigates this problem.

A directory is just a Wildland container or a collection of Wildland containers. No special Wildland-specific functionality (e.g. a dedicated Wildland service) is required on the backend infrastructure to use it as a directory. We discuss this further in the next chapter.

How does a directory verify uploaded key manifests?

In general this is up to the directory policy. We can imagine a national-level government-operated directory, which will require a citizen ID document in order to confirm a match between the submitted key manifest and the actual protein-based bipod. A different directory might require a proof of ownership (or access to) an email account, a DNS record, or perhaps something else. And a small, invitation-only local-community directory might run entirely on the basis of mutual trust between all the members. Wildland doesn't impose any policies on these matters.

2.9 A touch of movement

In many situations we need various automation to generate, adjust, or otherwise operate on our data and consequently on our containers. One such example would be a container which implements a Website, as discussed later.

A different example is shown below:

```
name: My Daily Press Brief
uuid: A854CF68-E634-47BB-B4A7-99B39BA9B0AD
paths:
  - /Reading/Press/Daily Briefs
  - /Politics/News
  - /Weather/Forecasts/Daily

# Container-defined automation called by Wildland infrastructure:
hooks:
    # define a time-based hook emulating Unix cron:
  - type: crontab
    # run this hook at 06:30 every day (crontab syntax)
    condition: 30 06 * * *
    # ... and at these times issue the following command:
    command: run daily-brief
    # ... using the docker as runtime, defined below
    # by the compose file included in the container:
    runtime:
      type: docker
      options:
        compose-file: automation/coffeebrief.yaml
```

The hook tells the hosting infrastructure to run a script every morning to generate a press brief. The resulting PDF produced by the script is to be stored in this container

In this example the `coffeebrief.yaml` defines a docker image, contains various scripts to gather front pages from various publications and produces a daily brief of PDFs. These PDFs are then stored within this Wildland container.

The resulting content of the above container, as seen by the user, might then look like this:

```
.
|-- wildland.yaml     # the container manifest
|-- automation
|   |-- do-yearly-archiving.sh
|   `-- coffeebrief.yaml
|-- dailybrief-2019-11-05.pdf
|-- dailybrief-2019-11-04.pdf
|-- dailybrief-2019-11-03.pdf
|-- dailybrief-2019-11-02.pdf
`-- dailybrief-2019-11-01.pdf
```

2.10 From files storage to self-hosted Organic Web

So far we've been considering containers mostly as collections of static files. But that, of course, is just one way of seeing things. What is a directory seen through a filesystem-perspective - or filesystem-access protocol - might also be a website when accessed over the HTTP protocol. It might also be a PDF document if the files in the directory got transformed through a PDF-generation script as in the case of the very document you're reading now, which has been produced by applying a chain of tools such as `pandoc` and `pdflatex` to produce the resulting PDF.

Going further, a container might represent not a mere static "homepage", but might become a 1st-citizen Web resource. For example, a user might create a container to represent some upcoming event or discussion forum. Others will be able to leave comments, likes, and share further, similar to what we do today on social-media platforms, like Facebook.

We discuss another example, a collaborative office suite, in the next chapter in 3.1.2.

This suggests that Wildland might become a new kind of Web-like platform, decentralized on many levels of abstraction. Or more precisely: platforms, i.e. plural, as there is nothing to enforce unification of all islands in Wildland under one global village.

Chapter 3

How

In this chapter we sketch our plan to bootstrap Wildland. Following the "bottom-up" paradigm, we want the path for Wildland to allow for smooth introduction of the technology into user workflows without requiring any abrupt changes in software or user habits.

As a mostly software-based technology, Wildland will require both client-side software as well as supporting infrastructure. However, we believe it should be possible to bootstrap Wildland *without* any Wildland-specific infrastructure, such as supporting some custom network protocol. At this stage, we even believe that Wildland might never require any such special infrastructure. We discuss the infrastructure later in this chapter, but first let's start with the client-side software.

3.1 Client-side software

Ultimately Wildland is to be "consumed" or used via client-side software running on client devices, such as user laptops, tablets, phones, home NASes, or some embedded devices.[1]

The Wildland client software (or *client* for short) is essentially concerned with the following two tasks:

1. Tracking of all the containers defined by the user for a particular instance of a client. The user signalizes to the client which containers it should track by pointing it to their manifests, e.g. via a Wildland URI. Once the client software has parsed the container's manifest and attempts to bring the container (i.e. together with its content) into the local device, exposing it to the user somehow, typically as part of the filesystem (as already discussed in the previous chapter, and also further elaborated below). This step involves continuous monitoring of the container's content and syncing it up- and/or down- with the external infrastructure. This mode of operation, from the user's point of view, is similar to how e.g. a Dropbox client works.

[1] Some examplary embedded devices running Wildland clients we can easily come up with: a Raspberry Pi monitoring plant's soil, a vacuum cleaning robot, or a living room big format poster-displaying screen.

2. For the containers which originate on the device, the client is responsible for finding and assigning the proper infrastructure to *take care of them*. At minimum, this involves finding suitable storage infrastructure, but might also require setting up compute nodes for execution of container-defined logic and scripts (which we call *hooks*, see 2.9). We discuss this more further down.

3.1.1 Generic Wildland clients

The most standard way of exposing containers to the user would be through a dedicated filesystem tree, similar to how Dropbox and similar products work. So, for example, on a POSIX-compliant system, the user might be offered access to the containers through a mount point like `~/WildlandForrest/`.[2]

The Wildland-exposed filesystem structure representing the exemplary containers discussed in the previous chapter might look like this (the tree has been manually trimmed for clarity and to fit on one page):

[2] i.e. a mount point is a pseudo-directory, which is a root for a whole another virtual filesystem underneath, attached to it.

```
.
|-- 0x32a14d70e5be862b8a15f3f5df7a7b46a906a598     <--------------.
|   |-- .uuid                                                     |
|   |   `-- 97C96490-9B67-475C-817C-DC911E8E6167     <-------.    |
|   |-- Work                                                 |    |
|   |   `-- SolarisMission                                   |    |
|   |    (...)                                               |    |
|   |       `-- kris-report -> /0x32a14../uuid/97C96490..67 -'    |
|   `-- social                                                    |
|       `-- (...)                                                 |
|-- 0x561629df72c1f89bd13280358d853595c3aa5dd6      <-------.     | <- @default
|   |-- directories                                         |     |
|   |   |-- RootDirectory -> /0x5616..d6/                         |
|   |   `-- SolarisStationRootDirectory -> ../../0x5616..d6/      |
|   `-- users                                                     |
|       |-- emails                                                |
|       |   |-- krisk@lem.universe -> /0x74663d..6a               |
|       |   `-- snaut@solaris.missions -> /0x32a14d..98/          |
|       `-- names                                                 |
|           |-- Snaut -> /0x32a1..98/           ------------------'
|           |-- Snow -> /0x32a14d..98/
|           |-- Harey Kelvin -> /0xb6dd..21/
|           |-- Rheya Kelvin -> /0xb6dd..21/
|           `-- Kris Kelvin -> /0x7466..6a/     ---.
|-- 0x74663d50d18411a112b55dfe3d73ed964cca506a  <--'               <- @me
|   |-- .uuid
|   |   `-- A67D6A3E-AF50-49C8-82FF-3DE7A49B1765  <----------------.
|   |-- Fun                                                       |
|   |   `-- Heli                                                  |
|   |       `-- Solaris                                           |
|   |           `-- low-level-ocean-manouvers -> .uuid/A67D6A3E.. -+
|   |-- Photos                                                    |
|   |   |-- Missions                                              |
|   |   |   `-- Solaris                                           |
|   |   |       `-- heli-trip -> .uuid/A67D6A3E..    --------------'
|   (...)
|-- 0xb6ddcc11f5818361b4ab7fc96ecfa72aa270e421
|-- 0xe2ed22ebcfb898baab9c8716e996e8684b39af75      <--------.
|   `-- directories                                          |         ^
|       |-- InterPlanetaryDirectory -> /0xe2ed22..75/ -------'         |
|       |-- RootDirectory -> /0xe2ed..75/  ------------------'         |
|       `-- SolarisStationRootDirectory -> /0x5616..d6    -------------'
|-- @default -> 0x5616..d6/
`-- @me -> 0x7466..6a/
```

Note how the entries for other users (names after their pubkey fingerprints) are filled *only* with these containers, which they explicitly made public or shared with the user whose view we see

above (in this case it is the user pointed by the symlink `@me`, while the `@default` symlink points to the default directory configured for this user's client).

Yet, exposing of containers can take other forms as well. For example, we could envision a dedicated GUI application, similar to macOS Finder or Windows Explorer, which allows the user to browse through, manage, and potentially take advantage of some Wildland-specific features[3]. Such a GUI application might take a form of a Web application.

Another form would be a Web browser extension, which adds ability for resolving `wildland:` URI schema. This would make it easy for users to share Wildland containers with other users, who might not have installed (or don't want to install) a full-fledged Wildland client.

In some more niche, but security-critical scenarios, we can also think of special security-hardened systems or devices, devised as terminals for secure access and management of Wildland containers. For example, it is tempting to consider security-hardened systems, such as Qubes OS [12] in this context.[4]

3.1.2 Domain-specific apps using Wildland as backend

As previously signalized (see 2.9 and 2.10), containers can host not just collections of files, but also apps. A containers can *be* an app, combining together user's data and automation (scripts, programs) operating on the user's data.

These programs can be used to automatically resolve conflicts (as discussed in 2.7) resulting from concurrent modifications of the resources hosted in the container by more than one user (or one user accessing the same container from multiple devices).

This automation can also be used to host Web-based UI interfaces, such as e.g. an Office-like suite for editing text documents, backed by the files hosted in the container. Packaging such an Office-like suite UI into a container, together with the files it is supposed to operate on, enables flexibility and freedom with regards how and where to host it and how and to whom it should be exposed (using bottom-up created infrastructures, as discussed in 2.8).

Another example of an application made to take advantage of Wildland unique features could be a multi-categorization note-taking app. Such an application could resemble e.g. the excellent Bear app [13][5]. But, in the case of Wildland, all the notes could be stored within a container, with each note being a further container, defining its own multi-path via the Wildland manifest as

[3]Such as setting additional constrains with regards to properties of the underlying infrastructure, e.g. that a container holding backups should be stored on a WORM-type of storage

[4]Qubes OS has been, in fact, promoting thinking about its "qubes" (or VMs) as data-focused containers, rather than code-focused "microservices" for many years. Admittedly, this might not be entirely a complete coincidence. . . ;)

[5]Some of the authors of this paper *love* the Bear app. Its killer feature, we believe, is the structured, multi-categorization tagging system. In Bear, every note is a Markdown document and it defines tags for itself by using a simple Markdown syntax (`#/this/is/my/structured/tag`). Bear can be seen as an excellent UI build around Markdown notes with a single aim: present easy navigation to notes, based entirely on their self-defined tags. Using this flexible and simple tagging system, some of us have been able to build very powerful information management and knowledge-gathering systems. This is a topic for another paper, though.

discussed in 2.3. This should allow to avoid some of the inconveniences of an app like Bear,[6] by decoupling the frontend logic/UI from the backend storage.

Yet another domain-specific example could be a Wildland-based email client. Working as a process in the background, it would monitor the user-defined mailboxes and then expose their content as part of the Wildland namespace. This would allow the user to access (and refer to!) their individual email messages via standard paths, such as: `[@me]/my mail/work/2019/11/@from/golem.foundation/...`. Such decoupling of the actual email service provider and logic (message fetch/send) should allow more meaningful management of email archives, including reliable referencing of important messages from other files, notes, or scripts. Even if the underlying email provider gets changed.

Longer-term wise, we could envision a standalone Wildland container *being* both the user mail server[7] and mailbox (and archive), rather than mere email clients as described in the previous paragraph.

3.2 Infrastructure

At the end of the day, each container must be hosted somewhere. This requires at least storage for the container data, but, for containers which define automation (hooks), also hosting for the actual running code is needed. We discuss now this mapping between Wildland containers and the underlying infrastructure.

3.2.1 Bootstraping Wildland without dedicated infrastructure

We want Wildland to grow on existing infrastructure. To grow "organically", by which we mean no centralized planning or top-down coordination in the process.[8]

We believe it should be possible to bootstrap basic functionality of Wildland without any specialized server software and Wildland-specific dedicated services. This might seem counter-intuitive – after

[6] Bear uses an undocumented internal format as a backend to store all the notes, preventing seamless concurrent use of the notes via other apps or user scripts. This is very limiting for users who have many thousands of notes with sophisticated categorization hierarchies and who would like to be able to run custom scripts for further processing of the notes, e.g. to present them in forms of graphs or automate creation of some notes, using automation often executed outside of user's devices (e.g. in dockers or as lambda scripts running Somewhere Else). While Bear provides a convenient mechanism to export the notes as plain Markdown files, admittedly without losing details or metadata, such workflow is cumbersome because needs to be repeated whenever one wants the automation to get the fresh view of the notes.

[7] i.e. talking SMTP protocol with other mail servers, not necessarily Wildland-based

[8] Some projects are heavily decentralized on implementation-level, yet still require global, centralized coordination on operational level, e.g. with regards to some global parameters of the system. A good example are various consensus-based blockchains, such as the Bitcoin [14] and Ethereum [15] networks. This is not surprising, since the goal of these systems is to implement a singleton data structure in a decentralized way. And the reason they need to implement singletons is because their ultimate goal is to introduce and provide means to manage a *scarce resource*, such as virtual coins. In order to avoid the double-spend inherent to such use-cases, they fall back to using a singleton data structure.

all, the containers must be fetched from *somewhere* (as in physical storage), and we need services to let the user *find* them.

Yet, as we discussed in the previous chapter (see 2.8), all that is needed to discover identities and resolve paths are Wildland containers, which act as directories. Wildland is thus logically self-contained in this respect. Obviously the containers, even for the directories, need to be stored somewhere on some (file) server(s), as already noted above. Yet, Wildland should be able to use the already widely used network protocols and infrastructures for file storage, such as WebDAV, SMB, or sftp. And general services, such as AWS S3 or IPFS.

3.2.2 Storage Manifests for containers

To bind logical containers, as defined by manifests that we saw in the previous chapter, with the actual hosting backend (such as e.g. an HTTP URL, an AWS S3 bucket, a WebDAV file directory, or an IPFS object), Wildland uses – what we call – Storage Manifests. Here is an example of a storage manifest for a container from the earlier example:

```
paths:
  # Note: it is up to the particular client software to decide how it will be
  # maintaining associasions between container menifests and assigned storage
  # manifests. Here we suggest a _convention_ in might use, based on uuid
  # nameing, but remember this is only a convention and there is no way to
  # ensure global uniqueness of these identifiers outside the realam of a single
  # client instance.
  - /uuid/A67D6A3E-AF50-49C8-82FF-3DE7A49B1765/storage_01
storage:
  - type: aws-s3
    credentials:
      access_key: AKIAUDFZL7YD5KNOSGN6
      secret_key: E1A5C4F07C6799F807A0408C45B497A73ADA4FFD
  - url: s3://kriskelvinthemindpicker-013/A67D6A3E/
```

One container can have more than one storage manifest assigned to it. Yet, one storage manifest is for use by one container only. It should describe everything that is required, including access tokens, so the client software could access the container and perhaps also be able to update the container content (subject to the container-defined access rights, of course).

A container manifest should include URLs to its backend manifest(s). This is critical to support seamless migration container between different storage backends - all clients with an access to a container need a way to learn about new storage backend(s), without manual intervention of a user. Access to those storage manifests can be protected by any of infrastructure-enforced access rights or cryptography. Below is an example part of a container manifest naming few storage manifests:

```
backends:
  # listing of storage manifests for this container:
  storage:
    # specific storage manifest hosted on my NAS (but might point to some other backend):
    - webdav://mynas.mydomain.org/heli-trip-storage-manifest-01.yaml
    # ... but maybe more generic would be also acceptable:
    - webdav://mynas.mydomain.org/default-storage-manifest.yaml
    # manifests can be also stored in a distributed networks:
    - ipfs://...
    - ipns://...
    - dat://...
    # or even in other wildland containers (but needs to be careful about cycles...)
    - wildland://...
  # compute manifests (in future versions)
  compute:
    - ...
```

We also envision binding container to special storage backends implemented as... Wildland containers. One should be watchful for potential self-loops, in such scenarios, of course. Yet, the potential advantages (as well as aesthetical joy) seem beneficial. We discuss this more in the Appendix, showing simulated examples of some interesting topologies.

We should note that, even in scenarios where some containers are stored in another Wildland container, it does not require any dedicated server software on the other end of the connection, as it could be entirely taken care by the end user's client software which processes the manifests and resolves the addresses. At the end of the day, it will (hopefully) translate such a Wildland-on-Wildland-defined address to some outside Wildland-realm backend, such as a bucket on AWS S3 perhaps[9]

3.2.3 Supporting automation

When a container defines one or more hooks (see 2.9), the client software additionally needs to also find compute nodes to host the container-defined automation code. This could be e.g. a Kubernetes cluster, a Lambda-like function, or perhaps a decentralized compute network like that by our sister-organization [16].

Similarly like in the case of finding storage for containers, as discussed above, this process also does not require any special, Wildland-specific support from the infrastructure side.

Analogous to Storage Manifests, we define Compute Manifests as bindings between a logical container and specific hosting infrastructure which can run container-defined automation.

Of course, the storage and compute manifests, assigned to one container, do not need to point to the same hosting organization. This should allow for healthy separation of storage of data from

[9] Assuming, naturally, no logical loops have been introduced by users when defining the topology. At least in the near future, we don't expect Wildland to be able to create storage out of thin air ;)

computations on data, which we believe to be important, even if most of the container data are to be end-to-end encrypted when stored.

3.2.4 Finding the right infrastructure to host containers

But how is the actual infrastructure obtained for each and every container?

We initially thought that there should be an ecosystem of container-hosting providers, which would essentially be providing a "caring service" for Wildland containers. So, if a container defines that each morning it needs to execute hooks defined as a docker image to run, they should take care about deploying and running these dockers and connect all the pipes between the container's files and the code hosting infrastructure.

Yet, we quickly realized this might lead to several problems. First, it is unnecessary to expect that storage and automation be handled (even managed) by one external 3rd-party entity. This not only complicates the life of service providers, but also introduces privacy and other risks related to metadata disclosure and general giving away of the control. Second, this offloading of infrastructure-selection to 3rd-parties complicates some scenarios, such as when using user's own NAS as hosting provider[10]. Last but not least, it is generally incompatible with the assumption that Wildland should "grow on existing infrastructure" and moreover, grow in an organic, decentralized way.

We thus next conveyed an idea of - what we called - "vouchers", or small files which describe everything that is needed to access some "quant" of infrastructure. For example a user could get a vouchers entitling them to store 1GB of data for a year (a 1 GB-year "quant") on some infrastructure.

Unfortunately, there does not seem to be a way to avoid a situation in which such vouchers, contrary to our intentions, would begin life as value-carrying tokens[11]. Once vouchers become effective medium-of-exchange tokens, a well known problem of double spending arises. Unfortunately, we cannot easily solve this problem here by "plugging" the system into some (global) blockchain. Not only such a sudden introduction of a centralized data singleton on this *core architecture level* would break the fundamental paradigm on which we want Wildland to be based, but would also imply that Wildland can only work with Wildland-specific hosting infrastructure. This would additionally break, again, the grow-on-existing-infrastructure principle.

Finally, we settled on a less specific solution. We recognize that most of the infrastructure will likely be provided by external service providers, which would expose general infrastructure (such as AWS S3 or IPFS) to Wildland by publishing offers to directories, which we can call "marketplaces". Such directories of offers could be, of course, implemented as Wildland-based directories, similar to how we discussed identity and container discovery in the previous chapter (see 2.8).[12]

[10] The user would need to install special software on the NAS to emulate such a container "care taking" service.

[11] Let's imagine a small local community of neighbors, who decided to buy together a large cluster for storage and compute and decided to share the vouchers, such that each gets their fair share. There would be nothing that could stop some persons from giving away these vouchers to other users, perhaps after obtaining some form of renumeration, outside of the system.

[12] We consider it aesthetically pleasing that in many places Wildland architecture becomes self-referential.

Yet, in general, we leave it to the client software to ultimately implement the code for infrastructure offerings discovery and selection. Needless to say, we plan to provide reference implementation for at least two cases:

1. The no-economy, best-effort case. This would apply e.g. in situation when a single user, or a small community of mutually trusting users, want to make use of the infrastructure. In this case the client software would be simply given a list of infrastructure it can use, together with required access credentials.
2. The blockchain-based economy case, suitable for large-scale deployments.

We discuss the fundamental assumptions and initial sketch of the economy model we envision for this 2nd case in the next section.

3.3 Wildland's economic framework

It is worthwhile to put the economic framework proposed for Wildland in the broader context of online services available today. All of them share the core problem of users' and service providers' goals being permanently misaligned, obviously to the detriment of the former group. Although every commercially viable service has to offer its customers something useful in order to achieve adoption, the ultimate objective of a service provider is to monetize users and their activities e.g. by selling data to third parties or exposing users to advertising content. The user has no control over the underlying service infrastructure and cannot influence the functionality and evolution of a service in any way other than opting out. In the long run, though, it is going to become increasingly hard to participate in many forms of social and economic activities without selling one's independence in exchange for access to various collective mind-augmenting services. The fate of independent minds is not sealed, but if we want alternative, privacy-preserving models of service provision to emerge, we have to create an all-new set of economic incentives.

In order to address these challenges and enable wide-scale deployment of Wildland we need a coherent economic framework, which ensures long-term alignment of goals of all participating agents as well as guarantees a sustainable stream of funding for the platform development.[13] While it is possible for the general concept of Wildland, in particular the best-effort case, to function in various economic setups - or even non-economic/marketplace environments - we believe that only an implementation based on the concepts described below can be considered a viable alternative to the broken incentive models of today's information platforms.

3.3.1 User Defined Organization and Unified Payment System

The following types of agents are expected to interact with each other in Wildland:

- Users, i.e. owners of containers deployed in Wildland. Users are responsible for setting the properties of each container and are charged a fee for various services required to manage the container as specified in the properties.
- Service providers, i.e. individuals or entities offering services related to the management of containers in Wildland, such as storage, resource directories, etc. In order to participate in the marketplace, service providers are required to freeze a specified amount as a stake, which gets partly or even entirely lost (burned) whenever a provider does not fulfil their commitments.
- Builders, i.e. individuals or entities engaged in the development of the core Wildland infrastructure and associated protocols or providers of various other services beneficial to the Wildland ecosystem. They may also be potential recipients of grants/bounties from accumulated build fees.

We propose to structure interactions between agents by introducing two novel concepts: the Unified Payment System (UPS), which, while facilitating payments, provides funding for the

[13] While Wildland will be bootstrapped by Golem Foundation, every project – including decentralized, open source initiatives – needs a long-term development model, including sustainable financing.

platform development; and the User-Defined Organization (UDO), which gives users the power to decide on how the platform is going to evolve. In economic terms, the proposed system can be seen as a way of capturing a part of the value generated by the platform and giving users the authority to decide how the accumulated resources should be used to further enhance the underlying technology. These two mechanisms, combined with the bottom-up, self-organizing nature of Wildland, have the potential to empower users and shift the balance of power away from owners of today's information platforms.

Both UPS and UDO have to be fully open (no arbitrary gatekeeper) and censorship-resistant (no arbitrary censor) for the economic framework to work as intended. To ensure that, we propose to build Wildland's economic framework on top of the Ethereum blockchain. Ethereum provides logic and security necessary to introduce features like conditional payments, stakes, and voting mechanisms in a way which is transparent, open, and censorship resistant.

3.3.2 PoU generation and PoU-based governance

Another key element of the proposed economic framework for Wildland is the mechanism of Proof-of-Usage, which rewards actual usage of Wildland by generating a scarce resource (a digital token – PoU/Proof-of-Usage) whenever a transfer of value takes place between users of Wildland. This makes the generation of that resource costly and, thereby, increases the robustness of the system. The PoU token is then permanently allocated to the party that triggered the transfer of value and gives the owner certain rights and privileges related to the platform/infrastructure, which may or may not be temporarily delegated to other entities. Higher usage implies more power and privileges. Holders of the PoU tokens are entailed to:

- Determine which 'builder' parties are eligible for a build grant/bounty and thus contribute to the development of Wildland. In order to simplify the procedure, the decision process determines the recipient of funds (builder) instead of choosing between specific agendas or projects.
- Participate in Wildland's User-Defined Organization (UDO), which determines e.g. the inner workings of the UPS, the parameters of the PoU token generation process, as well as the internal governance of the UDO itself.
- A very important feature of the PoU token is that it is designed to not become a speculative crypto asset. As it is permanently locked with the user's public key, it cannot be transferred and hence traded. While we envision delegation of voting power along the lines of liquid democracy principles, this delegation can always be revoked by the user.

We believe that the PoU token should not be tradable because decision making power should be related with the usage of the platform and usage of the platform alone. Any other use of the PoU would inevitably lead to the token distribution not reflecting the actual usage of Wildland, creating a variety of governance challenges. Two most obvious include passivity of users (who hold the token as a purely speculative asset) and possibility of economic attack on the protocol using malicious governance decisions and large-scale market operations.

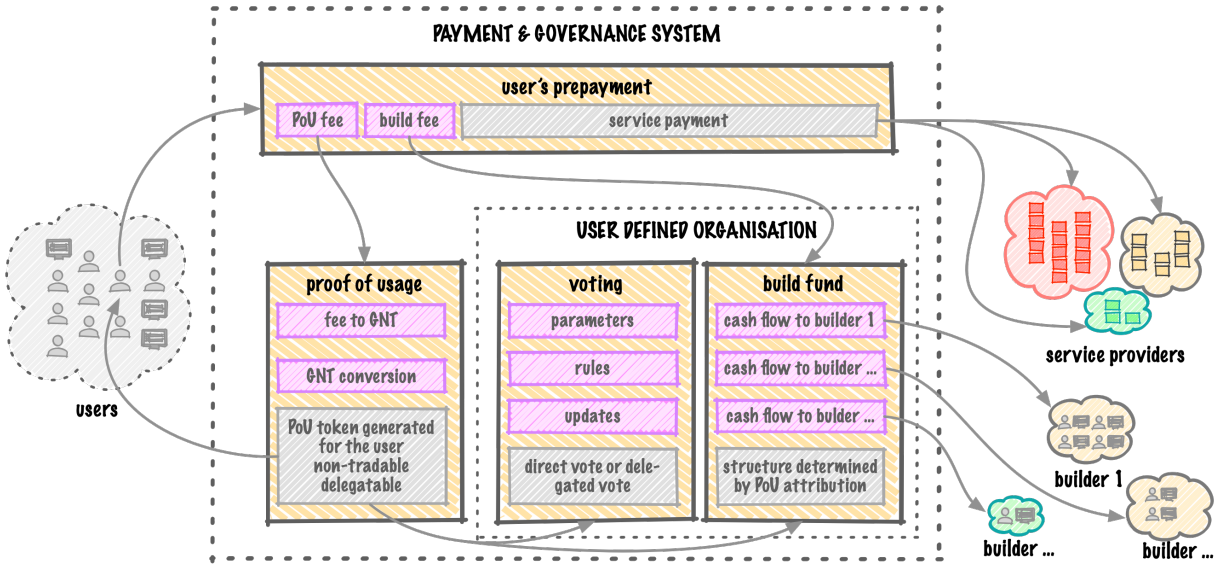

Figure 3.1: Wildland's economic framework

3.3.3 GNT and conversion mechanism

With each generation of the non-speculative PoU token, the corresponding PoU fee gets subtracted from each transaction amount. The PoU fee is then converted into a corresponding amount of GNT (Golem Network Token), which is then permanently removed from the overall supply of GNT. While any economic cost of generating PoU tokens would increase the security of the protocol, the specific association of that cost with a corresponding conversion into GNT results from the initial development of Wildland being financed by Golem Foundation.

GNT will not be directly involved in the governance of Wildland for the reasons described above. Also, we believe that for wider adoption and stability of the staking mechanism it should not be used as a medium of exchange.[14]

3.3.4 Marketplace and offer matching

The matching mechanism guarantees that the container gets connected with a service offer consistent with the properties defined by the user.

- Providers reveal service offers, which they fix by signing.

[14]Wildland transactions will be different to Golem Network, as we envision payments for services over longer periods of time with much lower granularity of payments than currently observed in Golem Network. That makes the problem of token price fluctuations much more significant and creates additional vector of economic attack if GNT is used as medium-of-exchange token. For the same reason, GNT cannot be the sole asset used for staking. At the same time, we are considering uses of GNT beyond the conversion mechanism – most likely in the area of entering UPS as a provider for registering services of UPS driven markets. This is still work in progress, as it depends on details of implementation of markets within UPS.

- Offers signed by service providers can then be published by any service aggregator. Service providers can become aggregators of their own offers, or aggregation can be performed by third-party entities.
- Once the offer has been signed by the service provider, it cannot be altered by the aggregator.
- User's requirements, as specified in the manifest, are matched with available offers by the client application. The client application receives parameters and location of an offer from a service aggregator and then establishes a direct connection with a service provider.

It is worth emphasising that Wildland is ultimately going to become a marketplace where not only users pay providers for storage services, but, as Wildland matures, another category of actors grows in significance, i.e. software vendors and developers. However, we envision their role not as providers of siloed, vertically integrated services that store and manage the entirety of users' data, but rather, as suppliers of various tools that interact with the data that still remains under the control of the user. By enabling that, Wildland has the potential to redefine the role of software to what it once used to be and always should have remained - tools in the hands of users that enable them to perform various tasks, instead of being opaque monetisation schemes built on top users' data. This is also likely to make the user-software-vendor relationship more transparent, built around subscription-based access/licensing arrangements with a user being the payer, not the product.

3.3.5 Payments

Each and every transaction in Wildland is going to be processed by the Unified Payment System (UPS), which is responsible for unifying various building blocks of Wildland at the UX level. We envision the following features of the UPS:

- Being entirely blockchain-based, in particular using Ethereum blockchain. The same blockchain also facilitates the PoU generation process.
- Using as the sole medium of exchange a widely-adopted decentralized stablecoin, not necessarily a fiat-backed one. Providers are able to offer users services that effectively act as gateways to the UPS, thus enabling other payment methods.
- The granularity of settlements between users and service providers is optimized for user-experience. However, users will be expected to make a prepayment in order to ensure security of the UPS.

3.3.6 Why economic incentives matter

As already mentioned in this paper, the entire economic model of today's online services is broken. The list of undesired consequences of the way platforms such as Google or Facebook gather, manage, and disseminate information is not limited to the ever-advancing process of users becoming products, whose attention is being constantly monetized. What's even worse, the lock-in of users' data within entrenched services intertwined with users' lack of control over

the underlying infrastructure gradually make it harder and harder for anyone to enjoy true digital independence.

We, as a society, have found ourselves in this situation not because it is technically impossible to build technologies that preserve freedom and individualism, but because of some profound problems with creating an economically viable and sustainable alternative. This problem is widespread and is not only about storing, managing and processing of our data.

While Wildland technology can work outside of the proposed economic model, we believe that only the combination of both is able to have a long term, sustainable impact on how we organize, share and use our data. Moreover, what we propose as Wildland economy may be a feasible alternative in other areas of software development, as well.

Comparison to Other Work

We can look at Wildland from different angles and at different levels.

It's probably most natural to first consider Wildland as a data storage solution. This domain is already tightly occupied by many established offerings, widely used by millions of users. To name just a few examples: Dropbox, Google Drive, Apple iCloud, Amazon S3, as well as more niche or specialized solutions, such as Tresorit [17], IPFS [18] or Tahoe-LAFS [19]. Yet, unlike these solutions, Wildland doesn't attempt to *provide* a storage service. In fact Wildland doesn't want to provide *any* kind of *service*-based offering. In order to escape the service-oriented paradigm, Wildland, unlike the above-mentioned services, needs to decouple addressing from the actual storage.

This decoupling of addressing from storage brings Wildland closer to projects such as the widely known Internet DNS system [20], as well as the more recent systems such as ENS [21] and upspin.io [22], the latter of which was, in fact, one of the inspirations for us. Unlike these systems, however, Wildland does not require any custom server-side software or specialized infrastructure, such as key or directory services. Wildland architecture allows users to implement DNS-like systems with mere use of *passive* storage. Even more important is the philosophical difference that Wildland's design assumes a bottom-up "organic" architecture, unlike more traditional top-down approaches (whether fully centralized or semi-centralized/federated, like in case of the mentioned projects).[15]

Perhaps it should be more clear now that the storage-offering services and projects mentioned above could be used *together* with Wildland, or more precisely *through* Wildland. In doing so, users gain independence from the caprices of the underlying storage providers because Wildland allows for implementation of seamless migration between storage backends, invisible to the user.

Yet, Wildland tries to be more than just a proxy for storage services. By introducing the concept of a *logical data container* which lets the user to natively use multi-categorization for any kind of data, as well as allowing to specify arbitrary attributes assigned to each container, Wildland tries to become a new kind of data management tool. Combined with the bottom-up, self-organizing nature of the infrastructure, we believe that Wildland could become not a mere data management *tool*, but a data management *platform*, allowing user-defined and user-owned sharing of data. With a bit of optimism, we could hope this would also mean an *information* management platform, which finally brings us closer to the final goal which is: *knowledge* management platform.

[15] While ENS is implemented with a help of a smart contract running on the decentralized Ethereum network, it still can be seen as logically "centralized" at some level of abstraction because it requites access to *the* Ethereum network and needs to follow certain rules of the *the* Ethereum ecosystem.

This might suggest comparison of Wildland with some established information and knowledge management platforms such as the Web "1.0" [23] (understood as the mostly read-only Web) and the Wikipedia [24]. Unlike these platforms, especially the "Web 1.0", understood as a mostly read-only set of resources, and the Wikipedia, Wildland does not have a goal of becoming a repository of unified, *global* knowledge. In fact, we do not believe there always exist universal, global knowledge.

Thus we would like to see Wildland as a *personal* knowledge-management platform. This brings us to comparison with many productivity applications, such as various notes apps (e.g. the excellent Bear [13]). Unlike these products, Wildland is not an *app*. Wildland is an infrastructure, which can be used by various apps (see also 3.1.2). In fact we would love to engage builders of various "mind-augmenting" apps, such as note managers, to use Wildland as an underlying platform.

This use of Wildland as a backend platform for end-user apps, should be made even more attractive by support of "hooks", which we described in 2.9. The support of automation, as defined by the hooks in Wildland containers suggests a comparison with Docker [25], of course. While Docker has been one of the inspirations for us, Wildland differs significantly in philosophy. Unlike Docker, Wildland focuses on *data* not *code*. The code is attached to a data container, rather than the other way round. Stated differently, Docker focuses on packaging and managing code for use in services, while Wildland focuses on packaging and managing data. Docker is an ideal tool for building service-based offerings, while – we hope – Wildland will be more useful for building data-based offerings.

This focus on personalized platform for data management also begs comparison with "Web 2.0" (understood as the user-writable Web), and some of its popular applications, especially social platforms, such as Facebook. The differences should be very clear, of course. First and foremost, Wildland does not want to be a user-account-based service. Second, Wildland's architecture, while allowing to also create and offer more traditional, centralized or federated services, also allows to create self-organized, decentralized offerings for social platforms.

We believe that the mere change of the fundamental paradigm which we have been stressing so many times in this paper (i.e. dropping the service-based user-account-attached approach) should allow us to explore previously uncharted territories and provide lots of useful offerings, which otherwise would be very hard to achieve.

Summary

We started this paper with a discussion of the challenges related to the way we as individuals manage our digital data. Most of the problems seem to arise from a broken system of incentives built on the premise that digital services usually come bundled together with the underlying storage infrastructure.

Next, we described a technology that we believe addressed these challenges. The technology we introduced – Wildland – is based on a fundamental change of the today's paradigm of how we do and consume computing and information technology. We propose a **switch from service-focused to data-oriented paradigm**. Another way of looking at this change of paradigms is replacement of the predominant top-down paradigm, especially in the context of infrastructure building, with a **bottom-up paradigm**. We use an adjective *organic* to describe the key property of this bottom-up paradigm. The connotations to naturally-occurring or growing phenomena such as plants, forests and other forms of wild life is intentional.

In the last part we attempted to sketch a plan of how we could bootstrap this project. An important part of the discussion was a proposal for a new kind of economy which we think could help bootstrap and then sustain Wildland in the long term, without the flaws of currently used models, which we believe easily degenerate into societies-not-serving scenarios, seeing users as "dumb fucks"[16], rather than partners.

[16] https://www.newyorker.com/magazine/2010/09/20/the-face-of-facebook

Acknowledgments

Reviewers

We would like to thank Marek Marczykowski-Górecki, Wojtek Porczyk, Gerben, Eric Grosse, Paweł Peregud (Pepesza), Piotr Janiuk (Viggith), WP, Andrew David Wong and Chris Robinson for reviewing the paper and providing constructive feedback.

Inspirations

Additionally we would like to express our gratitude towards the following projects and persons for being an inspiration for us:

- The Upspin Project [22]
- Bear Application [13]
- Lion Kimbro
- Golem Network [16]
- Ethereum Devcon5: 0x presentation, DAO workshops, P2P workshop, Taylor Monahan's presentation and many others.

References

[1] V. Bush, "As We May Think." The Atlantic, 1945.

[2] D. R. Hofstadter, *Gödel, Escher, Bach: An eternal golden braid*. Basic Books, 1979.

[3] D. Parfit, *Reasons and persons*. Oxford University Press, 1986.

[4] M. Minsky, *The society of mind*. Simon & Schuster, 1986.

[5] The Wachowskis, *Matrix*. 1999.

[6] D. R. Hofstadter, *I am a strange loop*. Basic Books, 2007.

[7] M. Marlinspike, "The ecosystem is moving." [Online]. Available: `https://invidio.us/watch?v=Nj3YFprqAr8`.

[8] "GitHub confirms it has blocked developers in Iran, Syria and Crimea." [Online]. Available: `https://techcrunch.com/2019/07/29/github-ban-sanctioned-countries`.

[9] "Iran bans Instagram - where the president has 2 million followers." [Online]. Available: `https://www.telegraph.co.uk/technology/2019/01/02/iran-bans-instagram-president-has-2-million-`

[10] L. Kimbro, "How to Make a Complete Map of Every Thought You Think." 2003.

[11] "Git." [Online]. Available: `https://git-scm.com`.

[12] "Qubes OS." [Online]. Available: `https://www.qubes-os.org/`.

[13] "Bear (notes app)." [Online]. Available: `https://bear.app/`.

[14] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System.".

[15] "A Next-Generation Smart Contract and Decentralized Application Platform." [Online]. Available: `https://github.com/ethereum/wiki/wiki/White-Paper`.

[16] "Golem Network." [Online]. Available: `https://golem.network/`.

[17] "Tresorit." [Online]. Available: `https://tresorit.com/`.

[18] "IPFS." [Online]. Available: `https://ipfs.io/`.

[19] "Tahoe-LAFS - The Least-Authority File Store." [Online]. Available: `https://tahoe-lafs.org/trac/tahoe-lafs`.

[20] "Domain Names - implementation and specification." 1983.

[21] "Ethereum Name Service." [Online]. Available: `https://ens.domains/`.

[22] "Upspin project." [Online]. Available: `https://upspin.io/`.

[23] T. Berners-Lee, "Information Management: A Proposal." 1990.

[24] "Wikipedia." [Online]. Available: `https://en.wikipedia.org/wiki/Wikipedia`.

[25] "Docker." [Online]. Available: `https://www.docker.com/`.