# Prototypes on IPFS: A Realization of globally distributed reusable Knowledge

## Prototypen im Interplanetary File System - Wiederverwendbares Wissen global verteilt

Dominik Hüser

RWTH Aachen University

dominik.hueser@rwth-aachen.de

**Abstract.** This bachelor thesis combines the Prototype Knowledge Representation with the InterPlanetary File System (IPFS). After introducing both systems, we give an overview of existing Prototype systems and comparable distributed knowledge representation approaches. Then, the thesis will focus on a mapping from the formal Prototype definition by Cochez, Decker and Prud'hommeaux to the specifications of IPFS. The main part of the thesis deals with the implementation of this mapping, including important design decisions and several benchmarks in a controlled system as well as in a real world environment. As a result, we created a system allowing the deployment of immutable and mutable prototype expressions on a globally distributed, peer-to-peer system. Prototype knowledge base's consistency can be checked and its fixpoint can be computed, right after. Strong reuse of prototype expressions is enabled due to IPFS' content based addressing as well as the way of how the presented implementation allows to reuse expressions from other users. Nevertheless, we will observe that the used peer-to-peer system is not suiting the created mapping for prototype knowledge representation on a large scale. Therefore, the thesis is rounded up with an outlook into possible improvements of the underlying peer-to-peer system, as well as alternative mappings.

**Keywords:** Prototypes, InterPlanetary File System, Peer-to-Peer, Distributed Knowledge Representation

# Table of Contents

## 1 Introduction

The World Wide Web is a well-known system to publish, find and access data from all over the world. Especially in the last decade the amount of data on the web increased heavily. With more and more growing amount of information, the need for a mechanized way of dealing with this data became bigger. At a peak in 2001, Berners-Lees publishes his vision of the Semantic Web [5], which describes the idea of a computer readable system right on top of the WWW, and next to the human readable web pages. The vision of structured data should allow software to interpret the data automatically, for example in an automatic rescheduling calendar. Mainly, this should be realized by the Resource Description Framework (RDF), which gives structure and meaning to the data. RDF is a graph containing a set of nodes connected by labeled links. Every node is identified by an IRI, and links are defined as triples $(Subject, Predicate, Object)$ [10]. Using this structure various kind of information can be expressed: For example that the Mars has a satellite Phobos.

(`http://dbpedia.org/resource/Mars`,
`http://dbpedia.org/property/satelliteOf`,
`http://dbpedia.org/resource/Phobos_(moon)`).

This knowledge then can be queried from a SPARQL endpoint. The vision was extended by the Linked Data [6] movement, which is a set of best practices for publishing and connecting structured data. These best practices basically proposed to use URI to name every kind of things. In detail one should use HTTP URI such that other people can look up those names. Recommended are the standards RDF and SPARQL, to link to other things as well. Community driven and supported by the W3C the *Linking Open Data Project* started, which is a first realization of the linked data principles. The idea of the project is to republish open licensed datasets in RDF, and interlink them with each other such that one can crawl and query all data via SPARQL endpoints.

Later, Cochez, Decker and Prud'hommeaux aim to optimize the reuse of structured data, by receding from SPARQL endpoints and downloading whole central authority's RDF graphs. Instead they introduced the prototype knowledge representation [13]: The previous form of vertical top-down sharing is replaced by a horizontal sharing, which means that instances of data should be shared directly between two peers without any central authority. Basically, the idea behind prototypes are expressions which derive from already existing expressions. The expression copies all properties from the base and is able to add or remove properties, as well. This kind of deriving from existing expressions generates strong reuse of data. An example of a prototype expression can be found in fig. 1. There is a prototype expression defining the planet Mars and now planet Earth is derived from Mars by removing its satellites and adding the Moon as a satellite and humans as habitants.


**Fig. 1.** An example of a prototype expression deriving from another one. $*$ removes all same-named properties from the base when we compute the fixpoint. On the right hand, side there is the fixpoint representation of the prototype expression earth.

Recently, saving knowledge in a distributed manner, readily peer-to-peer networks, has been put into the spotlight [6, 8, 22]. This comes along with the semantic web's vision of a decentralized system. But instead of working on top of the WWW infrastructure, this thesis will put the global graph of knowledge on a different infrastructure: A peer-to-peer network which combines several aspects of already existing projects like GitHub and BitTorrent is the InterPlanetary File System. A vision how IPFS and prototypes could be combined has already been presented by Cochez, Decker and me [12]. As you will see later, these two systems fit together pretty well, which is the reason why the realization and implementation is topic of this thesis.

The thesis is structured as follows: In the next section, there is a formal definition of prototypes, followed by a description of the InterPlanetary File System. The section is rounded of with the actual definition of the thesis' task. The third section deals with the initial situation, including related work, followed by a justification why it is important to combine prototypes and IPFS. In the fourth section, we present a formal way of how immutable and mutable prototype expressions and prototype knowledge bases can be mapped to IPFS. The fifth section describes the kernel point of the thesis, the implementation of the mapping which was mentioned before: ProtoIPFS. This section includes the functionalities and architecture of the program, as well as important design decisions which were made during the implementation process. An evaluation and several benchmarks under different conditions can be found, right after. The whole thesis ends with a conclusion including a critical view of the constructed system.

## 2 Two Systems matching well

This section includes definitions and explanations of the two systems which are combined in this thesis: First a formal definition of prototypes and then a description of the InterPlanetary File System. In the end of this section, the reader finds a description of the thesis' task.

### 2.1 Prototypes

We have already defined prototypes briefly, let us now introduce prototypes in a formal way. The following definitions are adopted form the prototype knowledge representation by Cochez, Decker and Prud'hommeaux [13] . First, we define the syntax of prototype including simple change expressions, prototype expressions and prototype knowledge bases. After that, we are going to give each of these constructions a meaning by introducing their semantics.

#### Syntax

**Definition 1 (Prototype Expressions and Language).** *Let $ID$ be the set of absolute IRIs (according to RFC 3987 [18]) with* `proto:P_0` $\notin ID$*. Then we define the prototype language as follows:*

1. `proto:P_0` *($P_\emptyset$) describes the empty prototype.*
2. *Let $p \in ID$ and $r_1, ..., r_m \in ID$, with $m \geq 1$. $(p, \{r_1, ..., r_m\})$ or $(p, *)$ are called* simple change expressions*. $p$ is the simple change expression's ID and $\{r_1, ..., r_m\}$ is the* set of values of a of a Simple Change Expression*.*
3. *Let $id \in ID$ and $base \in ID \cup \{$*`proto:P_0`*$\}$. $add$ and $remove$ are sets of simple change expressions such that each simple change expression's ID occurs at most once in each add or remove set and $*$ does not occur in the add set. Then, $(id, (base, add, remove))$ is a* Prototype Expression *with prototype expression ID $id$. We call the ID of a simple change expression a* property *of its prototype expression.*

*Let* `PROTO` *be the set of all prototype expressions and $PL = (P_\emptyset, ID,$* `PROTO`*) be the* Prototype Language*.*

An example of prototype expressions can be found from fig. 2. This is the formal representation of the introducing prototype expression example in fig. 1.

(`example:mars`, (`proto:P_0`,
{(`example:satellite`, {`example:phobos`, `example:deimos`}),
(`example:type`, {`example:terrestrial`}),
(`example:age`, {`example:4500000000`})}, ∅))

(`example:earth`, (`example:mars`,
{(`example:satellite`, {`example:moon`}),
(`example:habitant`, {`example:human`})},
{(`example:satellite`, ∗)}))

**Fig. 2.** Formal representation of the introducing prototype expression example about the solar system.

**Definition 2 (Prototype Knowledge Base).** *Let $PL = (P_\emptyset, ID, \texttt{PROTO})$ be a Prototype Language and $KB \subseteq \texttt{PROTO}$ be finite. $KB$ is called* consistent *or a* Prototype Knowledge Base *iff the following is satisfied:*

1. *$P_\emptyset \notin KB$*
2. *There are no two prototype expressions in $KB$ which have the same ID.*
3. *For every prototype expression $(id_1, (base, add, remove)) \in KB$ and for each value $id_2$ of the simple change expression's values set in add, there is a prototype expression $(id_2, (base, add, remove)) \in KB$.*
4. *All prototype expressions derive recursively from $P_\emptyset$.*

Since all prototype expressions derive recursively from $P_\emptyset$, there are no cycles in the inheritance links.

Later in this work, we will be deploying prototype knowledge bases on IPFS. This is a shared environment and hence consistency can usually not be guaranteed. Therefore, we will, in that context, call any set of prototype expressions a prototype knolwedge base. We will call this set *consistent* in case it is consistent according to definition 2.

**Semantics** After we have defined the syntax of prototype expressions and prototype knowledge bases we look at their semantics, now. We begin with the definition of Prototype Structures and Prototypes. Note that there is a difference between prototype expressions and prototypes. A prototype is the result of the interpretation of a prototype expression. Right after that, we define how we interpret prototype expressions and prototype knowledge bases, including Herbrand interpretations, simple change expressions, sets of simple change expressions and properties.

**Definition 3 (Prototype and Prototype Structure).** *Let SID be a set of identifiers. A tuple $pv = (p, \{v_1, ..., v_n\})$, with $p, v_i \in SID$ and $i \in \{1, ..., n\}$, is called a* Value-Space *for the* ID-Space *$SID$. A tuple $o = (id, \{pv_1, ..., pv_m\})$, with $id \in SID$, Value-Spaces $pv_j$ and $j \in \{1, ..., n\}$ for the ID-Space $SID$, is called a Prototype for the ID-Space $SID$. A* Prototype-Space *OB is defined as the set of all Prototypes of an ID-Space SID. A* Prototype Structure *$O = (SID, OB, I)$ for a prototype language $PL = (P_\emptyset, ID, \texttt{PROTO})$ consists of an ID-Space $SID$, a Prototype-Space OB for the ID-Space SID and an interpretation mapping function $I : ID \to SID$.*

This mapping is defined now:

**Definition 4 (Herbrand Interpretation).** *Let $O = (SID, OB, I_h)$ be a prototype structure for the prototype language $PL = (P_\emptyset, ID, \texttt{PROTO})$. $I_h$ is called a* Herbrand-Interpretation *if $I_h$ maps every entry of $ID$ to exactly one entry of $SID$.*

**Definition 5 (Values of a Simple Change Expression Set Interpretation).** *Let $KB$ be a prototype knowledge base and $v$ the values of a simple change*

*expression* $(p, v)$*. Then, the interpretation function of* $v$ $I_s(KB, v)$ *maps to a subset of* $SID$*:*

$$SID, \text{ if } v = *$$
$$\{I_h(r_1), ..., I_h(r_n)\}, \text{ if } v = \{r_1, ..., r_n\}.$$

**Definition 6 (Simple Change Expression Interpretation).** *Let* $KB$ *be a prototype knowledge base and a function* $ce = \{(p_1, v_1), (p_2, v_2), ...\}$ *be the set of simple change expressions, with* $p_1, p_2, ... \in ID$*. Let* $W = ID \setminus \{p_1, p_2, ...\}$*. Then, the interpretation of a simple change expression set* $I_c(KB, ce)$ *is defined as follows:*

$$\{(I_h(p_1), I_s(KB, v_1)), (I_h(p_2), I_s(KB, v_2), ...\} \cup \bigcup_{w \in W} \{(I_h(w), \emptyset)\}$$

For a prototype knowledge base $KB$ and a simple change expression set $ce$ we use this set of tuples as a function $I_c(KB, ce)(s) : SID \to \{s_1, ..., s_n \mid s_i \in SID\}$, where the first entry of each tuple is mapped to its second entry.

**Definition 7 (Value of a Prototype Expression's Property).** *Let* $KB$ *be a prototype knowledge base and* $id, p \in ID$*. Let* $(id, (base, add, remove)) \in KB$ *be a prototype expression. Then, the value of a property* $p$ *of a prototype expression with ID* $id$ *is* $J(KB, id, p)$*:*

$$I_c(KB, add)(I_h(p)), \text{ if } base = P_\emptyset$$
$$(J(KB, base, p) \setminus I_c(KB, remove)(I_h(p))) \cup I_c(KB, add)(I_h(p)), \text{ otherwise.}$$

Informally, this interpretation calculates the values of one property by traversing the inheritance chain from $P_\emptyset$ to the regarded prototype expression, recursively. If we derive a prototype expression from $P_\emptyset$ then we just interpret the property values which are defined in the add set of the prototype expression. In the case that we derive from a different prototype expression, we take the property values of the base and remove all property values which are defined in the remove set of the prototype expression. Right after, we add every value of the property which is defined in the prototype expression's add set. An example of interpreting a property value can be found in fig. 3.

After we defined how to interpret the values of one prototype expression's property, we will define the interpretation of a whole prototype expression, now. Basically, this interpretation applies the previous interpretation for every prototype expression's property.

**Definition 8 (Interpretation of a Prototype Expression).** *Let* $KB$ *be a prototype knowledge base and* $pe = (id, (base, add, remove)) \in KB$ *a prototype expression. Then, the interpretation of a prototype expression is defined as*

$$FP(KB, pe) = (I_h(id), \{(I_h(p), J(KB, id, p)) \mid p \in ID, J(KB, id, p) \neq \emptyset\}).$$

*We call the interpretation a prototype expression's fixpoint. The result is a Prototype.*

$$
\begin{aligned}
& J(KB, \texttt{ex:earth}, \texttt{ex:sat}) \\
=& (J(KB, \texttt{ex:mars}, \texttt{ex:sat}) \setminus I_c(KB, remove_E)(\texttt{ex:sat})) \cup I_c(KB, add_E)(\texttt{ex:sat}) \\
=& (I_c(KB, add_M)(\texttt{ex:sat}) \setminus I_c(KB, remove_E)(\texttt{ex:sat})) \cup I_c(KB, add_E)(\texttt{ex:sat}) \\
=& (\{\texttt{ex:phobos}, \texttt{ex:deimos}\} \setminus SID) \cup \{\texttt{ex:moon}\} \\
=& \{\texttt{ex:moon}\}
\end{aligned}
$$

**Fig. 3.** Examplary interpretation of the property satellite of the earth from the introducing example in fig. 1 and fig. 2. Le $KB$ be the prototype knowledge base which contains every prototype expression mentioned in the example. $add_M$ and $remove_M$ are change sets of Mars and $add_M$ and $remove_M$ are change sets of Earth. For the purpose of readability we write `ex:` instead of `example:` and `sat` instead of `satellite`.

The last definition describes how to interpret a prototype knowledge base.

**Definition 9 (Interpretation of a Prototype Knowledge Base).** *Let $KB$ be a prototype knowledge base. Then the interpretation $I_{KB}(KB)$ of $KB$ is $\{FP(KB, pe_i) \mid pe_i \in KB\}$. We call this the fixpoint of a prototype knowledge base.*

### 2.2 InterPlanetary File System

The following explanation is based on Juan Benet's paper about IPFS [4] as well as on the vision paper of combining IPFS and Prototypes [12] by Cochez, Hüser and Decker.

The InterPlanetary File System (IPFS) is a global, distributed peer-to-peer system. Easily, anyone can create a peer node in this network and store data in such a way that other peers can access it. When a user shares data then it is stored on his own node locally, first. By pinning, other users can decide to make your data available from their node, as well. Requesting means to find a node which offers that data. Then, these two nodes connect and transfer the requested information. To make this possible, IPFS combines several approaches, which are already known in practice: Distributed hash tables to access peer's objects on the peer-to-peer network, BitTorrent's exchange protocol to transmit data, version control approaches from Git and self certifying file systems. IPFS itself has a layer structure as shown below:

1. Naming
2. Files
3. Objects
4. Exchange
5. Routing
6. Network
7. Identities

Entry point for a user of IPFS is the *IPFS daemon*, which handles the peer node and serves as API. It is used to call IPFS functionalities via bash or its HTTP interface. The HTTP interface is used to define program language specific libraries, for example in C++, python, Haskell and Java[1]. A daemon has an IP, Port pair for listening to these API calls as well as several IP, Port Pairs to listen to other peers on the network.

First, this section deals with the object layer, then describes the file layer and right after the naming layer. In the end of this section, we look at the lower layers, which form the underlying peer-to-peer network. This should lead a deeper understanding of IPFS' behavior.

**The Object DAG** First of all, the whole IPFS system is based on *IPLD*, a huge *directed acyclic graph* (*DAG*) which is set up on a peer-to-peer network. Nodes of this graph are *IPFS objects*. The structure of an IPFS object can be found in fig. 4: It contains a data part, where we can store arbitrary information in form of a byte array, and a set of *Object Links* to other IPFS objects. These links build the edges of the DAG. Every link got a name, which must be unique in the object's link set, and a reference to the target object, called *Multihash*.

There has been a discussion on how link names should be restricted[2]. In order to keep an, later explained, IPFS link printable the developers decided to use UTF-8 encoding according to RFC 3629 [36], with several additional restrictions[3]. One restriction, for instance, is that "/" is not allowed in the link name, since it is a delimiter of IPFS links.

A multihash (or short *Hash*) is the identifier of an IPFS object. It is content based which means that the ID depends on the IPFS object itself. We can request an object via its *IPFS Link* `/ipfs/<hash-of-object>`. Transitive addressing over object links is also possible: `/ipfs/<hash-of-object>/<name-of-link>`. Let us now look into some examples how hashes identify objects: If we got two objects with different data, then their hashes will differ, since the data part influences the hash as well. If we got two objects with the same data and links which are same-named but the target of at least one link differs from the other one, then both hashes will still be different. Only when both objects are exactly equal, meaning their data as well as their links (name and target of each link) are equal, then they will have the same hash. The hash does not depend on the ordering of the object's link set neither on the peer node where we have stored the IPFS object.

When we request this data via its hash then we can not be sure (depending on the underlying layer of data exchange) from which peer we will get the object. Nevertheless, we can be sure that it is the data that we have requested, since it is content addressed. This kind of addressing also leads to immutability of objects, since changing an object would also change the content based hash again. That offers integrity of the object's components. Additionally, content

[1] `https://github.com/ipfs/ipfs\#api-client-libraries`
[2] `https://github.com/ipfs/go-ipfs/issues/1710`
[3] `https://github.com/ipfs/go-ipfs/pull/1740`

based hashes create the acyclic nature of the directed graph of objects (compare the left illustration of fig. 21). An other advantage is deduplication of objects which means that there are no two same objects with different multihashes.

All these promises rely on the assumption that we don't have any hash collision. Currently, SHA256 is used to generate hashes but the multihash format allows to change the hash function of the wohle IPFS system to a different one. A hash collision in SHA256 is very unlikely to happen and even if one would have been found it is easily possible to change the hash function of IPFS to SHA384, SHA512 or different hash function[4] with the consequence of dropping support for objects which use older hash functions[5]. Further reading about the security of the SHA versions can be found in [26]. Next to immutability, mutability might be useful in the days of high-frequently changing data, too. We regard this in a later section.

**Formal IPFS Object Representation** After we have defined IPFS object in their context of a DAG we will now introduce a formal definition on which we build up a graphical representation of IPFS objects:

**Definition 10 (IPFS Object).**

1. *A valid* link name *is a not empty UTF-8 sequence (according to RFC 3629 [36]) with up to 255 Unicode codepoints not including "U+000" - "U+001F", "U+007F", "U+002F", and not equal to "U+002E" and "U+002E U+002E". The set of all valid link names is called* LinkNames.
2. *A valid* Multihash *is a SHA256 value. The set of all valid Multihashes is called* MultihashSet.
3. *Let* $L \subseteq LinkNames, M \subseteq MultihashSet$. *Then the function* $l : L \to M$ *is a* Link List.
4. *Let* $l$ *be a Link List, and* $d$ *a byte sequence. Then* $(l, d)$ *is an* IPFS object. *We call* $l$ *the IPFS object's* Links, $d$ *its* Data.
5. *The Multihash of an IPFS object is the SHA256 hash over the IPFS object*[6].

Let $l : L_1 \to M_1, L_1 \subseteq LinkNames$, $M_1 \subseteq MultihashSet$, $target$ be a SHA256 hash and $d$ a byte sequence. In this thesis we use the following notation to represent an IPFS object $(l, d)$:

A box which contains $d$ has an outgoing arrow for each $(name, target) \in l$. An arrow is labeled with $name$ and is pointing to the IPFS object $o$ where the hash of the IPFS object equals $target$.

[4] `https://github.com/ipfs/faq/issues/24`

[5] `https://www.reddit.com/r/ipfs/comments/61r38j/hash_collision_resolution/dfktlan/`, Whyrusleeping (Jeromy Johnson) is a kernel developer of IPFS

[6] A JSON or Protobuf encoded IPFS object is parsed and converted to a DAG node (`https://github.com/ipfs/js-ipfs/blob/master/src/core/components/object.js`) and then serialized and multihashed (`https://github.com/ipld/js-ipld-dag-pb/blob/master/src/dag-node/create.js`)

**Create IPFS Objects** Next to requesting IPFS objects, it is of course also possible for every node to share own IPFS objects with other peer nodes. There are two ways to do so: *Put* an IPFS object directly on IPFS or *patch* an already existing IPFS object until it has the needed structure. When we put an IPFS object directly, then we need to construct it outside of IPFS, first. For example, this can be done by creating a JSON Object which is structured like the one in fig. 4. If the syntax of the IPFS object is correct, then this object can be put directly on IPFS. Patching on the other hand takes an already put IPFS object

```
{"Links":[{ "Name":"moon1",
            "Hash":"QmbdSD2gz8...",
            "Size":10},
          { "Name":"moon2",
            "Hash":"QmZYZUbMTC...",
            "Size":10}],
 "Data":"mars"}
```

**Fig. 4.** Example of a JSON encoded IPFS Object.

and changes its data, adds new links or removes existing ones. The patched object is still available after patching. An example of patching can be found in fig. 5. We create an empty object and derive the earth and the moon by setting the empty object's data. Then we add a link, named satellite, from the earth to the moon.

Note, that IPFS objects have limited capacity. The maximum size of an IPFS object is 2 MB. This does not mean that a user can not upload files which are bigger than 2MB. The file layer above is responsible for combining several IPFS objects to save a big file. More about that will be explained in "IPFS Files".

**Protect Objects from Garbage Collection** Especially, when we generate IPFS objects via patching, the amount of objects in the local storage increases fast, since every intermediate object is still in the storage even if it is not needed permanently. Therefore, IPFS introduces a *garbage collection* which deletes not needed objects. In the current version of IPFS (0.4.6) this garbage collection has to be called manually per default but could be switched to automatic collection as well. To know which objects can be deleted, IPFS includes a *pin* option. Only not pinned objects will be deleted by a garbage collection call. There are several ways how a user can pin objects: *Direct pinning* pins the object itself and nothing else. *Recursive pinning* pins the object itself and all its transitive linked objects. *Indirect pinned* objects, on the other hand, are pinned since they are linked to by an object which has been pinned recursively. An example for pinning can be found in fig. 5. We pin the earth recursively, therefore the moon is also protected from a possibly following garbage collection call. The moon object is pinned indirectly. Objects can of course be unpinned as soon as they are not

```
$ ipfs object new
  QmdfTbBqBP...
{"Links":[],"Data":""}

$ echo -n "earth" |
ipfs object patch QmdfTbBqBP... set-data
  QmZPu7jZzZ...
{"Links":[],"Data":"earth"}

$ echo -n "moon" |
ipfs object patch QmZPu7jZzZ...  set-data
  QmahjjNzGxD...
{"Links":[],"Data":"moon"}

$ ipfs object patch QmZPu7jZzZ... add-link
"satellite" QmahjjNzGxD...
  QmYKRJRhqT...
{"Links":[{ "Name":"satellite",
                "Hash":"QmahjjNzGxD...",
                "Size":6}],
 "Data":"earth"}

$ ipfs pin add -r QmYKRJRhqT...
  pinned QmYKRJRhqT... recursively
```

**Fig. 5.** Example of IPFS patch and pin usage.

needed anymore. When an object is pinned indirectly twice and one of these recursive pinned objects will be unpinned, then the indirectly pinned object will not be deleted by the next garbage collection call since it is still indirectly pinned by the other object.

**Reachability of Objects** At this point, the constructed object is stored in our own node, only. Currently, there is no self distributing service in IPFS, but these could be built on top of the existing layer structure. There are already projects realizing such a service (Filecoin[7], IPFS cluster[8]). In the current version of IPFS, an object becomes automatically available on your own node, when you have requested it from another one. Note, that the object is only temporarily available on your node since it is not pinned and therefore the next garbage collection call deletes it right away. Actively, a node's user can decide to pin other node's objects too, like it has been described in the previous section. When we store an IPFS object in our node's local storage every peer node of the node's *swarm* is able to reach this object via its hash or a link which points to this IPFS object. A swarm is the set of peer nodes which we are connected to in the network. When we request an IPFS object which is not available on the swarm, IPFS tries to get it until it is found. This means that currently, there is no timeout implemented here. Requesting IPFS object in general is done by the routing layer of IPFS.

In context of reachability, "The Permanent Web" as a nick name for IPFS might be a little bit misleading. We have to differ permanency and persistency: Permanency describes the ability that we got a link and every time we use that link we receive the same object as long as the object is available on the network. It does not include persistency, which, on the other hand means that objects are available at any time[9]. As already mention, the realization of such a mechanism has to be built on top of IPFS.

**IPFS Files** A higher layer of the IPFS stack structure is the file layer, which allows to use IPFS as a file system. It offers functionalities to represent whole file and folder structures on IPFS. Remember that the maximum size of an IPFS object is 2MB. To handle bigger files, IPFS automatically chunks files in to several object which are then called blocks. Several blocks are concatenated by lists and trees, which are basically IPFS objects containing links to their entries. Overall, this behavior is adapted from Git's data blobs and tree structure [30]. More about IPFS files can be found in the IPFS paper [4].

**Mutable Content - IPNS** Already integrated in the InterPlanetary File System is the *InterPlanetary Name Space* (*IPNS*), a naming service allowing mutability on top of the immutable, permanent IPFS DAG. It works by implementing

[7] `https://www.filecoin.io/`
[8] `https://github.com/ipfs/ipfs-cluster`
[9] `https://github.com/ipfs/faq/issues/93`

variable pointers to immutable IPFS objects. Every peer node in the network has one pointer which can be set to an immutable IPFS object. This IPNS pointer is addressed by the peer node's ID, which is the hash of the peer node's public key. Setting this pointer is called *publishing*. We address the linked object via an *IPNS Link* `/ipns/<peer-id>`. An IPNS link can be *resolved* to an IPFS link. Note the different between IPFS and IPNS links: These two concepts are not only different due to their first link segement (`/ipfs` and `/ipns`), they are also separated in their place of action: IPNS links can not be used as a reference inside the IPFS object DAG. An example where the IPNS naming service is used, can be found in fig. 6. At first, the IPNS pointer of our peer node with the hash `QmaXA8NViH...` is set to the "mars" object (`QmQsuXWzNK...`). After resolving the IPNS pointer, we publish the "earth" object (`QmYKRJRhqT...`). Resolving again gives us the IPFS link to the eath, now, and not the link to the mars object any longer.

```
$ ipfs name publish QmQsuXWzNK...
  Published to QmaXA8NViH...: /ipfs/QmQsuXWzNK...

$ ipfs name resolve
  /ipfs/QmQsuXWzNK... ⟹ mars

$ ipfs name publish QmYKRJRhqT...
 Published to QmaXA8NViH...: /ipfs/QmYKRJRhqT...

$ ipfs name resolve
 /ipfs/QmYKRJRhqT... ⟹ earth
```

**Fig. 6.** Example of IPNS usage.

**The underlying P2P network** Layer 1 to 4 create a peer-to-peer network on which the already explained IPFS objects can be stored in a DAG, combined to files, which then might want to be address in a mutable way. The peer node got a private and a public key. The ID of this node inside the network is the hash over the public key. To find other peer nodes and especially to find IPFS objects on the network, the peer-to-peer network uses a *Distributed Hash Table* (*DHT*). The IPFS DHT is based on Coral [23] and S/Kademlia [3]. To gain performance, objects which are smaller than 1 KB will be stored directly in the DHT. If the object is bigger than 1 KB, the DHT refers to the block where it is stored.

Next to finding peers and objects, publishing nodes IPNS referred hashes need to be handled by the routing system, too. The node's pointer is published on the routing system's DHT and therefore is not only available from your own node. So even if your node is offline, a resolution is possible for a certain time (standard is 24 hours depending on an editable time to live property). More

about the underlying peer-to-peer network and distributed hash tables can be found in the IPFS paper, as well as the papers about Coral and S/Kademlia.

### 2.3 Task of the Thesis

The primary goal of this bachelor thesis is combining the prototype knowledge representation with the distributed file system IPFS. That should lead to a stronger reuse of resources inside the knowledge representation than previous implementations. Center of the thesis will be the design and the implementation of a tool which takes a given file of prototype expressions and put the file's expressions on IPFS. At first, in an immutable way and then allowing mutability of prototype expressions, as well. A goal here are prototype expressions which contain change expressions with more than one value, which is, due to IPFS' unique link name per object assumption, not directly possible. Another task is to find a proper representation of a prototype knowledge base. Two reasoning functionalities on this system are desired: Computing the fixpoint representation of a knowledge base is as necessary as checking its consistency. A last goal is finding a method of pinning prototype expressions such that they are fairly distributed on the network. That should imply some kind of persistency, at least for frequently used prototype expressions. In the end, Benchmarks of the constructed ProtoIPFS should be presented.

## 3 From Distributed Knowledge to Prototypes - The State of the Art

In this section we look at several projects which relate to the task of combining prototype knowledge representation and the peer-to-peer network IPFS. First, we regard an early implementation of prototypes, which does not include saving knowledge in a distributed way yet, but still allows to share your knowledge by sending the whole data to someone else. Then, we look at two implementations of distributed knowledge: *Semantic SwarmLINDA* and *SWAP* set up RDF represented knowledge in distributed systems. Next, we look at Cochez's implementation of prototypes which already includes an option for distributed knowledge representation. Finally, we will see that there is no such combination of prototypes and P2P networks, which directly leads to the section about the justification of the thesis. But let us have a look at related work, first.

### 3.1 Related Work

**SubClassOf** The following overview over SubClassOf is based on Jonas Almeida's description of his implementation [1]. SubClassOf is an approach which comes up with a Javascript implementation of Decker's early idea of prototypes, which has been presented at the "Conference of Semantics in Healthcare and Life Sciences" in 2013 [17]. Inspired by the RDF property `rdf:subClassOf` it allows to derive from already defined data, encoded as JSON objects, and

then makes it possible to add new properties, afterwards. The data is somehow comparable to the current definition of a prototype like it has been visible in definition 3. For example, we can derive the earth from the mars by adding all additional properties of the earth to the mars:

```
mars={type:"terrestrial"}
earth={habitant:"human"}
earth.subClassOf(mars)
⟹ earth={type:"terrestrial", habitant:"human"}
```

It is also possible to use remote data as a base, by requesting an online stored JSON objects via HTTP. By introducing frequent data update callbacks, it is also possible to create self updating inheritances. This means that objects which are a subclass of a remote object always derive from the latest version of the the remote one, even if the author changes it. Note, that this approach had been implemented before Decker, Cochez and Prud'hommeaux presented their definition of prototypes, such that removing properties from the base, knowledge bases and their definition of consistency is not included in subClassOf. But still, in some sense fixpoint computation is. Since the JSON object dynamically derives from its base, subClassOf computes a fixpoint of a prototype expression on the fly.

**Semantic SwarmLINDA** The basis for Semantic SwarmLINDA is the distributed programming language LINDA [25], which offers a passive data value storage in form of a so called *tuple space*. A tuple space contains functions, which can be used to read and write into an external shared storage. Data inside the tuple space are tuples having an identifying name and arbitrary many entries of different types. Since tuples are still available in the space when the creating program has terminated, this approach is not only distributed in space but also distributed in time. Distributed in time means that data can outlast their program's lifetime.

Graff realized that the shared system has problems when it is set up on a huge network [28]: The centralized component of this system shrinks scalability and availability, which in his opinion will not be fixed by any centralized solution. Graff looked into models from nature, and designed a system which is comparable to a swarm of ants, though, other experts stick to the centralized plan [37]. In his system, each ant works locally, and only does small tasks. Some ants might even misbehave, but still the whole ant colony works. These principles are used in his *SwarmLINDA* approach. It is a set up of several, linked tuple space storages. Then, there are two types of crawler-like individuals, which search requested tuples on the cluster or which sort tuples into the right storage. The tuples are sorted by the tuple's type, which is defined by the amount of entries and their entries' type. Same typed tuples are stored in the same region of the network.

Afterwards, Augustin took this distributed tuple storage and saved RDF triples instead of arbitrary tuples, which leads to a distributed knowledge representation [2]. Sebastian Koske mentions that this kind of combination makes SwarmLINDA's cluster unstructured, since every RDF triple has the same tuple

type. To handle that, he approached to use semantic neighborhoods to order the cluster again [29]. His approach is called Semantic SwarmLINDA.

**SWAP** The *Semantic Web and P2P* (SWAP) project is a combination of conventional knowledge representation and P2P networks: Knowledge can be imported from local databases, emails and files to the local RDF repository of a peer [22]. Now, a node can use SeRQL or a graphical interface to query for information on the system. The peer tries to solve the query locally first, but also forwards the query to several neighbored and from a peer-selector chosen, peers. These chosen nodes evaluate the query against their local repository and send an answer back to the requesting node. Additionally, these peers forward the query to other neighbored, and from their peer-selector chosen, peers. Peer selection depends on a query's meta data and can also be learned from the previously recognized answer behavior of a node [34]. Loops and ever lasting queries are prevented by a time to live attribute of each query and a detection at each node whether the query has already been answered. Further details about query routing can be found in the SWAP deliverables D 3.5 [21]. The answer of a query can be saved in the requesting node's local storage, then. Complex queries, where parts of the needed information lay in different repositories, can be answered by splitting the query into sub-queries. An indexing structure is used to decide which part of a query has to be directed to which information source. More about complex querying can be found in the SWAP deliverables D 3.6 [20]. Since several stand alone RDF graphs are combined by such a query, inconsistency is possible. To improve querying, a meta data model for better requests has been developed [11]: For every published information several RDF formatted data, like labels, addition date and visibility, will be assigned, which allow more detailed queries. Also peer nodes themselves are assigned additional information, like a label and information about trust.

**Tools for Prototype Based Ontologies** Cochez, who has already worked on the definition paper of prototypes, implemented a tool for storing prototype expressions locally, remotely and distributed [14]: His idea is to create several stand alone knowledge bases of immutable prototype expressions, which then can be shared via HTTP. In the case of multiple sources have defined different prototype expressions with the same ID, the implementation offers several joining mechanisms. In contrast to the already regarded prototype implementation subClassOf, this implementation allows consistency checking and creating knowledge bases.

Additionally, the tool provides several generators for benchmarking data: *Baseline*, where prototypes are derived in a tree structure and have no properties. Then *Blocks*, where several prototype expressions are grouped into blocks and derive from a previously defined block. And *Incremental*, where every prototype expression derives from a random, previously defined prototype expression.

In general, Cochez used several design decisions: First, according to the definition of prototype expression the remove all operator should add all possible

IRIs to the remove set (cf. definition 5). Since this is not feasible in a real life application, this is treated as a special case. Secondly, we can not be sure that the user defines prototype expressions which use valid absolute IRIs. By introducing an IRI type class the user is forced to define syntactically valid IRIs. Thirdly, when a user defined a knowledge base, then it is possible to compute the fixpoint. The implementation introduces a method where not a single prototype is computed twice, due to reusing already computed prototypes.

### 3.2 Justification

Saving knowledge in a distributed manner is recommended [7], since it is closer to most applications structure and the distributed nature of knowledge itself. Especially peer-to-peer networks seem to be a good underlying system for the case of knowledge representation [8, 22]. A peer-to-peer network aims to set up a network of equal nodes for a shared usage of distributed resources where the organization should be decentralized [32]. Advantages of using P2P in the subject of knowledge representation have been presented in "Semnatic Web and Peer-to-Peer" [33]: Since every node organizes itself, we gain less administrative overhead, and due to equality of distributed peer nodes we avoid bottlenecks on the system. Additionally, "Peer-to-Peer Knowledge Management" [9] sees the ability to gain robustness of the knowledge system by using P2P approaches. The system can be designed in such a way, that nodes do not depend on the availability of other nodes. Instead, available nodes just add new knowledge to the system which can be used by other peers.

As we have seen in the previous chapter, there are already several implementations of distributed knowledge representation systems, even with an underlying P2P network. Now, one could ask why it is necessary to construct a prototype knowledge representation on top of IPFS, when we already have these implementations. The reason for that are disadvantages of these existing systems: Except for subClassOf and Cochez's prototype tools, all presented approaches work with the conventional knowledge representation RDF. As mentioned in the introduction, there are reasons to move away from this kind of representation and look at the horizontally sharing prototype representation. Unfortunately, the SubClassOf implementation does not offer a desired distributed representation and also Cochez prototype implementation has struggles with combining several sources of prototype expressions, for example if there are expressions with the same ID one have to merge these expressions. Furthermore, it is difficult to resolve prototype expressions and it is not possible to know which host might offer information about a resource. Therefore, it is necessary to look into other possible ways of distributing prototype expressions. Since peer-to-peer networks are recommended, putting prototypes on top of them seems to be a good idea.

A currently rising global P2P network, which combines several already established systems, is IPFS (cf. section 2.2). Its linkable object structure seems to fit naturally to prototype expressions: When we regard prototype expressions as objects and their base or add and remove properties as links, then we already establish the basic aspects of a mapping. Mutable prototype expression then

can be realized, by using IPNS links. An advantage of using IPFS is its content based addressing which solves the problem of inconsistency triggered by different expressions which were assigned the same ID. This is a problem for SWAP and had been for the first prototype implementations, which was then fixed by introducing several merging strategies. Furthermore, content based addresses make it possible to resolve expressions on the whole swarm only by knowing its hash. IPNS can additionally be used to advertise node's prototype expressions. Also interesting is the fact that all prototype expressions lay in one big DAG, which means that it is technically easily possible to reuse already defined prototype expressions, also from other peers' user. Plus, the layer like structure of IPFS allows to build a system on top, easily. Therefore, this thesis presents a mapping, as well as, an implementation of the combination of prototypes and IPFS.

## 4 Mapping Prototypes to IPFS

In the following we construct the mapping from prototypes to IPFS. We start with the representation of immtuable prototypes in IPFS. Then, we deal with prototype knowledge bases. In the end, we extend these mappings to gain mutable prototype expressions as well as accessibility of knowledge bases and prototype expressions. Therefor, we introduce the administrative directory which will be published via IPNS.

### 4.1 Immutable Prototype Expressions

First of all, the set of absolute IRIs *ID*, which has been used to identify prototype expressions, becomes obsolete since in IPFS objects are identified by content based hashes. But still, for readability reasons we keep the IRIs as additional information of prototype expressions in IPFS. This mapping is based on the vision paper of Cochez, Decker and me [12].

A simple change expression $(p, \{r_1, ..., r_m\})$, where $p \in ID$ and $r_1, ..., r_m \in ID$, is mapped to a directory like object "proto:SEVERAL", which holds a link to each IPFS prototype expression representation $pe_i$ with ID $r_i, i \in \{1, ..., m\}$. Thereby, the link to $pe_i$ is named $i$. The resulting IPFS objects are visible in the left image of fig. 7. For the purpose of generalization, even if $m = 1$ we will use the "proto:SEVERAL" object, here (right of fig. 7). To this directory like structure another object refers with a link named $p$.

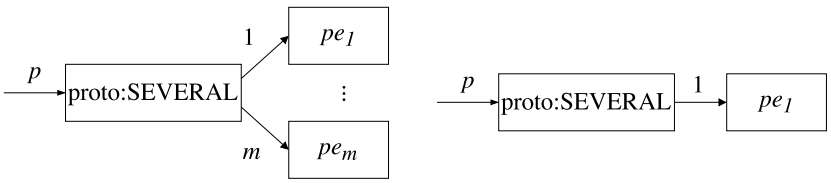


**Fig. 7.** The simple change expression in IPFS.

A special change expression is the remove all expression $(p, *)$ which is used in a remove set's simple change expression to remove all properties with the ID $p$. In IPFS this change expression is realized by an IPFS object with the data "proto:ALL" which we use as the target of a link named $p$:

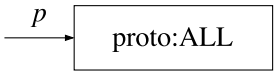


**Fig. 8.** The simple change expression with * in IPFS.

Let $id \in ID$, $base \in ID \cup \{P_\emptyset\}$ and *add* and *remove* be simple change expression sets. Now a prototype expression $(id, (base, add, remove))$ is mapped to an IPFS object whose data is the *id*. This object has a link named "base" to its base prototype expression, as well as links named "add" and "remove" to two directory like objects. These two objects with data "proto:ADD" and "proto:REMOVE" have links to all simple change expressions which should be applied on the base. Links to these simple change expressions, which have been introduced before, are indicated in gray:

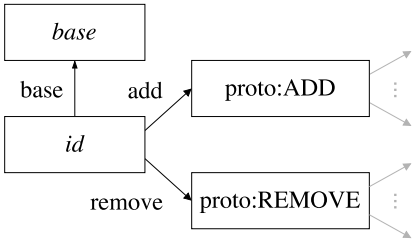


**Fig. 9.** A prototype expression in IPFS.

A special prototype expression is the $P_\emptyset$ (`proto:P_0`). The IPFS construction of $P_\emptyset$ has neither an add nor a remove change expression nor a base. Note, that it still has links to the IPFS objects representing the set of add or remove expressions, but for $P_\emptyset$ these objects do not contain any links:

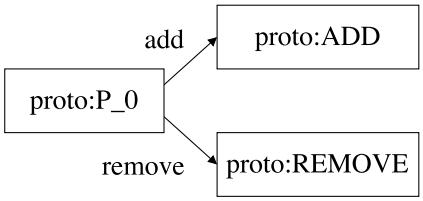


**Fig. 10.** $P_\emptyset$ in IPFS.

Now let us have a look at an example: The IPFS representation of the introducing example of mars and earth can be found in fig. 11.

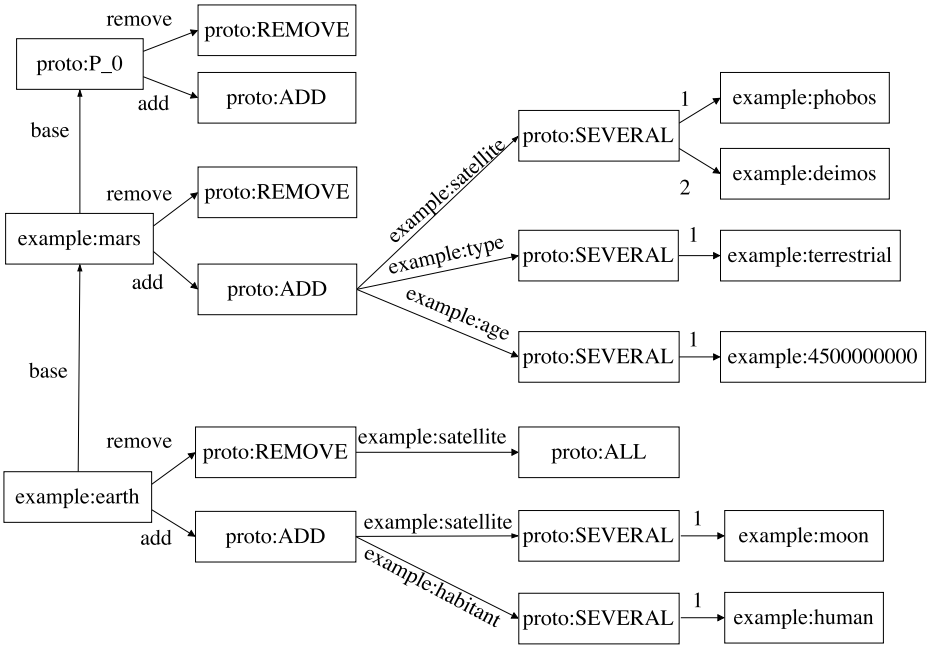


**Fig. 11.** Visible is the realization of the introducing prototypes example on IPFS according to the mapping in section 4. Note that, all properties of these two prototype expressions (for instance "example:phobos") are prototype expressions themselves, as well. For readability reasons their base add and remove objects have been left out.

### 4.2 Knowledge Bases

Let $PL = (P_\emptyset, ID, \texttt{PROTO})$ be a prototype language and $KB \subseteq \texttt{PROTO}$ a finite set of prototype expressions. Now, we use properties of Named RDF Graphs [35] to

realize knowledge bases. In IPFS we store $KB$ as an IPFS object containing a name $KB_{name}$ as its data. Similar to Named Graphs, the knowledge bases name is an IRI. This name is unique per peer node and should describe the content of the knowledge base. The links of this object refer to all prototype expressions $pe_i \in KB, i \in \{1, ..., n\}$. The name of these links have no function therefore they are just numbered. Note, that mutable prototype expressions are possible to use in the links set, as well. When we use mutable links inside a knowledge base's prototype expression it is recommended to add this mutable link to the knowledge base instead of the immutable one. That would automatically update the entries of a knowledge base as soon as a mutable link changes. How exactly mutable prototype expressions are realized on IPFS is presented in section 4.3. In the end fig. 12 describes the $KB$.

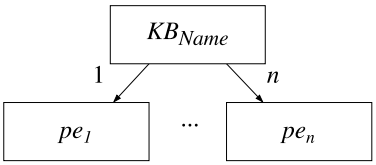


**Fig. 12.** Knowledge base in IPFS.

### 4.3 Mutable Prototype Expressions and Knowledge Base Addressing

When we talk about mutable prototype expressions one needs to be clear that it does not mean that prototype expressions become mutable. In IPFS every object is content base addressed, therefore it is not possible to change an object without changing the link referring to it. But alternatively it is possible to introduce mutable links, which simulate mutable objects in IPFS. The system which provides these mutable links is IPNS. So there are neither for prototype expression nor for knowledge bases essential changes in their definition except the following introduction of mutable links.

To offer mutable prototype expressions and knowledge base addressing we generate an *administrative directory*. This structure holds A) links to all prototype expressions which the node offers and B) links to all knowledge bases which that node has published. Then the administrative directory has to be made public, such that other peers can use these information about offered prototype expressions and knowledge bases. For that, we publish the whole administrative directory on the peer node via IPNS. For a peer node which has published knowledge bases $KB_{Name_1}$ to $KB_{Name_n}$ and additionally offers prototype expressions $pe_1$ to $pe_m$ the administrative directory is defined in fig. 13.

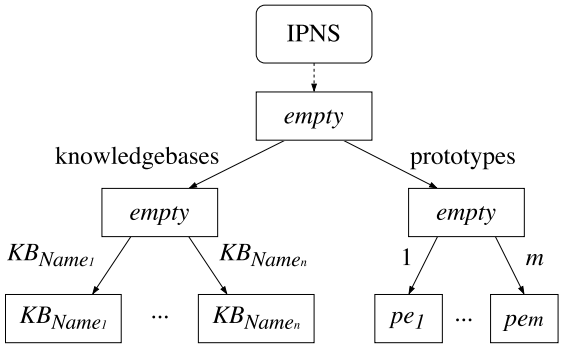


**Fig. 13.** The structure of an administrative directory.

The administrative directory allows every peer node from the swarm to browse the published knowledge bases via `/ipns/<peer-id>/knowledgebases`. Additionally, it is possible to refer to the peer node's prototype expressions via `/ipns/<peer-id>/prototypes/`$i$, $i \in \{1, ..., m\}$. Whenever there is a change inside a knowledge base or a prototype expression, the administrative object and the IPNS pointer have to be updated. A result of this construction are mutable prototype expressions:

Let $pe_i$ be a prototype expression which should be changed to a prototype expression $pe_i$*. A different prototype expression $pe_j$ should always refer to the latest version of $pe_i$. To gain that, $pe_j$ refers to `/ipns/prototypes/`$i$. When we update $pe_i$ we change the administrative object such that `prototypes/`$i$ refers to $pe_i$* and publish the new administrative object on IPFS. The result is that via `/ipns/prototypes/`$i$, $pe_j$ now addresses the new $pe_i$*.

One point to be mentioned here is the strict separation between IPFS and IPNS links. Therefore, it is not possible to use IPNS links in the link list of an object. The result is that we need to store IPNS links differently: For each mutable link we need to generate an *IPNS tunnel object*, which is an IPFS object containing the IPNS link inside the data part. The naming service resolves this link and refers to the current version of the prototype expression. A mutable link named $linkName$, pointing to the current version of a prototype expression $pe_i$, controlled and distributed by a peer node *peer-node*, is setup in fig. 14.

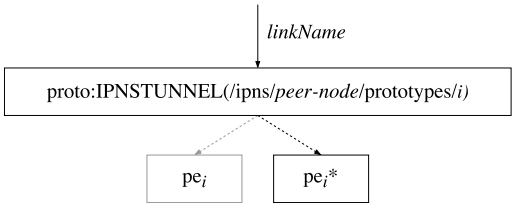


**Fig. 14.** An IPNS tunnel object in IPFS.

In fig. 15 we see a prototype expression on the left hand side which represents the interplanetary traveler Rick. Rick's current position changes often which is the reason why this property is mutable. Via the IPNS tunnel object we refer to the administrative directory of peer 1 which holds the current location of Rick. By creating a new administrative directory where we replace mars by planet earth and publishing this new directory, we change Rick's current location to earth.

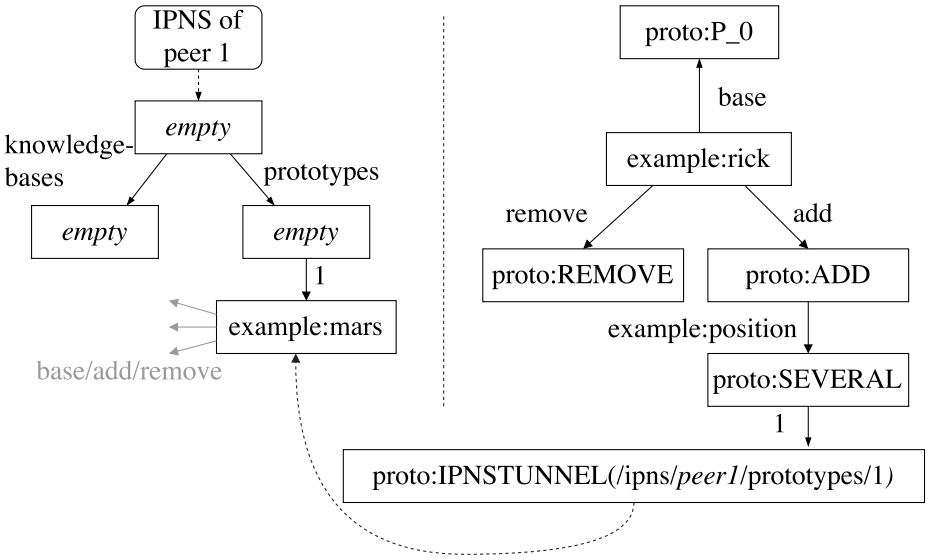


**Fig. 15.** An exemplary usage of an IPNS tunnel object to create mutable links inside the immutable IPFS DAG. Note, that `example:mars` is an own prototype expression. Its change sets and base are hinted. *peer1* is the peer ID of peer 1.

## 5 Realization of the Concept

This section deals with the implementation of the previously defined mapping from prototypes to IPFS: ProtoIPFS. In the beginning, we regard functionalities which ProtoIPFS provides. Then, we look into the system's architecture and ProtoIPFS' program structure. After that, we regard several programming decisions which have been made. These include inter alia: Made IPFS API adaptions, how ProtoIPFS handels adding prototype expressions, checking consistency and computing fixpoints. We also describe ProtoIPFS' local storage management, such as how it handles a problem of mapping IRIs to IPFS object's link names.

### 5.1 Functionality

Before the user can work with the implementation, he has to start an IPFS daemon first, which must be bound to ProtoIPFS, then. Several ProtoIPFS instances can run at the same time on one device, as long as there is an own IPFS daemon, with its own port configuration, for each ProtoIPFS instance. After that, the system offers following core services:

- create new prototype expressions from a file
- create new prototype expressions derived from already existing ones
- build prototype knowledge bases
- check consistency of a prototype knowledge base
- compute the fixpoint of a prototype knowledge base

ProtoIPFS allows to import a file of prototype expressions and put as well as publish them on your own node with the IPFS hashes as their new ID. By default every defined link is immutable. As it has already been mentioned, mutability is often wanted. Therefore, it is possible to define mutable links in this file via a flag, as well. Note, that in such an input file, it is only possible to refer to other prototype expressions which are defined in that file, too. Prototype expressions which refer to already existing prototype expressions have to be defined differently:

The user can construct a prototype expression by handing over IPFS or IPNS links. These links can be used to refer to an already been published base or a property value. The newly constructed prototype expression can be published on ProtoIPFS, such that it can be reused by other users. Note the difference between putting and publishing: Putting generates the IPFS object representation of a prototype expression. It allows us to address the prototype expression via its IPFS link. Publishing a prototype expression after that is necessary to allow mutable links and to allow other nodes to see which prototype expressions you are providing. Since IPNS links are allowed here, mutable links in bases or as property values are possible. More about adding prototype expressions can be found in section 5.5.

After prototype expressions have been published, they can be used to define a prototype knowledge base. A user can build a knowledge base out of his own prototype expressions, as well as, expressions which have been published by other users and which are reachable from the user's node. A discussion about reachability can be found in section 6.8. Then, the user is able to check whether a knowledge base is consistent. If it is so, ProtoIPFS can compute the fixpoint of the knowledge base. Note, that the result of both services depends on the reachability of every prototype expression inside the knowledge base, as well as, the state of mutable links. A detailed description how consistency checking and fixpoint computation work, can be found in the section 5.6 and section 5.7.

Of course, it is also possible to depublish prototype expressions and prototype knowledge bases from your local node. But still, other users can decide to pin your own prototype expressions, such that, even if you remove one from your local node, it still might be available in ProtoIPFS. We conclude the functionality

section with an examplary workflow of ProtoIPFS, which can be found in fig. 16.

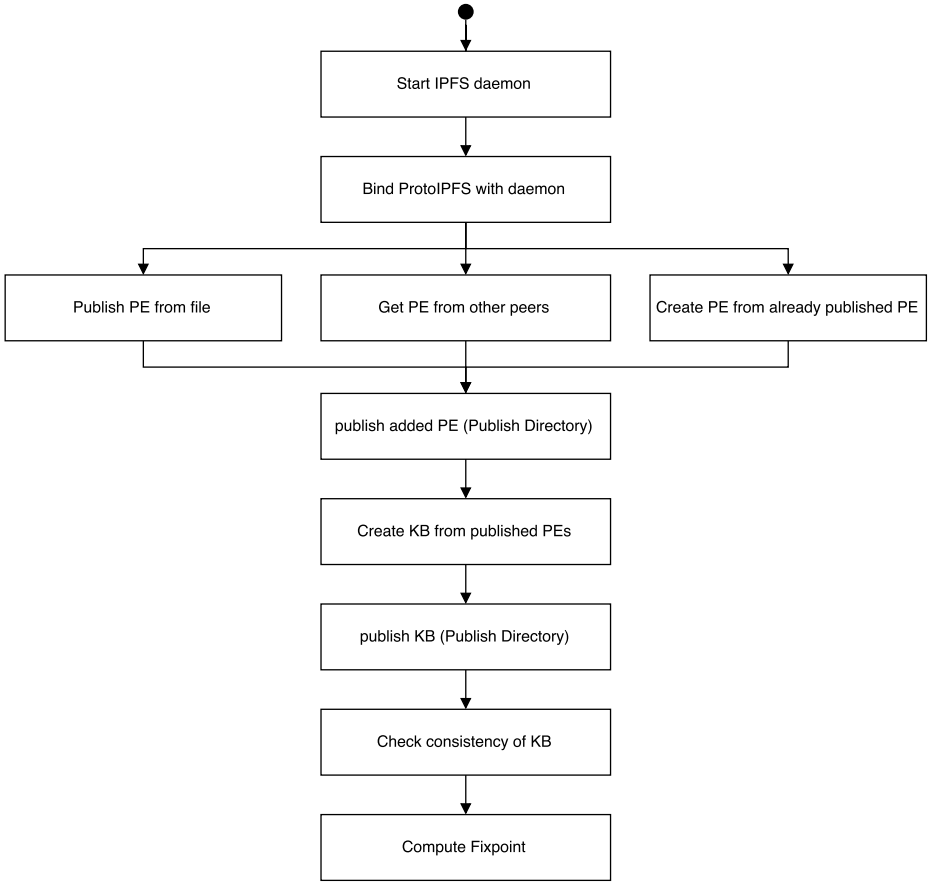


**Fig. 16.** An exemplary work flow with ProtoIPFS. After opening and binding a new IPFS peer node, a user can request published prototype expressions (PE) or define new PE. Then the user publishes the PEs to allow mutable links. These PE can be combined in a prototype knowledge base (KB), which needs to be published afterwards. ProtoIPFS can check whether a KB is consistent and is able to compute its fixpoint.

### 5.2 General Architecture

The architecture of the whole system is a layer structure. A visualization can be found in fig. 17. The lower layers are the InterPlanetry File System's layers, which have already been presented in section 2.2. The IPFS daemon is responsible for these layers and interacts directly with our local storage. For accessing

the functionalities of IPFS, ProtoIPFS uses the IPFS Java API, which interacts with the daemon via HTTP requests. Unfortunately, this API does not offer all functionalities which are needed in the layer above, which is the reason why there is an API extension for exactly these functionalities. A description of the API extension can be found in section 5.4. ProtoIPFS is placed on top of the extended API layer. It is responsible to construct prototype expressions and knowledge bases, to manage the node's IPNS link and to generate the administrative directory. These structures have already been presented in the section 4. Then, ProtoIPFS uses the lower layers to make these constructions available to the network. The ProtoIPFS layer itself can be used in third party code, therefor it serves as a lower layer for a new project.

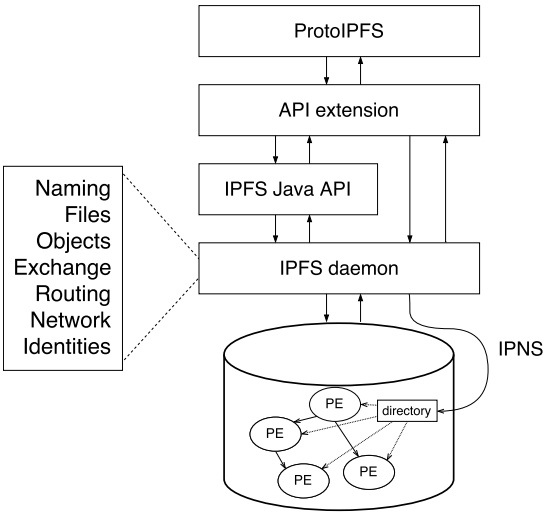


**Fig. 17.** Visible is the layer structure of ProtoIPFS with its underlying system, containing the extended Java API and the layer structure of the InterPlanetary File System. IPNS refers to the administrative directory holding links to all published prototype expressions and knowledge bases of the peer.

One aspect, which has not been regarded yet is the case of several peers interacting with each other. This happens, when the user requests prototype expressions or knowledge bases from other peers, as well as if there are prototype expressions which refer to objects laying in other peer's local storage. Basically, there is no change for the user here, since communication, exchange and dereferencing of IPFS objects (including prototype expressions) is done by the IPFS layer. The only difference here is, that the user might need to know how to address the objects from other peers. For objects which were referred by immutable IPFS links, there is no need to know where the object is stored as long as it is inside your swarm. On the other hand, there are two cases were we

need to know the peer's ID: A) if a user wants to request a knowledge base from that peer or B) if a user wants to create a mutable link which is controlled by the peer. Since the peer node's ID is the hash of the peer node's public key, the public key alone is enough to determine the peer node. A visualization of inter peer communication can be found in fig. 18.

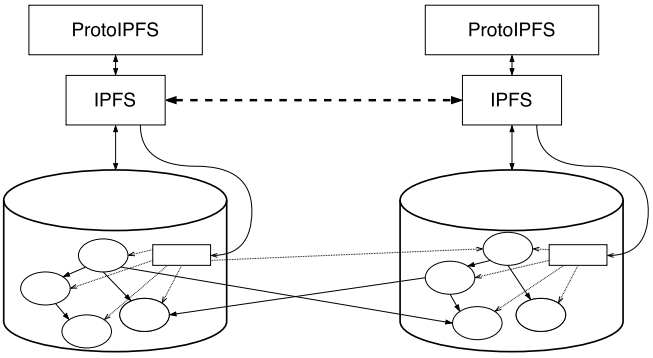


**Fig. 18.** Two Prototype participants sharing knowledge, inside one swarm of nodes. Links between objects which lay on different nodes are handled by the InterPlanetary File System.

### 5.3 Program Structure

After dealing with the architecture of the ProtoIPFS system, this section describes the implementation's structure and important design decisions. In the following sections, we always refer to the ProtoIPFS class diagrams in appendix A. But before we look in to classes of ProtoIPFS, let us regard the overall structure. First of all, ProtoIPFS is split into several packages:

- `domhues.prototypes.peer` for core functions.
- `domhues.prototypes.expressions` for data structures representing prototype expressions and prototypes.
- `domhues.prototypes.knowledgebase` for knowledge base representation.
- `domhues.prototypes.links` for IPFS and IPNS link representation.
- `domhues.prototypes.examples` for example usage and benchmark tests of the whole ProtoIPFS system.

**Core Classes** The most important class for the user is `PeerNode`. This class describes the interface of ProtoIPFS. It is bound to exactly one IPFS daemon, which is accessed by an IP and a port during construction. Via this class get, put and publish operations of prototype expressions and knowledge bases can be executed. To do that, this class uses the `IpfsApi` class, which will be described

in section 5.4 in detail. For prototype expression file imports, the `Parser` helper class is used. Since this class becomes quite large when everything is implemented right in `PeerNode`, we use the delegation pattern [24] and therefor shift PeerNode's put-implementations to the `PeerNodePut` and get-implementations to the `PeerNodeGet` class, which both are attributes of `PeerNode` and each is handed over a `PeerNode` reference. Publishing prototype expressions and knowledge bases is handled by the `Directory` class, the representation of the administrative directory.

**Prototype Expressions** Before something can be put and published, it must have been constructed, first. Therefor, there are several representations of prototype expressions defined in ProtoIPFS: `IncomingPrototypeExpression`, `PrototypeExpression` and `PublishedPrototypeExpression`. Each of them contains a base, an add and a remove set. Depending on the phase where we are looking at a prototype expression, one of these classes must be used. `IncomingPrototypeExpression` is needed as a data structure to represent expressions which were imported from a file. This class does only operate in the background and is not directly needed by the user. `PrototypeExpression` must be used when the user wants to define a new prototype expression deriving from an already put and published one. How exactly this works can be found in section 5.5. The difference to a incoming prototype expression, is the fact that IPFS or IPNS links to already defined expressions are used here, whereby an incoming prototype expression only contains IRIs as links. `PublishedPrototypeExpression` is the type which is returned whenever the user requests a prototype expression from the system. As it is visible in the class diagram, this class extends `PrototypeExpression`, such that it could easily be casted to that type. The only difference between these two classes is that a published prototype expression additionally has an IPFS hash, which a not yet put one does not have.

All three classes do not have any set methods such that all attributes have to be set via the expression's builder class. This design decision was made, since the idea of prototypes is the reuse of objects. If the user wants to change an existing prototype expressions, then he can derive from this prototype expression (use it as a base) instead of changing an existing expression's change sets. After that he can add or remove links to the derived one. This does especially hold with the immutable definition of IPFS objects.

For every prototype expression there are two `ChangeList`s which represent an expression's add and remove set. Depending on the prototype expression type, the change list's change expression is a `IncomingChangeExpression` or a `PublishedChangeExpression`. `PrototypeProperty` represents a property of a prototype including its value. These three classes extend the `Property` class.

**Prototype Knowledge Base** Another class which is important for the user is `PrototypeKB`. It is the representation of a prototype knowledge base and contains, next to an IRI as its name, links to all of the knowledge base's prototype

expressions. The class is used to construct a new knowledge base. An extending class is `FixedPrototypeKB`, which is constructed out of a `PrototypeKB` and represents a knowledge base where every mutable link is fixed at the time the object is constructed. That is necessary, to avoid a change of mutable links during computations on the knowledge base, which might influence their result. `FixedPrototypeKB` then offers methods for consistency checking and fixpoint computation, which will be presented in section 5.6 and section 5.7. The result of a fixpoint computation is a set of prototypes, represented by `Prototype` objects. These objects contain a name and a list of `PrototypeProperty` objects.

**Links** Important components of ProtoIPFS are links, since they address objects in the underlying IPFS. We differentiate between links and `Multihash`es: A link represents a path of the form `/ipfs/<hash>/...` or `/ipns/<hash>/...`, whereby a `Multihash` represents the `<hash>`, only. A `Link` is an abstract class from which is extended by three type of links: `IpfsLink`, `IpnsLink` and `AllLink`. The first two links represent the two possible link types in IPFS. `AllLink`, on the other hand, describes the $*$ operator in a remove change expression $(p, *)$. A `PrototypeReference` is a tuple which contains an IPFS and an IPNS link. It is used to hold a mutable and an immutable link of a published prototype expression, which a user can use in following ProtoIPFS operations.

### 5.4 API adaption

ProtoIPFS uses the officially recommended Java IPFS API to interact with IPFS. This implementation uses requests to the HTTP interface of a local IPFS daemon. For ProtoIPFS the most important features of this API are object, pin and name commands. Even if there is no documentation of the Java API, it is usable, since it is mostly analogues to the IPFS HTTP API[10]. But still, the Java IPFS API does not suit for the ProtoIPFS implementation, completely. Therfore, the `IpfsApi` class extends the IPFS Java APIs `IPFS` class due to adding several functionalities:

**Method Adaption** First, there are some functionalities of the Java IPFS API which are complicated to use in the main code of ProtoIPFS, such that it is helpful to wrap them in newly defined methods. Especially for patching objects this is the case. The Java IPFS API offers one method which is responsible for all kinds of patching objects: Set data, add links and remove links. The method has a long parameter list whose entries are not all necessary, depending on the patch type. We now generate one wrapper method for each of the three patch types. Their suiting method headers result to more structured ProtoIPFS code. Same argumation holds for the methods `putObjectAsJSON` and `newObject`.

[10] `https://ipfs.io/docs/api/`

**Direct HTTP Requests** Secondly, the offered method to request an IPFS object is not suiting for ProtoIPFS. `IPFS.object.get` returns the JSON representation of an IPFS object as it is shown in fig. 4. With the difference that it does not contain any link's name, which are important for ProtoIPFS since they contain important information according to the presented mapping from prototypes to IPFS. Therefore, `IpfsApi` includes an alternative `getObjectAsJSON` method whose returned JSON object includes the link's name, additional. This is done by requesting the daemon via HTTP directly.

The method `selfResolve` is used to get the hash of the object, which has been published by your own node, without a need to know your own peer's ID. This functionality, and a function which lists all pinned objects of your node (`pinls`), are implemented in the daemon but not in the Java API. Therefore, we use an HTTP request to the daemon directly to implement these missing functionalities.

**Additional Functionalities** A method which converts a link of the form `/ipfs/<hash>/<link name1>/.../<link nameN>` or `/ipns/<hash>/<link name1>/.../<link nameM>` to the multihash of the referred IPFS object is useful. `IpfsApi` includes the method `resolveLink` to do that. If it is an IPNS link, the method first resolves the peer ID. After that it treats the rest of the link as an IPFS link. Via IPFS API's resolve function, we then request the multihash of the IPFS link's target.

A method that searches an IPFS object's link list for a special link name is implmented as `getLinkofObject`. The method expects the link's name and the hash of the object which contains that link. It requests the JSON representation of the object and then return the target hash of that link, which is determined by traversing the link list.

### 5.5 Adding Prototype Expressions

ProtoIPFS offers two ways of adding prototype expressions to the system. *File Import* for a big amount of prototype expressions and *Separated Import* for deriving from already existing prototype expressions.

**File Import** The first way allows the user to import prototype expression by handing over a file. This file contains several prototype expressions, with links to other prototype expressions from that file. In other words, a correctly defined file describes a closed system.

The file's format can be found in fig. 19. The format defines that two consecutive prototype expressions are separated by an empty line. The expression itself contains several lines describing the expression's ID, base, and add and remove sets. Each ID which is used here is an IRI. ID's which are used as a base or inside the add or remove set, have to be the ID of a prototype expression from the same file. An exception is `proto:P_0` which must not be defined in this file,

```
ID:
<Prototype Expression ID id>
BASE:
<Prototype Expression  ID base> [-mutable]
ADD:
<Link name> <Prototype Expression ID add 1> [-mutable]
...
<Link name> <Prototype Expression ID add n> [-mutable]
REMOVE:
<Link name> <Prototype Expression ID rem 1> [-mutable]
...
<Link name> <Prototype Expression ID rem m> [-mutable]

ID:
...
```

**Fig. 19.** The format of an input file which contains several prototype expressions.

but is still allowed to be used as a base. Base, add and remove links can also be defined to be mutable by using the optional flag `-mutable`.

After ProtoIPFS read the file in, it is not necessarily possible to add the prototype expressions in the given order. A prototype expression can only be added to the IPFS DAG, when its dependencies (base, add and remove) have already been put on the DAG, before. The reason for this is that ProtoIPFS needs the context based hash of a change expression's value, when it should refer to it. This hash is created when we put the object on IPFS. Therefore, it is necessary to find an ordering of putting prototype expressions. To make this easier, we consider prototype expressions as a graph: Every prototype expression itself is a node on this graph. Its base or add and remove change expression's values are directed edges from the prototype expression to the target expression. The first item we can put on IPFS is that one which has neither a base nor an add or remove expression, meaning no outgoing edges. This holds for `proto:P_0`. So we put `proto:P_0` on IPFS first and get its hash. After that, ProtoIPFS can put all prototype expressions which contain links to already added prototype expressions. Inductively we can add all prototype expressions from the file as long as there is no inheritance cycle in the file. Since such cycles are not allowed in prototype expression's bases, in this case ProtoIPFS throws an error. Property cycles, like prototype expression *Mars is neighbor of Earth and Earth is neighbor of Mars*, have to be constructed by a separated import with IPNS links (cf. fig. 21).

The required insertion order can be found by computing the so-called topological order of the DAG. In fig. 20 we see a directed acyclic graph and its topological order below. Applied on our problem, ProtoIPFS would add all prototypes expressions from the right to the left side.

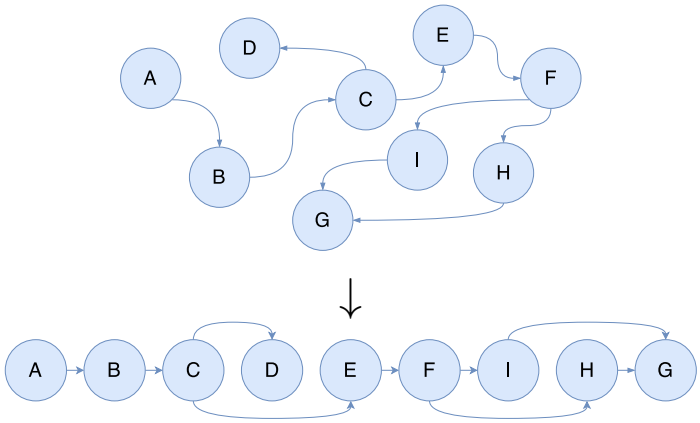


**Fig. 20.** A topological sorting of a DAG. Applied on prototype expressions, D and G would be `Proto:P_0`.

ProtoIPFS uses a variation of the Deep First search algorithm (DFS), which is described in "Introduction To Algorithms" [15], to compute the topological ordering. When a vertex is marked black by the DFS, the algorithm inserts it into the front of a linked list, which is in the end the lower representation of fig. 20. For a graph $G = (V, E)$, with $V$ the vertices and $E$ the edges, we can calculate the topological ordering in $\Theta(|V| + |E|)$.

After defining the order in the `Parser` class, ProtoIPFS uses the mapping from section 4 and puts the expressions on IPFS, each treated as a single separated import. How this works is mentioned in the next section. In the end, the user gets a list of `PrototypeReference` objects back, which can be used to link to these prototype expressions in an immutable or mutable way.

**Separated Import** As already mentioned, the file import does not allow references to already put prototype expressions, since all used IDs (except of `proto:P_0`) must be defined in the same file, too. One advantage of prototypes as knowledge representation is the ability of reusing already existing prototype expressions, though. To make this also possible in ProtoIPFS, a second kind of putting expressions can be found: The user can define new prototype expressions via `PrototypeExpression`'s builder class. The user needs the link of a prototype expression to use it as a base or in a change expression. Depending on the case whether it should be a mutable or an immutable link, the user has to hand an `IpnsLink` or an `IpfsLink` over. The expressions, which are referred to, do not have to be published by the same user who publishes the newly constructed prototype expression. Instead, it is also possible to use a prototype expression from another peer to derive a new prototype expression. After putting a prototype expression, the users gets its `PrototypeReference`.

In the end, independent form whether we used the file import or the separated import, the users have to publish the newly constructed prototype expression

themselves by calling `publishChanges`. This does not need to happen directly after constructing each prototype expression. Several constructed prototype expressions can be grouped and published together (cf. section 6.5). The publish call has to be done at least right before you resolve a mutable link to a newly constructed prototype expression.

**Distinction between Seperated Import and File Import** One could argue that two kinds of adding prototype expressions is too confusing for the user and therefore it might be beneficial to integrate the features of a separate input into the file input. But still, ProtoIPFS offers these two ways of adding, because only then a file always defines a closed system. It is independent from other prototype expressions which are put (and published) on ProtoIPFS. Then, exchanging prototype expressions with other implementations might be possible if future implementations are able to handle the ProtoIPFS' input format.

### 5.6 Consistency Check

As already defined, a consistent knowledge base has to satisfy several conditions. As a recap these conditions are listed below:

- `proto:P_0` is not allowed in the knowledge base
- every prototype expression's ID inside the knowledge base must be unique in the knowledge base's prototype expression set.
- every prototype expression, which is a base or inside the add set of a knowledge base's expression, must be inside as well.
- no cycles in base links are allowed.

Now, ProtoIPFS checks exactly these four conditions when it runs a consistency check. Whenever a condition is not satisfied, ProtoIPFS throws an error. Before these tests are possible, we have to keep in mind that we look at mutable knowledge representation. Since changes of mutable links during a consistency check could influence the result, ProtoIPFS has to create an immutable version of this knowledge base, first. Therefore, when ProtoIPFS downloads the knowledge base, all mutable links are converted to immutable ones. On this fixed set, the consistency test and the fixpoint computation can be applied. This implies that the user can be sure that a consistency checked knowledge base is still consistent, when the user applies the fixpoint computation, afterwards. Let us have a look at the conditions, which need to be checked:

The first condition can be checked by traversing the list of prototype expressions inside the knowledge base in linear time. If an expression equals $P_\emptyset$, then the first condition is not satisfied. The method uses the advantage of permanent, content based hashes: Instead of checking whether the prototype expression does really have all characteristics of $P_\emptyset$, ProtoIPFS only checks whether a prototype expression's hash equals the hash of $P_\emptyset$.

The second condition is not needed to be checked explicitly, since we define the IPFS hash to be the ID of a prototype expression. Since this hash is content

based, two different objects are always described by two different hashes, when we assume that there are no hash collisions.

To check the third condition, ProtoIPFS uses the idea of M. Cochez's prototype implementation, which has already been presented as related work [14]: For each knowledge base's prototype expression, ProtoIPFS checks whether all prototype expressions from the add set and the base expression are inside this knowledge base, too. Note, that this is not done recursively. Since the algorithm iterates over all prototype expression in the knowledge base, the expressions from the add set and the base will be checked, anyway. Therefore, no prototype expression's add set and base is checked twice which is more efficient than a recursive approach.

The last condition meant to be checked is the cycle check. This is done by constructing a graph containing prototype expressions as nodes and base links as directed edges. Via a slightly changed topological sorting, which has already been used in section 5.5, cycles can be detected. Add and remove sets will not be cycle-tested since, according to the definition, cycles are not forbidden, here. One could argue that the whole cycle test is not necessary, since IPFS stores all its objects in a DAG, where cycles can not appear. But remember that ProtoIPFS uses the IPNS service for implementing mutable links. These mutable links can be used to create cycles. An example, which illustrates why cycles are not possible with immutable but are with mutable links, is shown in fig. 21.

### 5.7 Fixpoint computation

As already mentioned, it is possible to interpret a prototype expression with respect to a prototype knowledge base. Before ProtoIPFS calculates a knowledge base's fixpoint, consistency should be checked, first. After that, ProtoIPFS proceeds as follows:

For each knowledge base's prototype expression, the algorithm calculates exactly one prototype. First, ProtoIPFS needs to get the base's fixpoint of the prototype expression. If there is already a fixpoint representation of the base, then the new prototype is built by deriving all properties from the base except those which are also in the prototype expression's remove set. Change expression ID as well as the value of the change expression have to match here. Change expressions which should be removed but are not a property of the base's fixpoint are ignored. After that, ProtoIPFS adds the properties from the prototype expression's add set, and is done with this prototype. In the case that there has not been computed a fixpoint representation of the base yet, this will be calculated, first. An exception is the base prototype expression `proto:P_0`. In this case the fixpoint can be constructed directly, without deriving any base's properties. Otherwise, this is done by a recursive call of the same algorithm and by saving the resulting fixpoint afterwards, such that it can be reused later and does not have to be calculated, again.

The algorithm's output is a set of `Prototype`s but can also be exported as described in fig. 22. The output contains the ID of the prototype, its IRI as description and its properties. Note, that the ID is defined as the IPFS hash of

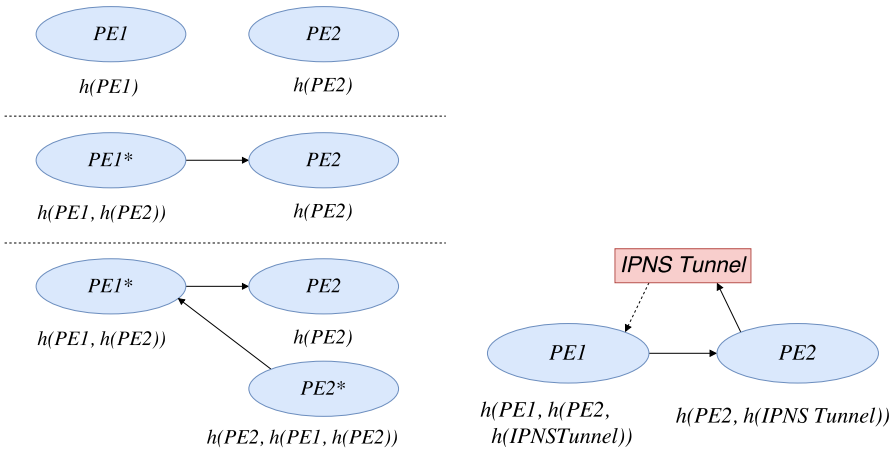


**Fig. 21.** $h$ is the hash function which is used to generate the IPFS object's hash. And we assume that there are no hash collisions. Parameters of this function are components of the IPFS object, which define the hash. On the left hand side there is a visualization why cycles can not occur when immutable links are used only. The two prototype expressions *PE1* and *PE2* have no links first, therefore only the content defines their hash. Then *PE1* adds a link to *PE2*, which changes the hash of *PE1* to *PE1**. The hash of *PE1** includes the hash of *PE2*. Now, we try to define a cycle by adding a link from *PE2* to *PE1**. Then, *PE2**'s hash inlcudes the hash of *PE1**, which is the reason why $h(PE2) \neq h(PE2^*)$. Since *PE1** continues to link to *PE2*, there is still no cycle. On the right hand side, we see a mutable link via an IPNS tunnel. Since the resolution of the IPNS link does not influence the hash of the IPNS tunnel, a cycle can be created here. Therefore, for base links this has to be tested during a consistency check.

the former prototype expression. But still, we use the IRI as a descriptor of the output format, because it often helps to find out what a prototype represents.

```
HASH-ID:
<Prototype Expression IPFS hash>
DESCRIPTION:
<Prototype Expression ID>
PROPERTIES:
<Link name> <Prototype hash-ID 1>
...
<Link name> <Prototype hash-ID n>

HASH-ID:
...
```

**Fig. 22.** The output file format of a fixpoint computation.

### 5.8 Local Storage Management

As a recap, the local storage is managed by the garbage collection. In the current state of IPFS the garbage collection has to be called manually. A mechanism, which calls it automatically, has already been implemented[11], but is disabled in IPFS by default. In IPFS, objects are either pinned or not. Pinned objects stay in the local storage, as long as they are pinned. Not pinned object are removed by the next garbage collection call. The same holds for prototype expressions, since they are realized through IPFS objects. Prototype expressions can be pinned in several ways:

- direct: Only that prototype expression will be pinned on IPFS.
- recursive: A prototype expression and all its transitively linked prototype expressions will be pinned on IPFS.
- indirect: A prototype expression is pinned indirectly if it has been pinned transitively due to a recursive pin operation.

**Manage pinned Objects** ProtoIPFS pins every prototype expression, which is published by the node's user indirectly due to a recursive pin of the administrative directory. Therefore, also prototype expression which are a base or a property value of a published expression will be pinned indirectly. Due to the recursive pinning of the administrative directory, ProtoIPFS also pins every prototype expression indirectly, which is part of a published knowledge base.

[11] `https://discuss.ipfs.io/t/performance-batch-requests-and-gc-control/675/2`

If a user wants to delete a knowledge base or a prototype expression from his node, then it is deleted from the administrative directory and then unpinned recursively. All transitive linked objects will be unpinned, as long as they are not pinned indirectly through an other recursively pinned object. Note, that indirect pinning does not work for IPNS links: For example, a user deletes a prototype expression from the administrative directory and there is still a prototype expression which refers to this via IPNS (and it is not indirectly pinned through an IPFS link). Then this mutable link will point to the empty object, afterwards.

Choosing which objects can be unpinned and which need to be pinned additionally is implemented in the following way: Every time we want to publish changes to our IPFS system we pin the updated administrative directory, first. After that, the old administrative directory will be unpinned recursively. Prototype expressions which are not also in the new directory will be unpinned, since there is no indirect pin for them, anymore. Prototype expressions which are still in the new directory will not be unpinned, because there have been two recursive prototype expressions by which they were pinned indirectly: The indirect pin from the old directory and the indirect pin from the new one. Unpinning the old directory recursively does not affect those prototype expressions since they are still pinned by the new directory. Therefore, the garbage collection can not delete needed prototype expressions, when it is called after this process. The IPFS garbage collection should not run in the automatic mode. Instead, ProtoIPFS can call the garbage collection after updating the directory and reorganizing the pinned objects. Accordingly, ProtoIPFS can organize its local storage on its own.

**Advatages of Recursive Pinning in ProtoIPFS** Due to recursive pinning we also pin prototype expression's bases and property values indirectly, too, even if they are created by another peer's user. The advantage is no dependency on other peers during the fixpoint computation of the prototype expression. This could otherwise fail, when we can't find a prototype expression's base on IPFS, for example because nobody needed that expression anylonger. Additionally, it is possible to access your own knowledge base's prototype expressions faster, since they are stored locally. Another positive side effect is that the problem of persistence, which is not solved by IPFS, is settled for ProtoIPFS, now: Popular prototype expression, for instance, are often used as a reference in other prototype expressions or are included in other user's knowledge bases. Every time someone uses this prototype expression he makes it available on their node due to recursive pinning of the administrative directory. On the other hand, a prototype expression, which is not repinned by other nodes, is only stored in your own local storage. If you unpin it and the garbage collections cleans your local storage, then it will no longer be available on ProtoIPFS. But still, since no one has wanted to use this expression and even the creator himself has not needed it anymore, it might be no loss if the prototype expression will no longer be available.

### 5.9 Handling IRIs as IPFS Link's Name

Remind that a change expression's ID has been defined as an absolute IRI. Applying the defined mapping of section 4, the ID becomes an IPFS object's link name. As we have already seen, IPFS link names have to satisfy a certain norm (cf. section 2.2). Especially for guaranteeing transitive addressing of the form `/ipfs/<hash>/<link-name>/.../<link-name>`, the link's name is not allowed to include the path delimiter, for instance. Nevertheless, the path delimiter might be part of an IRI, which is why we need to encode the reserved link characters, before we use the IRI as an object's link name. By using an URL encoder whenever we put a link's name on IPFS and a URL decoder whenever we request one, we ensure that every IRI can be used as the ID of a change expression in ProtoIPFS.

## 6 Evaluation and Benchmarks

In this section, we want to measure and evaluate how much time certain operations of ProtoIPFS take. Before we can do that, we need to define the testbed, and the data sets which are used during the benchmark tests. Then, we start with tests inside a controlled environment: We look into the construction of prototype expressions and compare two approaches to generate IPFS objects which represent prototype expressions. After that, we compare two ways of requesting a set of prototype expressions from the system. Another important part of ProtoIPFS is publishing the administrative directory, where we also present two different approaches. Of course, we also want to work with prototypes, so we look into benchmarks of creating knowledge bases, investigating its consistency and computing its fixpoint, right after.

After we have looked into the time consumption in a controlled environment and have presented several methods to increase the performance of the system, we regard the time consumption of ProtoIPFS tasks on the global IPFS network.

### 6.1 Testing Composition

The results are obtained by running several versions of ProtoIPFS' `Benchmark` class on an Intel Core i5 @ 2,6 GHz, having 8 GB 1600 MHz DDR3 and SSD hard drive[12]. Except for the last test, ProtoIPFS runs in a controlled environment, which only contains two connected IPFS nodes, each bound to one ProtoIPFS instance. In a last test we connect this setting to the global IPFS network and compare the resulting time consumption with the benchmarks of the controlled environment. Note, that all IPFS nodes run on the same device as the ProtoIPFS system. Each test runs 40 times and we report the median running time. For the real world application we run the benchmark test a whole day long, such that we include variating local traffic peaks. This is necessary because the amount of traffic, for example at an IXP like the DE-CIX Frankfurt Internet Exchange

[12] `https://support.apple.com/kb/sp704`

Point [27], differs regionally according to the time we request data over a certain route: Traffic peaks are visible around 10 p.m., whereby at 4 a.m. the lowest amount of traffic can be recognized. For these tests we are going to introduce four data set generators in the next section.

### 6.2 Data Sets

For the purpose of evaluation we need several prototype expression data sets, encoded in prototype expression files like they have been defined in fig. 19. We are going to introduce four synthetic data set generators, each having different specifications: First Cochez's *baseline(n)* and *blocks(n)* generators [14] and then the *propline(n)*, and *random(m)* generators. $n$ and $m$ are scaling parameters. Note, that the last two generators are variations of Cochez's data set generators.

First of all, each data set is shuffled after it has been constructed. That prevents ProtoIPFS from taking advantages of the construction order, for example during the process of finding a putting sequence. The *Durstenfeld Shuffle* (Algorithm 235) [19] is used to achieve that. Secondly, we use a certain amount of mutable links as the base or as property values of prototype expressions; 20% of all links are mutable, the others are immutable. Furthermore, if we interpret each of the constructed data sets as a prototype knowledge base, all four generators' knowledge bases will be consistent. Now, let us have a look at where these four sets differ in:

Cochez's baseline($n$) generates data sets which contain prototype expressions without any properties. The Prototype expressions are arranged in a tree structure: The first prototype expression derives from $P_\emptyset$. In the second step ($i = 2$), two other prototype expressions derive from the first one. Inductively, from every prototype expression, which is defined in step $i$, derive two prototype expressions in step $i+1$ until we reach the maximal step limit $n$. Then, our data set contains $2^{n+1} - 1$ prototype expressions ($P_\emptyset$ not counted in).

The second data set generator is blocks($n$): Here, we create $n$ blocks of 1000 prototype expressions, each having one added property. Each prototype expression out of block $i + 1$ derives from a randomly chosen prototype expression out of block $i$. Only in the first block ($i = 1$) each prototype expression derives from $P_\emptyset$.

Different to Cochez, we are going to use remove expressions in our test sets, as well. The resulting data sets of propline($m$) are built exactly like baseline($m$), but they have properties which are added or removed. A prototype expression will have up to nine entries inside the remove and add expression set, each. Property values are randomly chosen from a pool of twenty prototype expressions. These twenty prototype expressions are additionally part of the data set.

The fourth type of generators is random($m$) which is comparable to Cochez's incremental data set generator: Here, we create $m$ prototype expressions, each deriving from a random prototype expression which has already been created before. Again, every prototype expression has up to nine add and up to nine remove properties, which are randomly chosen from a pool of twenty prototype expressions, used as property values. Again, these prototype expressions are

additionally part of the data set. An overview of the data sets' characteristics can be found in table 1.

| Data set | Prototype Expressions | | | Change Expression per PE | | |
|---|---|---|---|---|---|---|
| ba(10/12/13) | 2047 | 8191 | 16383 | 0/0 | 0/0 | 0/0 |
| pro(10/12/13) | 2067 | 8211 | 16403 | 4.5*/4.5* | 4.5*/4.5* | 4.5*/4.5* |
| blo(1/5/10) | 1000 | 5000 | 10000 | 1/0 | 1/0 | 1/0 |
| rand(1/5/10) | 1020 | 5020 | 10020 | 4.5*/4.5* | 4.5*/4.5* | 4.5*/4.5* |

**Table 1.** Amount of prototype expressions and properties per prototype expression in each data set. $*$ marked values are average values. The actual values are in $[0, 9]$.

We introduced several data set generators each with its own use cases. Comparing baseline and propline data sets allows us to find out how strong the presence of properties influence certain functionalities. Comparing random and block data sets allows to test how the needed time is influenced by the data set's structure. Blocks and baseline allow to compare ProtoIPFS to Cochez's implementation since these set generators have been used there, too.

All in all, we use test data which is rather small for the use case of knowledge representation. There are two reasons for this: First, during the development it has become clear that IPFS calls are very time consuming, hence huge data will generate infeasible runtime during the benchmark tests. Second, ProtoIPFS is limited to a 2MB maximum IPFS object size. Huge data sets result in huge IPFS administrative directory objects, which become bigger than 2MB when we deal with prototype expression sets in the scale of baseline(14) (32767 prototype expressions) or larger. But still, there is a possible workaround to allow bigger data sets in the future work section 7.1.

### 6.3 Add Prototype Expressions to ProtoIPFS

A time consuming task during the construction of prototype expressions is putting the huge amount of IPFS objects onto the network. A first approach used the IPFS `object.patch` function, which iteratively allows us to create new IPFS objects by changing alredy existing one. With each patch call we can add or remove a link, or set the IPFS object's data.

In a second approach the amount of API requests decreased by using the IPFS function `object.put`. In this case whole IPFS objects have to be defined as JSON objects first, which then are added to the local storage by calling the IPFS put function once per object. Table 2 shows both approaches' time consumption in comparison.

At first sight, we see that using the put method decreases the needed time by roughly $\frac{1}{3}$ in comparison to the patch approach. The reason for that might be less API calls to the IPFS daemon. Consequently, ProtoIPFS uses the put

| Data set | patch (seconds) | | | put JSON object (seconds) | | |
|---|---|---|---|---|---|---|
| ba(10/12/13) | 17.61 | 76.57 | 153.19 | 5.345 | 21.810 | 44.722 |
| pro(10/12/13) | 53.45 | 210.64 | 420.72 | 19.607 | 77.453 | 152.094 |
| blo(1/5/10) | 10.17 | 62.83 | 127.83 | 3.061 | 21.680 | 47.070 |
| rand(1/5/10) | 26.94 | 130.26 | 258.1 | 9.537 | 46.644 | 93.019 |

**Table 2.** Add prototype expression file with the given data set schema, comparing the patch and put approach.

approach whenever we create prototype expressions. Another pattern we see, is that each baseline test needs less time than the according propline test and each blocks test needs less time than the according random test. The reason for that is the additional amount of change expressions in the propline and random sets. For each change expression, proto:SEVERAL objects have to be constructed. Additionally, the proto:ADD and proto:REMOVE objects have to be patched more often in the patch approach when properties are present. A third pattern which we can observe is the roughly linear dependency of the computation time on the amount of prototype expressions in both approaches.

### 6.4 Expensive Mutable Links

In the following test, we are going to look at requesting prototype expressions, putting prototype expressions and creating a knowledge base. Once with a data set where one out five prototype expressions are addressed trough mutable links, and where 20% of the property values and base references are specified using a mutable link. And once with a data set which only contains immutable links. The results can be found in table 3. We only regard random($n$) data sets here. The reason for this is that testing the overhead of mutable links is not that much dependent on the structure of the data set, but rather on the amount of prototype expressions dressed.

| Data set | request data set (in s) | | put data set (in s) | | create KB (in s) | |
|---|---|---|---|---|---|---|
| rand(1) | 9.241 | 21.81 | 8.56 | 9.537 | 0.085 | 0.335 |
| rand(5) | 46.234 | 141.872 | 41.948 | 46.644 | 0.449 | 1.777 |
| rand(10) | 92.583 | 367.106 | 83.709 | 93.019 | 0.897 | 3.593 |

**Table 3.** Comparison of consumed time of three functions without any mutable links (left) and a data set with 20% mutable links and where 20% of the prototype expressions are addressed via those mutable links (right).

First of all, we again see a trend towards a linear dependency of put's and create KB's time consumption on the size of the used data set. In general, we

see that time consumption is much higher when we deal with a data set which is addressed via mutable links and also contains mutable links in its simple change expressions. When requesting a data set the reason for this is that we need to resolve IPNS links to reach 20% of the prototype expressions. It has been clear early that the IPNS resolution takes a lot of time. In the other two cases, the reason for the higher time consumption is that we need to generate additional IPNS tunnel objects to store mutable links while constructing prototype expressions or knowledge bases. These objects also need to be put which increases time consumption. We can conclude from this test, that mutable links should only be used when it is really necessary. This also leads to the assumption that IPFS is not the most suitable environment for huge mutable data.

### 6.5 Publish Directory

As already mentioned, ProtoIPFS uses IPNS to realize mutable links and knowledge bases. A peers' IPNS pointer is set via calling IPFS' `name.publish`. In a first approach this method was called after every change which had been made to the administrative directory. For ProtoIPFS this means that after every added prototype expression and every change on a knowledge base the method has to be called. We call this approach *direct publishing*. A problem here is that publishing a new IPFS object on a peer node takes some time.

An approach, which may make adding or changing prototype expressions and knowledge bases more efficient, groups several changes together. We call this approach *batch publishing*: Instead of putting and publishing the latest version of the administrative directory after every update, ProtoIPFS updates the directory only locally and publishes the resulting directory not until all changes in the batch have been made. For instance, when we add a file of one hundred prototype expressions to ProtoIPFS, then there would still only be one call to put and one to publish the new directory, instead of one hundred put and one hundred publish calls in the first approach. An imaginable race condition, which might happen because ProtoIPFS does not publish the changed directory immediately, is prevented by the fact that only one ProtoIPFS client is allowed for each IPFS peer node. Since the IPNS directory can not be changed by other participants except the host of the node itself, race conditions are not possible. To enable advantages of the approach on a larger scale too, ProtoIPFS includes a functionality which allows the user to decide manually when the IPNS directory should be updated. In ProtoIPFS, the directory is represented by the `Directory` class, which manages the changes to the directory locally until `publishChanges` gets called. After calling `publishChanges` the updated version of ProtoIPFS is available for other nodes on the network, of course depending on the time which is needed for the IPNS update to be propagated on the network. In table 4, one can find a comparison of both presented approaches.

As expected, we see that the time consumption decreases enormously when we use the batch publishing approach. Whereas using the direct publish, testing with large sized sets is so time consuming that it became unreasonable to continue measurements.

| Data set | direct publish* (minutes) | | | batch publish (ms) | | |
|---|---|---|---|---|---|---|
| ba(10/12/13) | 2.34 | 33.00 | - | 121 | 476 | 953 |
| pro(10/12/13) | 2.39 | 34.11 | - | 133 | 484 | 958 |
| blo(1/5/10) | 0.64 | 12.50 | - | 59 | 294 | 585 |
| rand(1/5/10) | 0.69 | 12.99 | - | 61 | 296 | 585 |

**Table 4.** Publish a prototype expression file with the given data set schema. Visible are the direct publish and batch publish method. * denotes that this is only the median of twenty measurements. - denotes that measuring the value is not feasible.

We additionally see that the batch publishing time does not strongly depend on the amount of properties, since baseline's and propline's time consumptions are nearly equal. This also holds for comparing blocks and random data sets. Their structural differences as well as random's additional properties do not influence the batch publish time. On the other hand, the direct publish approach experiences higher time consumption, due to its additional 20 prototype expressions in the data set. In the batch approach we additionally recognize a trend towards a linear dependency of the time consumption on the data set's size.

### 6.6 Knowledge Base Benchmarks

Next, we will look into operations on prototype knowledge bases: First, into the construction time of a prototype knowledge base, based on already put and published prototype expressions. Then, into the consistency check of an already requested knowledge base, and in the end, into the computation of the knowledge bases' fixpoint. Again, we choose 20% of all links of the knowledge base to be mutable. The measured results can be found in table 5.

| Data set | create KB (in s) | | | consistency (in ms) | | | fixpoint (in ms) | | |
|---|---|---|---|---|---|---|---|---|---|
| ba(10/12/13) | 0.707 | 2.861 | 5.816 | 170 | 677 | 1352 | 97 | 386 | 780 |
| pro(10/12/13) | 0.732 | 3.031 | 5.953 | 358 | 1423 | 2928 | 477 | 1952 | 3808 |
| blo(1/5/10) | 0.372 | 1.612 | 3.219 | 97 | 506 | 1013 | 64 | 335 | 678 |
| rand(1/5/10) | 0.335 | 1.777 | 3.593 | 172 | 845 | 1689 | 225 | 1122 | 2326 |

**Table 5.** Several knowledge base operations.

At first sight, we recognize that consistency and fixpoint computations are way faster than creating a prototype knowledge base, and also scale on larger sets well. The reason for this is that these computations are done locally and mostly offline on an already downloaded prototype knowledge base. Still, when we compare the ProtoIPFS' consistency checking and fixpoint computation with

Cochez's implementation (cf. table 6), we see that Cochez's implementation is less time consuming. The reason for this is that the hashes of base references and property values have to be resolved during these computations in ProtoIPFS. Even though the prototype expressions are locally stored we still need to do expensive API calls to get their interconnections. The resolution of property value hashes also implies why in ProoIPFS the property-sparse baseline and blocks sets have faster consistency and fixpoint computations than the propline and random sets.

When we look at creating a knowledge base in ProtoIPFS, we have to interact with the IPFS daemon, even more often. Especially mutable prototype expressions inside the knowledge base increase the amount of API calls, since we need to put a new IPNS tunnel object for each mutable reference. Additionally, we need to put the actual knowledge base object, which holds links to all included prototype expressions. Here, we see that the time consumption of block and random, as well as baseline and propline data sets are quite similar. Therefore, the amount of prototype expression's properties is not influencing the time consumption of creating a knowledge base. That was expected, since we do not regard a prototype expression's properties when we create a knowledge base. We just need the prototype expression's link.

| Data set | consistency (in ms) | | | fixpoint (in ms) | | |
|---|---|---|---|---|---|---|
| ba(10/12/13) | 170 / 4 | 677 / 10 | 1352 / 21 | 97 / 8 | 386 / 17 | 780 / 34 |
| pro(10/12/13) | 358 / 6 | 1423 / 25 | 2928 / 55 | 477 / 26 | 1952 / 96 | 3808 / 203 |
| blo(1/5/10) | 97 / 0 | 506 / 7 | 1013 / 18 | 64 / 1 | 335 / 14 | 678 / 40 |
| rand(1/5/10) | 172 / 2 | 845 / 15 | 1689 / 33 | 225 / 8 | 1122 / 57 | 2326 / 122 |

**Table 6.** Comparing time consuption of knowledge base operations in ProtoIPFS (first) and in Cochez's portotype implementation (second). The results of Cochez's implementation are gained by importing the newly defined data set generators propline($n$) and random($m$) to Cochez's implementation. Note, that Cochez's consistency check also includes a part of the construction of a knowledge base. The results of Cochez's implementation are the average values of 40 tests, whereby the values of ProtoIPFS are copied from table 5.

### 6.7 Real World Application Benchmarks

Now we look at how ProtoIPFS behaves in a real world setting. As already mentioned this test ran 24 hours, to reduce the influence of varying throughput. The results of the test can be found in fig. 23.

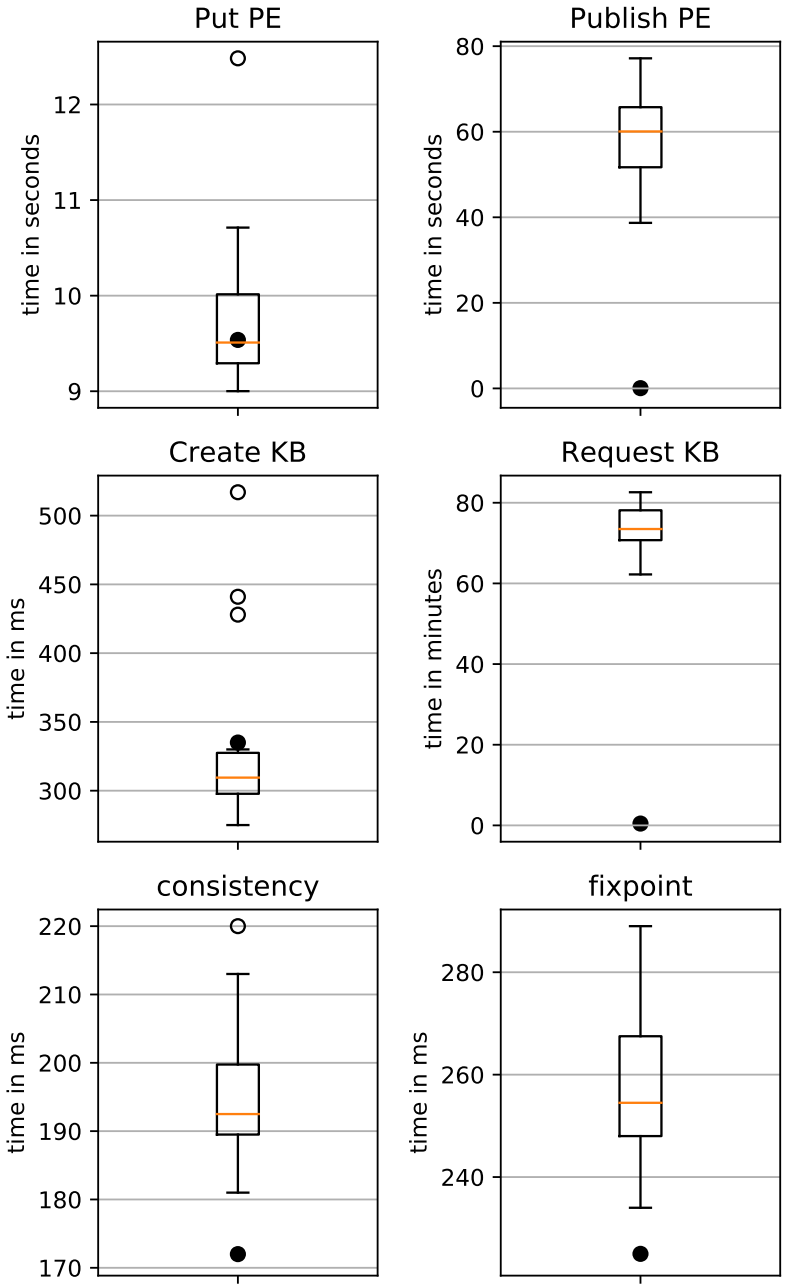


**Fig. 23.** The time consumption of several ProtoIPFS operations during the real world application test. A random(1) data set is used here. • denotes the median of the time consumption which has been measured in the controlled environment test, before. The median is represented by a green line. The box describes the upper quartile with its upper edge and the lower quartile with its lower edge. The whiskers denote the maximum ($\top$) and the minimum values ($\bot$) of our samples, except for outliers, which are denoted by ∘. More about boxplots can be found in [31].

We see that put prototype expressions and create knowledge base operations are nearly equally time consuming compared to their counterparts. Therefore, the online setting does not influence the time to generate prototype structures. Both operations face some outliers. It seems that the only explanations are either variations in the speed of the network or variations in the speed of the IPFS P2P system.

The publish administrative directory call on the other hand is way more time consuming in an online setting than in the closed environment. The reason for the increase from 61 ms to 1 minute might be the larger underlying DHT over which IPNS distributes its data. The same holds for requesting a knowledge base where we have to resolve a certain amount of IPNS links (20% of all links due to the choice of our data set). With a larger DHT the crawl time for IPNS information increases as well. Here, from 27 seconds to one hour on average and up to nearly 80 minutes in our test's worst case.

While testing consistency and computing the fixpoint, we can see a small increase in the magnitude of several 10ms, in comparison to the offline test. The reason for that might be the needed requests to resolve immutable links from one prototype expression to another one. Except for these resolutions, every other computation of these two functionalities is done offline, without interacting with the IPFS daemon.

All in all, we gained 21 samples in these 24 hours, whereby we experienced three resolution failures when we requested a knowledge base. Likely, the reason for this is that the target's peer ID had been resolved before its latest administrative directory publish reached the requesting peer. Therefore, the requested knowledge base could not be addressed. Longer waiting before requesting the knowledge base would likely have fixed this issue. More about the difficulties of finding good timeouts for requesting IPNS links can be found in section 7.4.

### 6.8 Reachability of Prototype Expressions

Since prototype expression are realized as IPFS objects, reachability has been discussed briefly in section 2.2. The reachability of protoype expressions and knowledge bases depends on the the available nodes of your swarm. A prototype expression is reachable via an immutable link as long as it is stored in at least one available node of your swarm. On the other hand, a prototype expression is reachable via a mutable link only if the peer which controls the mutable link is available in your swarm. The same holds for knowledge bases, where we request the knowledge base object from a certain peer.

### 6.9 Summary

From the benchmarks we can conclude that we gained much more efficiency due to the introduction of batch publishing and the use of IPFS' put service instead of the patch approach. Additionally, we have seen that mutable links are expensive and should only be used if they are really necessary. Nevertheless, we have also seen that especially requesting prototype expressions does take

infeasible time even on a small data set in a real world application. Unfortunately, the time consumption of IPNS resolutions will always be present, as soon as we need to address a mutable prototype expression. But still, we will look into several improvements which may increase efficiency of the whole system, in the next section. Reasonable are the time consumptions of consistency and fixpoint computations in a real world and in the closed environment.

## 7 Future Work

In this section we look into several aspects which can be added to ProtoIPFS, to gain more usability and more efficiency. First, we deal with the 2MB size limit of IPFS objects, and how to handle prototype expressions or directories which have a larger size. Then, we present an approach to achieve exclusive usage of a ProtoIPFS' bounded daemon. Because when we introduced ProtoIPFS, we assumed exclusive usage to guarantee consistency of the ProtoIPFS system, but these mechanisms have not been implemented, yet. After that, we look into the topic of querying, as an important aspect of a knowledge representation system. The next part deals with the overall problem of IPFS' efficiency. In the benchmark section (cf. section 6), we have seen that ProtoIPFS does not scale well in a real world swarm. Therefore, we will describe A) needed improvements of the underlying IPFS system and B) possibly more efficient mappings from Prototypes to IPFS. We will further describe how these approaches affect the current ProtoIPFS design.

### 7.1 Limited Object Size

According to Johnson, who is a core member of the IPFS development community, the size limit of an IPFS object is 2MB[13]. We could easily confirm this by trying to put an object which is larger than 2MB. The limitation is currently also a restriction for ProtoIPFS: Not only for the size of a prototype expression, but also for a knowledge bases' amount of included expressions. Remember that a knowledge base is represented by an IPFS object which has links to all included prototype expressions. Even worse, also the administrative directory object, which holds links to all node's prototype expressions, is an IPFS object. The amount of prototype expressions which we can publish is therefore likewise limited by the maximum size of the administrative prototype directory object. Johnson already mentioned, that larger objects will be supported soon. Until then, ProtoIPFS could split the directory object, every time when it reaches the maximal size. A newly created root object links to these two objects, then. The same approach is used on the IPFS File layer (cf. section 2.2), to allow files far bigger than 2MB in the distributed file system. Since we work on the lower IPFS Object layer, we can not simply use the functionality of the IPFS File layer to solve the problem.

[13] `https://discuss.ipfs.io/t/file-systems-chunk-small-files-big-files/343/3`

### 7.2 Locking IPFS Daemon Access

An assumption we have made right in the beginning allows only one ProtoIPFS instance to run on each IPFS node, since otherwise multiple instances could influence each others. Currently, there is no realization of restricting multiple binding to one node, but one could imagine to handle the situation like this: To mark whether an IPFS daemon is already bound to a ProtoIPFS instance, an additional flag object in the administrative directory can be used. This object is added to the directory as soon as a ProtoIPFS instance binds to the daemon and which is removed when the instance unbinds. When a second instance of ProtoIPFS tries to bind to an already bound daemon, then it would notice the flag object and stops the binding process, immediately.

Next to other ProtoIPFS instances' bindings, API calls from other programs might cause inconsistencies, too. Imagine for instance a program which works with the same IPFS daemon and uses its IPNS service to publish a different object than ProtoIPFS. This would overwrite ProtoIPFS' administrative directory, and make knowledge bases and mutable links unavailable. Controlling these kind of IPFS accesses is not possible with the previous approach, since these programs can simply ignore the flag object. To solve this issue, access restriction has to be done elsewhere, for example inside the IPFS daemon by restricting the resources which are allowed to send HTTP API requests.

### 7.3 Querying in ProtoIPFS

The current version of ProtoIPFS allows us to reuse prototype expressions, as soon as we know their hashes. But users also have to find these hashes first, before they can use them as the base or as a property value of their new prototype expression. Finding suiting prototype expressions (and their hashes) can be made possible by introducing query mechanisms. Daswania [16] distinguishes several types of querying for peer-to-peer systems. In [33] these categories are grouped into *schema-based*, *keyword-based* and, through IPFS hashes already integrated, *key-based* querying.

As we have already suggested in our paper about Prototypes and IPFS [12], crawlers, well known from search engines and the related SWARMLinda approach, can be used to answer queries about knowledge bases or prototype expressions: A crawler would look at each ProtoIPFS instance of the swarm and resolves their IPNS link, which points to the node's administrative directory. Remember that an administrative directory contains information about the published knowledge bases and all stored prototype expressions. To use query mechanisms we have to compute prototype expression's fixpoints first. On these prototypes we can then search for a query's keywords or its structure. A structure would be a description of a prototype whereby we can use wildcards as property, as property values or as its prototype ID. To fine grain the query's results we can use additional meta data, like it has been done in SWAP (cf. section 3.1). Especially querying for knowledge bases can profit from meta data, since they offer less options of schema- or keyword-based querying than prototype expressions.

To reduce the overhead for a ProtoIPFS node to crawl a knowledge base or a prototype expression we could introduce IPFS nodes whose single task is crawling the network. ProtoIPFS nodes then can send queries to a nearby crawlers who can answer these question faster, due to its already cached data.

### 7.4 Optimal Situation

The IPFS system is still under development, therefore it is clear that IPFS is not fully sophisticated. There are several aspects which need to be improved in the ongoing process of developing, to make ProtoIPFS real life applicable, and IPFS easier to use.

First of all, a better documentation of the whole IPFS project is needed. The current API is sufficient for basic IPFS calls but lacks in exact specifications, for example the limitations of IPFS object's link name can only be found in a GitHub commit. Additionally, it is sometimes not clear what the current state of the art is. Only a github change log[14] or reading the source code gives information about changes, here. Next, an optional object size bigger than 2MB is desirable too, to get rid of the presented workaround.

Then, there are several aspects about the official recommended Java IPFS API, which is used in ProtoIPFS: Next to a missing documentation, we have already discussed misconducts and missing basic functionalities, for instance requesting complete IPFS objects as a JSON object (cf. section 5.4). Additionally, there are aspects which could lower the amount of ProtoIPFS workarounds: To reduce API calls a working implementation of IPFS batch puts would be a good way to decrease the amount of IPFS API calls.

Next to the already mentioned improvements, the two main aspects of IPFS, which have to be enhanced, are:

- Performance of object operations (put and get)
- Performance of IPNS calls (publish and look up)

Putting IPFS object, like we have seen in table 2, is already quite slow for 10000 prototype expressions. Now, if we imagine data sets of several million prototype expressions, then putting will become infeasible.

More problematic is the time needed to publish and resolve IPNS objects, which seems to depend on the swarm size and the connectivity to members of the swarm. Next to the high time consumption, it is unpredictable when an IPNS publish call is distributed on the DHT. It may happen that an IPFS publish has not reached a second node, yet. The directory which this node then tries to resolve might not be up to date or not even exist anymore. Working with a no-cache option of IPFS name resolution fixes this problem in many cases, since we do not cache old resolution entries on our node anymore. Still, due to the way of distributing IPNS as meta data in the routing system, this problem occurs from time to time.

[14] `https://github.com/ipfs/go-ipfs/blob/master/CHANGELOG.md`

A related, general problem of IPFS is choosing timeouts of HTTP API requests. A small timeout is often reached when an object needs to be searched on the DHT or a name resolution is done. A large timeout decreases the speed of the system when an object does not exisit on the DAG, because IPFS will try as long as possible to reach the object. This topic is discussed on the IPFS forum[15].

### 7.5 Alternative Construction

As we have seen during the benchmark tests, a high amount of API calls strongly decreases the efficiency of ProtoIPFS. Currently, in the approach of the thesis we need up to $2a+2r+4$ IPFS objects for each prototype expression, where $a$ is the amount of added properties and $r$ the amount of removed properties. This is the case when we assume that all links are mutable, such that every link needs a tunnel object, there are no change expressions with multiple values, so as for every single property a proto:SEVERAL object is needed. Now, we will look into several approaches of how we could map prototypes on IPFS differently, such that less API calls are needed and time consumption may decrease.

**Single Object Approach** Now, this first approach saves for each prototype expression every of these up to $2a+2r+4$ IPFS objects, and all prototype expressions it refers to, in a JSON Object. This JSON object is then stored in the data part of a single IPFS object. Thereby, we would reduce the amount of put and request calls form $2a+2r+4$ to 1. Additionally, we can compute the fixpoint of this prototype expressions without requesting additional prototype expressions or property values, since all linked prototype expressions are already stored in this IPFS object. But still, this way of construction has several disadvantages: First, there is redundancy due to less reuse of IPFS objects inside the IPFS system. For example, a prototype expression which is the base of two different other prototype expressions would be saved three times: The base as an object itself, and one time in each JSON object of the deriving prototype expression. Additionally, we do not use advantages of IPFS, like the DAG structure which ensures cycle free transitive IPFS link chains and content based hash addressing. Another problem is that the data part of an IPFS object becomes very big. Therefore, the whole IPFS object could easily become bigger than 2MB, which of course can be worked around (cf. section 7.1), but then would lead to several IPFS objects again.

**Light Single Object Approach** This second approach does not save every prototype expression that we refer to in the data part of an IPFS object. Instead, it only saves links of the add and remove change expressions directly inside the IPFS object of the prototype expression. Then the amount of put calls would also drop from $2a+2r+4$ to 1, since we do neither need an own object for the

[15] `https://github.com/ipfs/js-ipfs-api/issues/71`

proto:ADD set, nor the proto:REMOVE set, nor the proto:SEVERAL, and nor the mutable tunnel link objects. Also, the size of a single IPFS object which represents a prototype expression is feasible, since we only store the links to prototype expressions and not, like in the approach before, the whole expression. But still, a disadvantage here is the abuse of the IPFS system due to storing IPFS links in the data part instead of the link set of an IPFS object. Therefore, advantages of indirect pinning and direct traversing disappear.

**Prototypes instead of Prototype Expressions** A last approach of combining IPFS and Prototypes has already been presented in our vision paper of "Prototypes on a Global Distributed File System" [12]. Instead of saving prototype expression in IPFS, we directly look at the result of a fixpoint computation: We save prototypes in IPFS. Then an IPFS object would contain a prototype ID and links to all its property values. For same named properties we still would use a proto:SEVERAL object. Using that method allows us to get prototypes from IPFS without checking consistency and computing a fixpoint, first. Meaning that we do not need to request IPFS objects that often. Additionally, we can directly request property values via an IPFS link of the form `/ipfs/<Prototype-hash>/<Property>`. The introducing example would then look like fig. 24. Notice, that this approach is not in the spirit of Prototypes since we do not reuse information anymore, which was once ensured by the use of prototype expressions. It is more like a set of objects having properties which is the core function of IPFS itself. Alternatively, I could also be set on any distributed file system directly, where every prototype is represented by a single file.

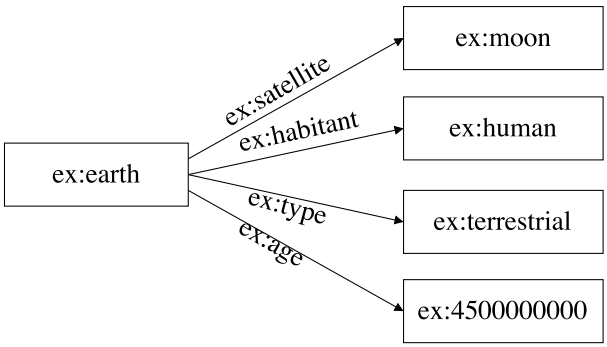


**Fig. 24.** An alternative mapping of prototypes instead of prototype expressions on top of IPFS.

## 8 Conclusion

This thesis presented the prototype knowledge representation system, which allows to derive new information from already existing information. Using prototypes implies a better re-usage of knowledge than in conventional approaches like RDF. After that, we looked into the InterPlanetary File System. IPFS uses content based hashes to address objects in the network and profits from well established systems like GitHub and BitTorrent, which have been adapted in its implementation, as well. The task of the thesis was to combine these two systems to create a distributed prototype knowledge system, which then also allows mutability. Therefore, we introduced a mapping from Prototypes to IPFS. We used IPFS objects as representation of prototype expressions and gained mutabiltiy via IPNS Links in combination with the construction of the administrative directory. Then, we looked into the Java implementation of this mapping: ProtoIPFS. We presented its layer architecture and important design decisions like the project's structure and import/export formats, the usage of graphs to check consistency and efficient fixpoint computation. We have seen that Prototype and IPFS fit well, mostly. But still not every time: At some points (for example IRI as link names) we had to introduce workarounds which allowed us to keep close to the formal definition of Prototypes. After we had described the implementation, we looked into the evaluation of the ProtoIPFS system: Generated data sets were used to benchmark the system in a closed environment, as well as, on the whole IPFS system. We saw that IPFS API calls are quite time consuming, hence we had introduced several adaptations which decreased the amount of API calls and the time consumption. As part of the future work section, we discussed several aspects which can be improved upon increasing the usability of ProtoIPFS. For example, how we could deal with the limited size of IPFS objects and mutual exclusion of the IPFS daemon. But still, the benchmark tests also showed that the mapping and implementation are not scalable to large data sets. This would have been important, since prototype knowledge bases with several hundredthousands of prototype expressions are realistic, for example, when we look into the magnitudes in which Cochez has tested his system [14]. In the end, we must admit that IPFS is still under development. Changes are made quite often. It is likely that the efficiency of IPFS becomes better with ongoing changes. Maybe then, this implementation will be large scalable, as well. Until then, alternative constructions were shown which do not use every feature of IPFS but could gain more performance. A possible application for the current state of ProtoIPFS migth be a company's internal knowledge representation where we have a small amount of nodes connected via high-throughput connections. For that we would set up a private swarm of nodes on which we run the ProtoIPFS instances.

## References

1. Jonas Almeidas. subClassOf. `https://github.com/mathbiol/subClassOf`, November 2015. [Online; accessed 19-July-2017].

2. Anne Augustin. RDF extensions of SwarmLinda. Technical report, Freie Universität Berlin, 2008.
3. Ingmar Baumgart and Sebastian Mies. S/kademlia: A practicable approach towards secure key-based routing. In *Parallel and Distributed Systems, 2007 International Conference on*, pages 1–8. IEEE, 2007.
4. Juan Benet. IPFS - content addressed, versioned, P2P file system. *CoRR*, abs/1407.3561, 2014.
5. Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web. *Scientific American*, 2001.
6. Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data-the story so far. *Semantic services, interoperability and web applications: emerging concepts*, pages 205–227, 2009.
7. Matteo Bonifacio, Paolo Bouquet, and Paolo Traverso. Enabling distributed knowledge management: Managerial and technological implications. Technical report, University of Trento, 2002.
8. Matteo Bonifacio, Roberta Cuel, Gianluca Mameli, and Michele Nori. A peer-to-peer architecture for distributed knowledge management. Technical report, University of Trento, 2002.
9. Matteo Bonifacio, Fausto Giunchiglia, and Ilya Zaihrayeu. Peer-to-peer knowledge management. Technical report, University of Trento, 2005.
10. Dan Brickley and Ramanathan Guha. RDF schema 1.1. W3C recommendation, W3C, February 2014. http://www.w3.org/TR/2014/REC-rdf-schema-20140225/.
11. Jeen Broekstra, Marc Ehrig, Peter Haase, Frank Van Harmelen, Arjohn Kampman, Marta Sabou, Ronny Siebes, Steffen Staab, Heiner Stuckenschmidt, Christoph Tempich, and Stuckenschmidt Christoph Tempich. A metadata model for semantics-based peer-to-peer systems. In *In: Proceedings of the WWW?03 Workshop on Semantics in Peer-to-Peer and Grid Computing*, 2003.
12. M. Cochez, D. Hüser, and S. Decker. The future of the semantic web: Prototypes on a global distributed filesystem. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1997–2006, June 2017.
13. Michael Cochez, Stefan Decker, and Eric Prud'hommeaux. *Knowledge Representation on the Web Revisited: The Case for Prototypes*, pages 151–166. Springer International Publishing, Cham, 2016.
14. Michael Cochez, Stefan Decker, and Eric Prud'hommeaux. Knowledge representation on the web revisited: Tools for prototype based ontologies. *CoRR*, abs/1607.04809, 2016.
15. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction To Algorithms* -. MIT Press, Cambridge, 2001.
16. Neil Daswani, Hector Garcia-Molina, and Beverly Yang. Open problems in data-sharing peer-to-peer systems. In *International conference on database theory*, pages 1–15. Springer, 2003.
17. Stefan Decker. From Linked Data to Networked Knowledge. Slides from CSHALS Conference, available via `https://www.slideshare.net/stefandecker1/stefan-decker-keynote-at-cshals`, 2013. [Online; accessed 21-July-2017].
18. M. Duerst and M. Suignard. RFC 3987: Internationalized Resource Identifiers (IRIs). RFC 3987 (Proposed Standard), see `http://www.ietf.org/rfc/rfc3987.txt`, January 2005.
19. Richard Durstenfeld. Algorithm 235: Random permutation. *Commun. ACM*, 7(7):420, July 1964.

20. Marc Ehrig, Peter Haase, Björn Schnizler, Steffen Staab, Christoph Tempich, Ronny Siebes, and Heiner Stuckenschmidt. SWAP deliverable D3.6 refined methods. `https://web.archive.org/web/20070723063933/http://swap.semanticweb.org/public/Publications/swap-d3.6.pdf`.
21. Marc Ehrig, Peter Haase, Steffen Staab, and Christoph Tempich. SWAP Deliverable D3.5 Method integration. `https://web.archive.org/web/20081006174544/http://swap.semanticweb.org/public/Publications/swap-d3.5.pdf`, 2003.
22. Marc Ehrig, Christoph Tempich, Jeen Broekstra, Frank van Harmelen, Marta Sabou, Ronny Siebes, Steffen Staab, and Heiner Stuckenschmidt. SWAP: Ontology-based knowledge management with peer-to-peer technology, 2003.
23. Michael J Freedman, Eric Freudenthal, and David Mazieres. Democratizing content publication with coral. In *NSDI*, volume 4, pages 18–18, 2004.
24. Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.
25. David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, January 1985.
26. Henri Gilbert and Helena Handschuh. *Security Analysis of SHA-256 and Sisters*, pages 175–193. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
27. DE-CIX Management GmbH. DE-CIX Frankfurt statistics. `https://www.de-cix.net/de/locations/germany/frankfurt/statistics`, 2017. [Online; accessed 29-July-2017].
28. Daniel Graff. Implementation and evaluation of a SWARMLINDA system. *Masterarbeit. FU Berlin*, page 4, 2008.
29. Sebastian Koske. Swarm approaches for semantic triple clustering and retrieval in distributed RDF spaces. Technical report, Freie Universität Berlin, 2009.
30. Jon Loeliger. *Version Control with Git*. O'Reilly Media, Inc., 2009.
31. Kristin Potter. Methods for presenting statistical information: The box plot. *Hans Hagen, Andreas Kerren, and Peter Dannenmann (Eds.), Visualization of Large and Unstructured Data Sets, GI-Edition Lecture Notes in Informatics (LNI)*, S-4:97–106, 2006.
32. Ralf Steinmetz and Klaus Wehrle. *2. What Is This "Peer-to-Peer" About?*, pages 9–16. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.
33. Heiner Stuckenschmidt, Frank van Harmelen, Wolf Siberski, and Steffen Staab. *Peer-to-Peer and Semantic Web*, pages 1–17. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
34. Christoph Tempich and Steffen Staab. *Semantic Query Routing in Unstructured Networks Using Social Metaphors*, pages 107–123. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
35. David Wood, Markus Lanthaler, and Richard Cyganiak. RDF 1.1 Concepts and Abstract Syntax. W3C recommendation, W3C, February 2014. http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.
36. F. Yergeau. RFC 3629: UTF-8, a transformation format of ISO 10646. Technical report, Alis Technologies, 2003.
37. Liangzhao Zeng, Hui Lei, and Badrish Chandramouli. *Semantic Tuplespace*, pages 366–381. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

# A Class Diagrams

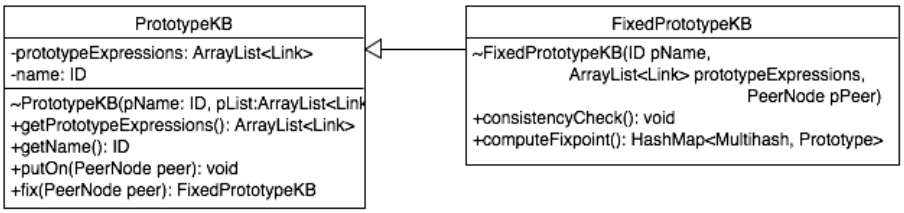


**Fig. 25.** ProtoIPFS knowledge base package. Some, for the user not visible object properties are left out. We construct objects of both classes via Builder classes, which are not visible in the class diagram.

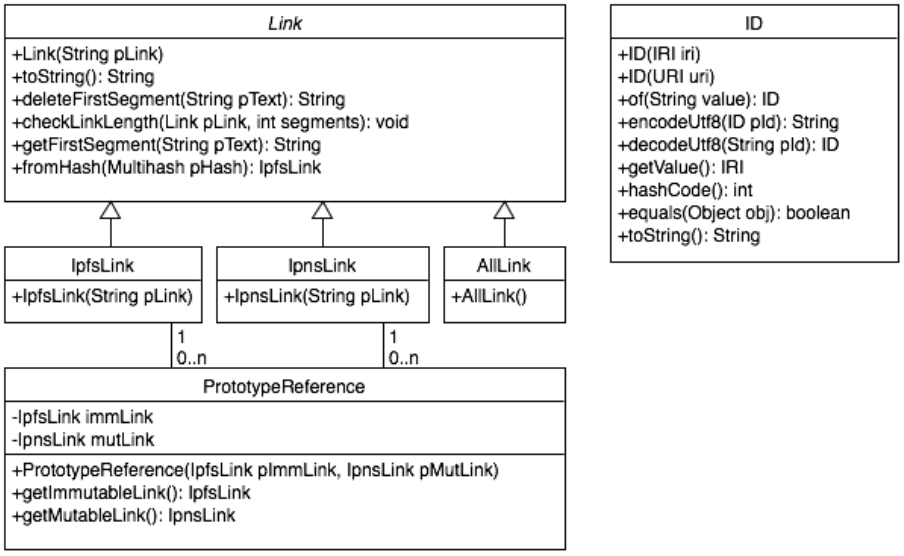


**Fig. 26.** ProtoIPFS link package. Visible is the abstract `Link` class which is extended by the IPFS and IPNS link classes. A pair of pbjects of these two classes form a `PrototypeReference`. The `AllLink` is used to define $*$ in a remove expression. `ID` is a wrapper class for an `IRI` which is used in incoming prototype expressions and names for knowledge bases.

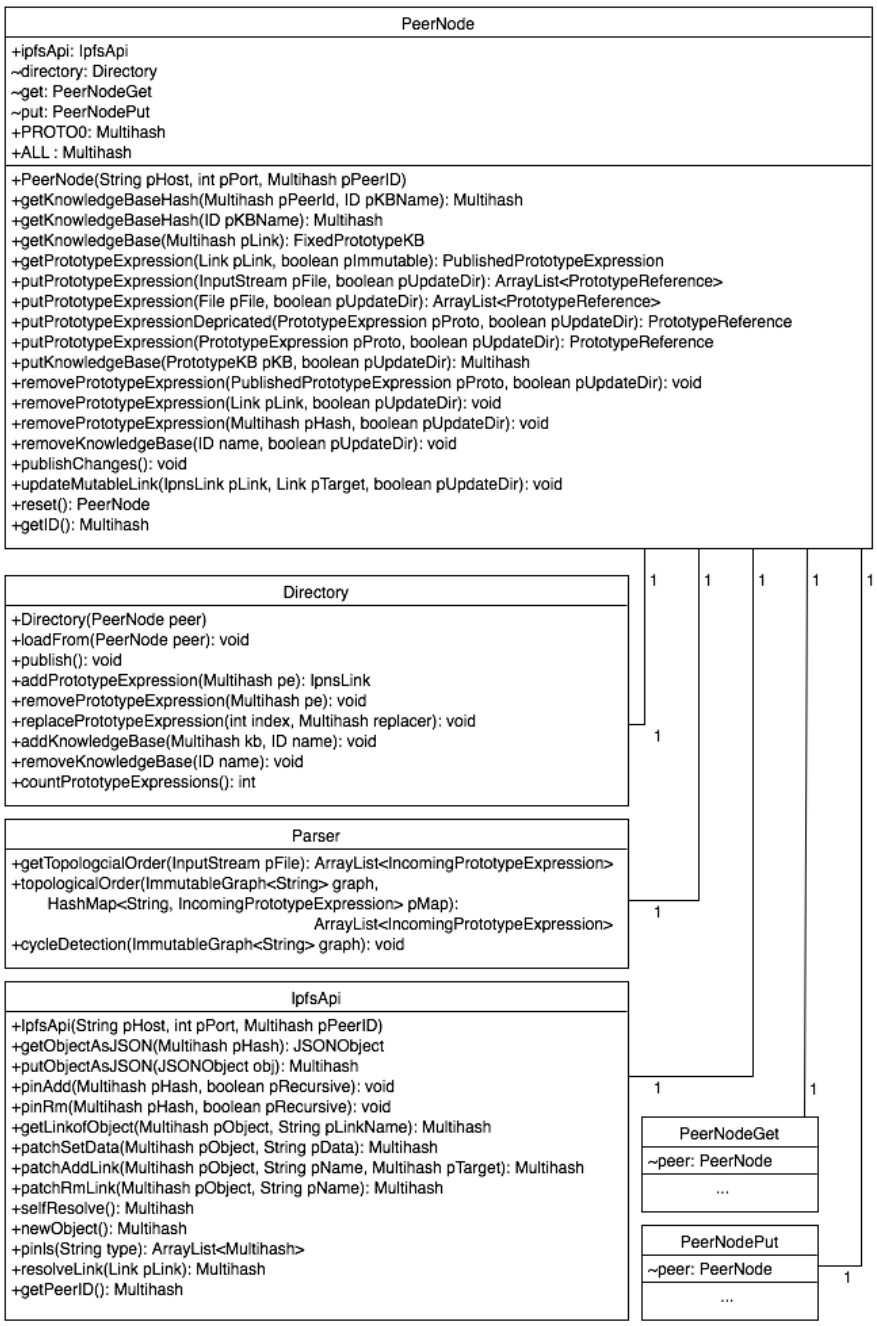


**Fig. 27.** ProtoIPFS peer package. `PeerNodeGet` contains the implementations of `PeerNode`'s get Methods. `PeerNodePut` contains the implementations of `PeerNode`'s put Methods. `PeerNode` calls these methods inside the listed method. We use the delegation design pattern, here.

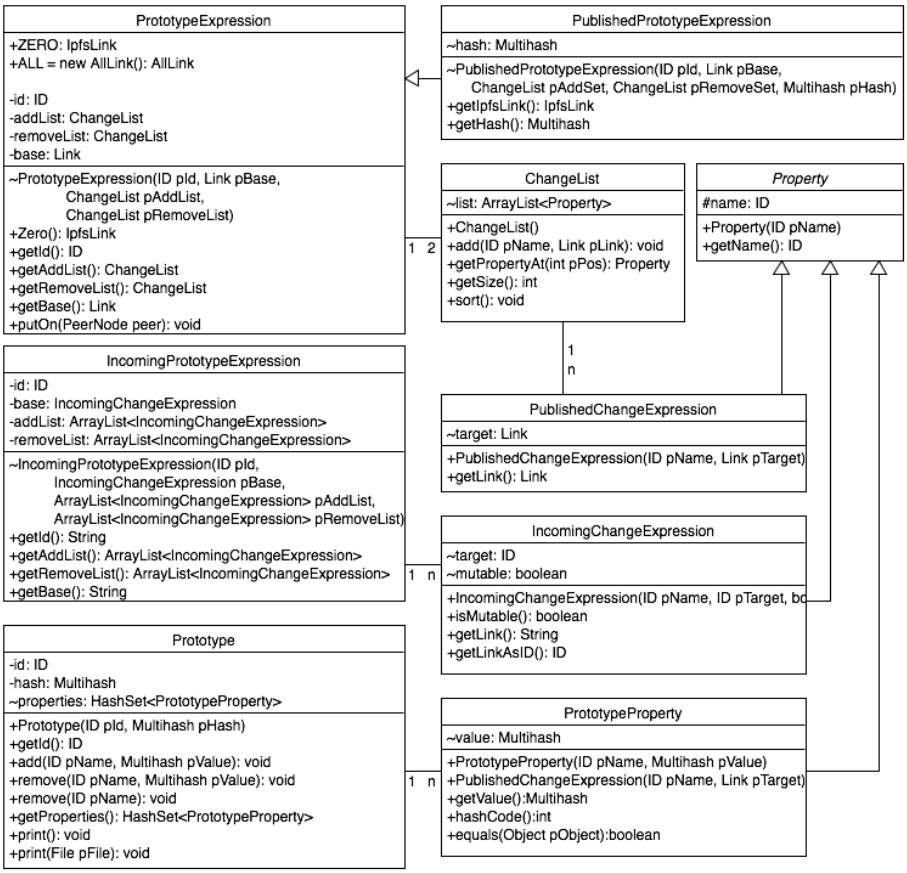


**Fig. 28.** ProtoIPFS prototype package. Visible is the abstract `Property` and its extensions. Also included are three types of prototype expressions and the prototype representation. Not visible are builder classes of `PrototypeExpression`, `IncomingPrototypeExpression` and `PublishedPrototypeExpression`.

# Eidesstattliche Versicherung

Name, Vorname

Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift

*Nichtzutreffendes bitte streichen

**Belehrung:**

**§ 156 StGB: Falsche Versicherung an Eides Statt**

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

**§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt**

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum

Unterschrift