RDA PID Information Types WG: Final Report

Document date: 2015/07/10
Authors: Tobias Weigel, Timothy DiLauro, Thomas Zastrow

Executive summary

The working group on Persistent Identifier Information Types of the Research Data Alliance concerned itself with the essential types of information associated with persistent identifiers. The working group developed a conceptual model for structuring typed information, an application programming interface for access to typed information and a demonstrator implementing the interface. The demonstrator is accompanied by a set of illustrating type examples which together with the application interface lay the groundwork for future adoption by individual science communities and institutions to contribute to a well-accepted set of core types.

The WG work has clarified the core terms related to PID Information Types and the relationship between the technical components involved in enabling typing. Part of this is a conceptual model for PID record properties, aggregated types and profiles. The formal deliverables of the WG work are this report including the use case documents, the API specification and demonstrator, and a number of exemplary types registered in the type registry.

1 Introduction

In complex data domains, unique and persistent identifiers (PIDs) are at the core of proper data management and access. In the simplest identifier services (e.g., PURLs[1]), the identifier is associated with a data object (including collection/aggregation objects), providing an identity that enables – through external means – relating that object (in web parlance, a resource) to other attributes or resources. Through more sophisticated ID services (e.g., DataCite[2], CrossRef[3], Handle[4]) some of these relationships can be captured in and exposed directly through the service itself as identifier metadata. This standardizes the manner in which that information is retrieved, reducing complexity. Generally, this identifier metadata may be accessed without visiting the resource itself, reducing load on the repositories and catalogs and enabling some use cases that would otherwise not be possible for dark archives. Depending on which types (clusters of properties) are provided, identifier metadata can enable services that support discovery, access, verification of integrity and authenticity and a variety of other use cases. To achieve its goals, though, such a service must be able to determine that the types of information it needs are available in the PID metadata.

In this context, the RDA PID Information Types Working Group was established to specify a framework for PID information types (PITs[5]), to develop consensus on some essential types, and to define a process by which other types can be integrated. The PID WG was formally approved after initial discussions at the first RDA Plenary in Göteborg in March 2013, together with the Data Type Registries (DTR) WG, which also formed a core dependency for the work of the PIT WG, since the core types designed by the PIT WG were planned to be registered in the type registry. The PIT WG was co-chaired by Tobias Weigel and Timothy DiLauro and planned to end after 18 months at the 4th RDA Plenary in September 2014 in Amsterdam.

The mission of the PIT WG was formulated under the goal of *harmonizing the basic information types associated with persistent identifiers*. To work towards this goal, the case statement of the PIT WG was designed around the idea that first and foremost, the necessary tools and concepts would have to be developed to finally achieve harmonization. The most important conceptual outcome was intended to be a framework for defining types, including aspects of technical interoperability but also suggestions for social governance. The essential practical outcomes of the WG were set to be an Application Programming Interface (API) specification and a prototypic implementation that demonstrate the use of some core types.

A “PID Information Type” is understood here as an umbrella term covering the more precise notions of identifier metadata elements and their aggregates, described in more detail further below. There is a related notion of PID types, which may be understood as a classification for different kinds of identifiers (or their respective identifier providers); however, this is not to be confused with typing the information associated with PIDs.

[1] http://purl.org
[2] http://www.datacite.org
[3] http://www.crossref.org
[4] http://www.handle.net
[5] The acronym PIT was first used by Larry Lannom.

2 Working Group process and outcomes

Discussions over the WG lifetime made clear that it will be impossible for a small group with a limited timeframe to come to a set of *widely accepted* core types. This lead to an overall pragmatic shift in working group activities and goals. The group was eventually able to design core types, and as seen below some core candidates have emerged, but these do not claim wide acceptance – a top-down approach was perceived as being futile in this form. Community use cases differ strongly, and there are often several interpretations or nuances of single property names. The conclusion is that an agreed set of core types will only emerge from practical use, and even then, the set of core types will most likely not be stable, but require some governance effort. The type registry is considered a key element in this respect. To deal with individual communities, the notion of namespaces was introduced.

Despite the lack of a fixed set of accepted core types, the discussions in the WG contributed to a better understanding of what information types are and what they could be used for. The essential element is the structure of registered properties and their aggregates. The functionality of the type registry that can lay the foundation for governance and interoperability approaches is a key element. The main conceptual outcome of the WG in this respect is thus the framework for structuring types, making them available through the type registry and guiding the social processes that can eventually stabilize accepted core types.

There is a possible overlap of information typically available from domain metadata catalogs and information from what is called here a PID record. In the most general sense, all such information is metadata. More precisely, PID records contain a small subset of digital object metadata, something also called the *persistent state information*. One possible view on defining this subset may be that it is useful to encourage repositories to expose at least some state information in a simple format through the generic PID Information Types API. The final distinction on what is considered to be part of a PID record and what is served as general metadata depends on the exact use case and varies between disciplines.

The original working plan included several ancillary elements that were discussed over the WG lifetime. Identifiable *collections* were integrated in the use cases where applicable, but a larger solution based on types was left open to be part of the adoption phase. The notion of a *profile* was originally perceived as a much larger, overarching concept than the actual aggregation-style entity described further below. This and the more general terminological clarifications form a valuable secondary outcome. Other high-level concepts were also discussed but postponed and suggested to be addressed at higher layers in a coherent identifier architecture.

3 Use Cases

The following use cases were initially gathered as examples for extended use of PID information types with the idea of cross-disciplinary relevance. Instead of using free text descriptions of user scenarios, the use cases were described through formal specification documents as are used in software engineering. Please refer to the **Appendix A** for these technical documents specifying each use case.

In short, the use cases are:

- **Data replication:** Manage replicas created from master objects for data safety purposes.
- **Data access load leveling:** Provide access to alternative data objects depending on availability and performance.
- **Format obsolescence audit:** Assess the format obsolescence risk of individual resources.
- **Versioning:** Give access to newer versions of an identified object and providing suitable context information.
- **Composite objects:** Reflect discipline-specific ranges of object granularity through type-encoded relations.
- **Managing data objects and metadata objects in combination:** Coverage of specific community scenarios that require both individually identified and interrelated data and metadata objects.
- **Managing object access permissions:** Enable fast decision-making of access control systems based on pure envelope evaluation.
- **Managing write control:** Controlling and tracking changes of data object collections.
- **Custom data citation:** Construct custom aggregates of multiple independent sources and provide citation information.
- **Modifying data:** Provide accountability of object modification and replacement in data infrastructures to ensure a fundamental level of service quality.
- **Provenance tracing:** Connect objects with their predecessors across repositories and provide forensic tools.

3.1 Use case example

Figure 1 (below) illustrates an exemplary scenario that reflects some cross-cutting issues of the more detailed individual use cases and provides an understanding of possible entry points. A user encounters an identifier to an unknown resource. She will submit the identifier name to a generic service that uses the typing information to guide the user to possible actions. For instance, the management service may simply provide access to the user according to given credentials. In case the dataset bears typed information indicating an obsolete format, the service may provide the user with a corresponding choice to initiate the follow-up migration use case. A third alternative may be that the data object identified is actually not available anymore. The user will be able to view a “tombstone page” providing some minimal information on the deleted dataset, such as the reason for deletion. In case the data is gone because a newer version was available, there may be an offer to access the newer version as is described in the versioning use case.

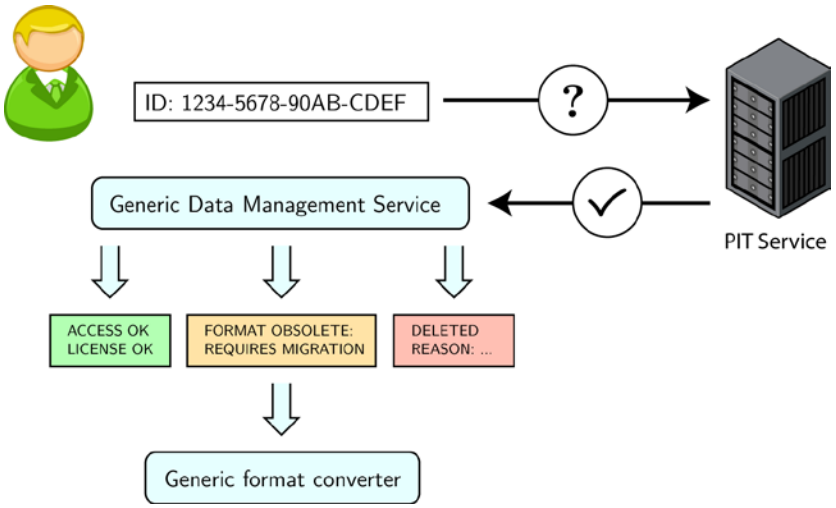

Figure 1: An exemplary workflow initiated by an end-user encountering an identifier to an unknown resource. Using a generic service endpoint designed for such end-user queries, the identifier will be resolved and typed information returned to an intermediate data management service. This service can then decide upon the typed information on the status of the identified resource and finally redirect the user to services matching both the resource type and its current status.

4 Conceptual model

The high-level architecture working model that emerged from initial discussions placed the information types between PID systems or infrastructures (e.g. the Handle System, the DOI system, URN resolvers, the ARK system and others) and applications that deal with PID operations. Because the capabilities and design of existing PID infrastructures differ, it was decided that the service providing the PIT API should be designed in a modular way so that potentially different adapters could be provided for the various infrastructures. The PIT API would be used by different clients as well, which may occasionally be end-user services, but more typically other infrastructure components (e.g. iRODS-based services or individual components of archival systems). If the PIT API is adopted by PID infrastructures and integrated into their services, the two lowermost layers of the architecture might converge.

A particular point of discussion in the early phases of working group activities revolved around metadata in general and metadata catalogs as a significant existing solution with wide usage across disciplines. A separation of concerns between metadata solutions and the ecosystem concerned with PID records was sought. It was deemed roughly acceptable that PID records are a form of metadata, though smaller in scope and not capable of replacing established metadata systems. The concrete dividing criteria were however left open.

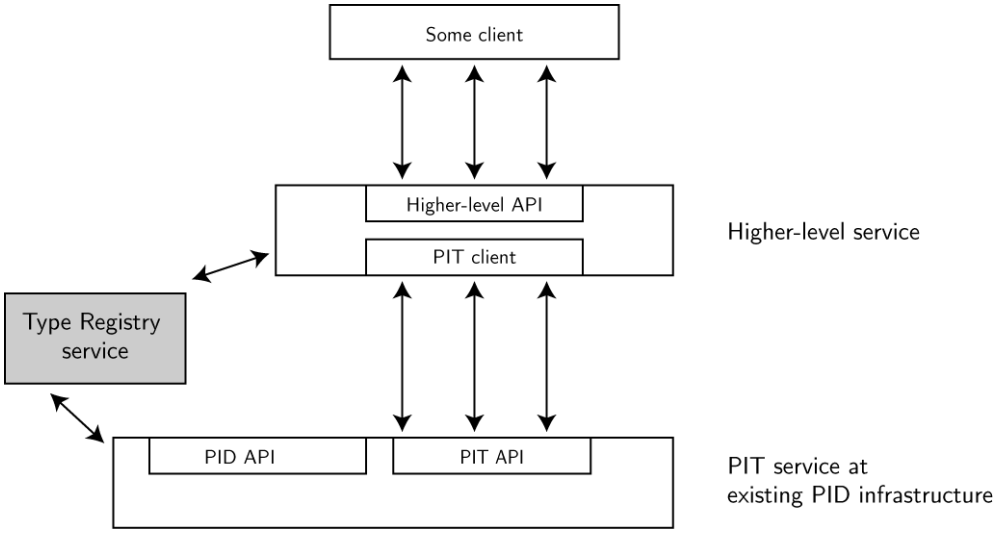

Figure 2: The high-level architecture for the PID Information Types API prototype consists of several layers with the PIT API situated between existing PID infrastructures and their native PID APIs and some consumer clients. The Type Registry forms a core dependency situated aside from the layer structure.

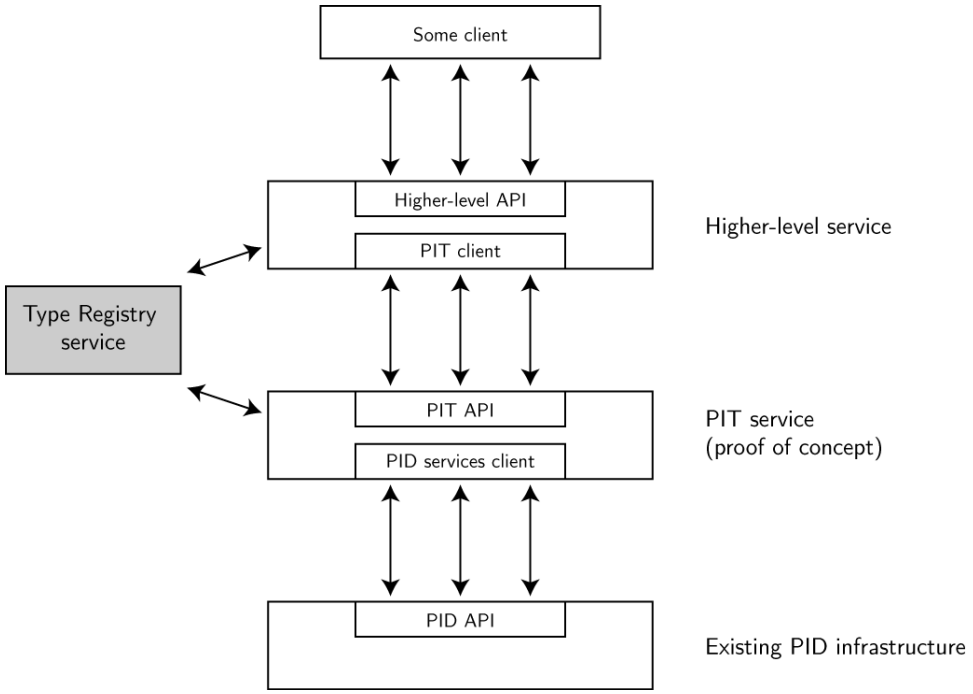

Figure 3: Desired high-level architecture after a long-term adoption process. PID infrastructure providers should offer the PIT API services in addition to existing PID services.

4.1 Data model and terminology

Over the course of the WG, terminological issues and clarifications were required that essentially revolved around two different understandings of the core entities. This section presents two fundamental models for these core entities that explain how they are used and should be named.

4.1.1 The property-type-profile model

This is the model used in the prototype. It may be more familiar to an understanding of domain metadata records.

In this model, every PID record consists of a number of properties. Every property bears a PID and its essential elements are a name, a range and a value. Only the PID and the value are stored in PID records, while the name and range are available from the registered property definition in the type registry. Every property is registered in the type registry, and aside from the property range and name, the type registry record provides additional information such as a description text and provenance information such as author, creation date and a contact address.

A type consists of a number of properties, which are subdivided into mandatory and optional. Every type is registered in the type registry, thus bearing a PID, description text and provenance information. A PID record is said to conform to a type if it provides all mandatory properties of that type. If a PID record is filtered by type, all mandatory and optional properties will be returned.

A profile consists of several types. A PID record conforms to a profile if it provides all mandatory properties of all types of the profile. There are two models for understanding profiles. In the first model, a profile is not necessarily globally discoverable and thus does not bear a global PID. This lessens the barriers of using profiles by individual communities, but such profiles are less shareable. In the second model, a profile is registered in the type registry. This motivates re-use of profiles and may increase interoperability, but the mandatory registration of profiles may make their usage too costly. The prototype implements the first model through a somewhat minimalistic emulation.

One motivation for including profiles is to preclude the proliferation of types that may occur if e.g. a community wants to reuse a predefined type but requires just one additional property or wants to exclude a particular property. It must be noted however that the emergence of quite similar types cannot be prevented completely.

4.1.2 The type-profile model

This is an alternative, simpler model, which is closer to the idea of types as underlying the type registry and the Handle System.

In this model, the elements of a PID record may be typed, and those types may be registered in the type registry. Every type bears a PID and consists of a name, range and value, and thus is equivalent to the property of the property-type-profile model.

Likewise, types are aggregated into profiles, and profiles may specify optional and mandatory types. A PID record can be checked for conformance with a profile. This is thus equivalent to the understanding of the type in the property-type-profile model.

In this model, there is no equivalent entity for the profiles of the property-type-profile model.

4.1.3 Discussion: Considerations on picking a model

Overall, the property-type-profile model may be closer to an understanding of PID records from a domain metadata perspective, because the notion of properties and types is closer to an understanding of properties and classes from e.g. the UML world. The type-profile model is more in line with the traditions of the Handle community and some communities experienced in dealing with Handles such as DARIAH.

Property-type-profile model	Type-profile model	Description
Property	Type	Atomic elements of a PID record
Type	Profile	Aggregations of first level entities
Profile	N/A	Aggregations of second level entities

The recommendation is thus: The type-profile model is simpler and if adoption is hampered by inherent complexity, the property-type-profile model should be abandoned in favor of the type-profile model.

For the remainder of this document, we will stick to the property-type-profile terminology, but keep in mind that fully-fledged profiles are somewhat optional.

Figures 4-7 (below) summarize the current data model. In the full property-type-profile model, types aggregate properties and profiles aggregate types. Services will typically work with types, i.e. require that a specific PID record provides information conforming to a particular type. Profiles may or may not be used. The type-profile can thus be expressed as a particular subset of the full model.

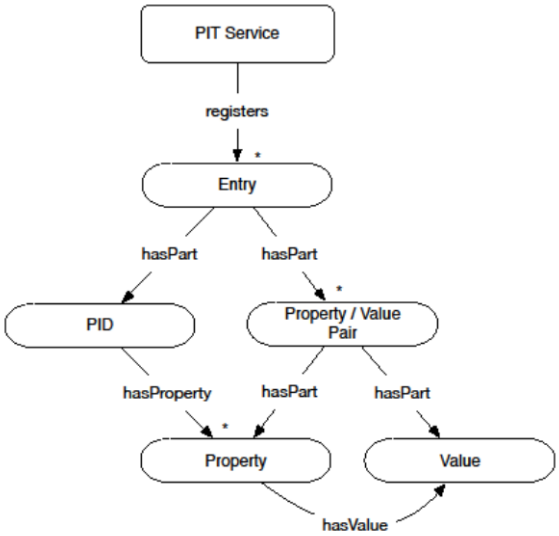

Figure 4: PIT data model.

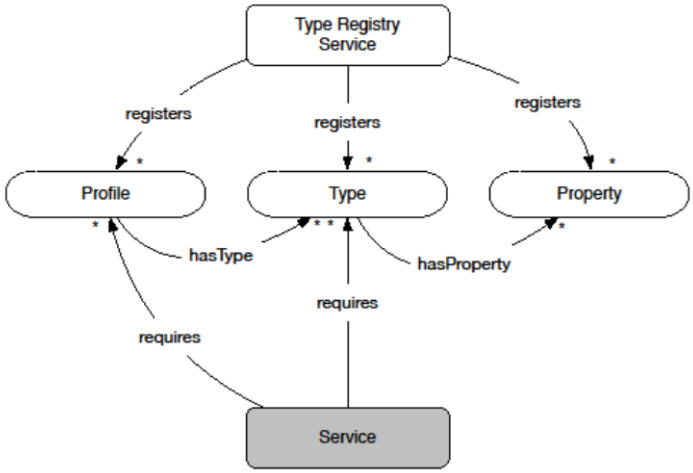

Figure 5: Type Registry data model. A service may require types and/or profiles; however none of these are mandatory.

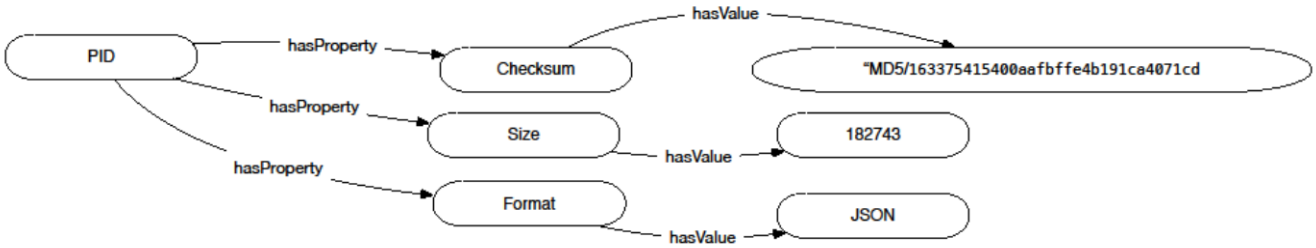

Figure 6: Exemplary PID record entry that uses registered properties from the Type Registry and offers exemplary values.

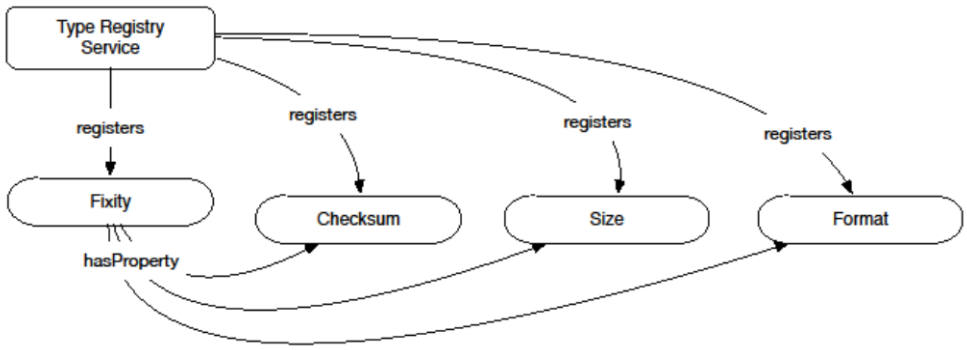

Figure 7: Examples for registered property entries in the Type Registry from the PID record above. The PID record conforms to the exemplary 'Fixity' type.

5 Type examples

As discussed earlier, there is a number of type example provided by the WG that illustrate how information types and the API can be used, but do not claim to be normative core types widely accepted across communities.

The following table summarizes the examples that are included as part of the WG outcomes; the table and specific entries are discussed in more detail further below.

Name	Range	Identifier	Flags
Type: Citation Information (EXAMPLE namespace) 11314.2/d5396a97c316a0eaca055846ba4233ac			
Title	STRING	11314.2/07841c3f84cbe0d4ff8687d0028c2622	
Creator	STRING	11314.2/31810b2c24913929bb5e0d4d949de9f7	
Publication date	DATE	11314.2/daed5901fbbe2570ee95c4009c739de2	
Language	STRING	11314.2/56211d62153b3500ce3b16cf86d6b403	optional
License	STRING	11314.2/2f305c8320611911a9926bb58dfad8c9	optional
Type: System level access information (EXAMPLE namespace) 11314.2/09d35f22e48b60284029ba51c17e2944			
Creation date	DATE	11314.2/6b3e1230d1b68965e290b16a43d2f46d	
Deletion date	DATE	11314.2/7e78be9736ad7f6bb5fb31218821eba5	optional
Permissions	STRING	11314.2/d057258f7b406fd9aad5a3893aba8208	optional
Checksum	STRING	11314.2/56bb4d16b75ae50015b3ed634bbb519f	
Object size (in bytes)	STRING	11314.2/0006e2b8e2f6e1ecce836e593bed38ae	
Type: Aggregation information (EXAMPLE namespace) 11314.2/699d487eff50c2e10982f4b85ed053a9			
Parent object identifier	IDENTIFIER	11314.2/f9e66e5f64ba3179d8f1e64138c69e04	optional
Child object identifier	IDENTIFIER	11314.2/f8db9e3b5f97aa8168fbd59788476375	optional
Type: Versioning information (EXAMPLE namespace) 11314.2/6b507d787dd06e4eb8f23b5bb56ae8bb			
Predecessor identifier	IDENTIFIER	11314.2/467d9ba30e2d9879fd9d483f319e462c	optional
Successor identifier	IDENTIFIER	11314.2/fc78024cb9dac0b0a80ed631ea650d4b	optional
Type: Preliminary example for EUDAT core information (EUDAT namespace) 11314.2/5f45666fc8689e3565728ca512c1b5e7			
Checksum	STRING	see above	
Format	STRING	11314.2/1a4f53a28b72d4bf4f8fdda7a2089595	
Data identifier	IDENTIFIER	11314.2/24dd85c4a3d39fb0d7e83a510a5041c6	
Metadata identifier	IDENTIFIER	11314.2/58a44100d2bcd1a34fb87eb87bc6f701	
Repository of Record	IDENTIFIER	11314.2/5546b0166091d9ae869f081f5548f3fc	
Mutability flag	BOOLEAN	11314.2/7c81e954eaead6a2f772abd83986d3e9	
Landing page address	URL	11314.2/66af2639d388977e81b85f6413df1e2c	
Date of deposition	DATE	11314.2/35837218f18dcc54a2d32e0fb30fa7fb	

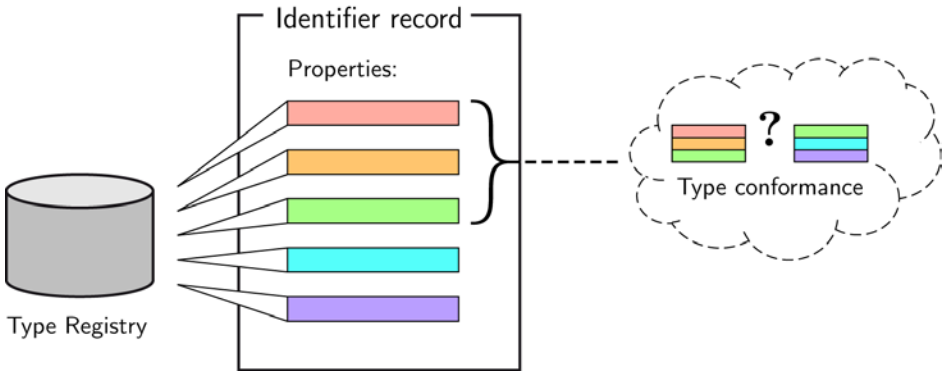

Figure 8: Relationship between a number of properties constituting an identifier record and types as aggregations of a defined set of properties. A conformance query of a specific type against a specific identifier record will then match if all of the mandatory properties of the type are present in the identifier record. Both types and properties are registered in the type registry.

Note that the namespace field is currently ungoverned. The recommendation is to come to a governance mechanism in the mid-term and use persistent identifiers instead of simple vocabulary terms.

The range entries (value types) of the example properties are currently taken from a simple controlled vocabulary with entries such as the generic STRING and more precise BOOLEAN, DATE, IDENTIFIER and URL. This is another example where mid- to long-term adoption must lead to a viable solution. A future solution could for example employ a properly governed controlled vocabulary or registered persistent identifiers. The latest prototype of the Type Registry lists some of these elemental types, fully registered with individual PIDs. Some restrictions on the actual values should be enforced, such as agreeing on a particular standard for date and time encodings. Governance may be simpler than for the other elements such as namespaces, properties and types if the scope is limited to elemental types and complex or compound types are part of an ungoverned extension. The prototype is agnostic towards any solution since it does not interpret the range field.

The EUDAT core information type illustrates a use case particularly requested by WG members from the EUDAT project. The current set of properties in this type is just an example, also to demonstrate how individual communities can combine (i.e. re-use) more generic properties with their community-specific ones. This example is far from being finished, further input and development from EUDAT members is required and likely to continue as a process of its own after WG termination. A particular construct is hinted at by the use of the IDENTIFIER range for the Repository of Record: This would typically point to a repository address and the entity maintaining this repository (such as DKRZ maintaining its local EUDAT repository). The use of organizational identifiers is very appealing to ease the maintenance of individual repository locations and affiliations.

The example properties given as part of the aggregation and versioning types are kept intentionally simple to reflect the lightweight requirements of the corresponding use cases. This does not preclude more sophisticated approaches in the future that may be based on a full (possibly cyclic) graph model if such is required by more complex usage scenarios.

There are also two example instances for regular PID records that make use of the properties listed above and conform to some, but not all of the types. These examples are taken from the Earth System Grid Federation, which is a potential adoption candidate for the WG outcomes:

- 10876.test/esgf_data1
- 10876.test/esgf_data2

5.1 Practical examples for using types and profiles

To understand how the type examples, the API and the overall architecture fit together, we will provide a small example on citation.

Imagine that there is a citation aggregator service which regularly harvests various repositories for citation information available from the identifier records of the curated objects. Such an aggregator service would get harder to implement the more diverse the information entities from the individual sources become. If however all repositories agree on a certain set of elemental properties to be provided for each object, the aggregator service is much easier to implement and maintain. These elemental properties would optimally be combined into a single type, e.g. the citation information type example given above. Doing so does not prevent individual repositories from gathering and providing more properties that are not listed in the type, because each record may carry any number of properties, and the type may just be a subset. There may be other, possibly overlapping, types in use by the repository. In case these are supported, the multitude of types may be purposefully arranged with profiles; an example is given further below. Note that profiles are not related to the notion of collections that aggregate several objects into a compound entity with a separate identifier.

Given that a citation information type is agreed upon by all repositories, the harvester may then limit its queries to PID records conforming to the type. Moreover, other services may evolve that make use of the same type, for example for a custom citation service that an end-user can use to construct collections of objects across repositories. Such a service would benefit from the abstracted information view provided through the type, effectively being able to ignore the intricacies of the individual source repositories.

Another usage example is given by a version tracking scenario. Imagine that a data object in one of the repositories is replaced with a revised or otherwise improved version. The repository maintainers decide at this point to assign a new identifier to this object, but relate it to the older version using the predecessor property from the versioning type example in Table 5. Also, the older object would have a successor property pointing to the new object's PID.

An aggregator service could then use conformance queries to not only filter for citation information types, but also for versioning information. Since not every repository may provide versioning information (and even those that do will only do so for a fraction of their objects), all kinds of combinations of the two types will emerge. A profile that aggregates all properties from the citation and the versioning type could therefore be used to check conformance for a specific service that both displays citation information and points to older or newer object revisions.

Another general scenario for using profiles is the case where a repository offers additional information related to versioning beyond the two properties in the type example. For instance, a specific property "version number" may be provided. Since not all repositories may do so, however, the generic services shared across them will still solely work with the general versioning type. The repository with additional version number information may therefore formulate an additional type for this (bearing just one property) and design its own services so they check for conformance with a profile that finally consists of all three properties of both types. Doing so will not affect the shared

services but still allow for internal filtering and possibly sharing the advanced profile and services based on it with other adopting repositories later on.

With regard to relational model theory, types can be seen as a mechanism to achieve some amount of functional dependency even if the data model behind identifier records follows a schema-less NoSQL-like approach: It can generally not be assumed that the combinations of specific properties and property values are the same across all identifier systems. With registered types and conformance checks used by services that span identifier systems and repositories, some amount of coherence may become feasible.

6 API overview

Interaction with information types and typed PID records happens via a webservice API. The PIT WG provides a first prototypical implementation of the API. The API is designed to be agnostic towards the underlying identifier system with the distinct requirement that it must be possible to store and retrieve additional information associated with a single identifier.

The range of properties is not controlled or checked by the prototype, all data is treated as simple strings. It is up to the clients to convert the JSON data to their needs. The recommendation for future work is to establish a set of core range types, such as string, date, integer, boolean or similar. However, there are foreseen problems with this approach, such as the question how to deal with different integer sizes and time formats.

The prototypical implementation including full documentation is available from **git://redmine.dkrz.de/rdapit.git**.

The state of the source code at the official RDA outcome process is available as the git tag 'rda-outcome'.

API base path of the current installation: **http://smw-rda.esc.rzg.mpg.de/rdapit-0.1/pitapi**

Demonstrator client web-GUI: **http://smw-rda.esc.rzg.mpg.de/PitApiGui/**

6.1 Important decisions and trade-offs

An issue left open for now is how to deal with multi-valued properties. Three cases can be roughly distinguished:

1. A property with same name may be used multiple times, e.g. there are multiple occurrences of the property "author" with a single atomic value (e.g. a person ID) each.
2. A single property with multiple values, e.g. a single occurrence of the property "author" with a list of values (e.g. a list of person IDs).
3. A generic property with sub-types variations, e.g. a "checksum: SHA-1" and a "checksum: MD5".

The current API and implementation naively assume that (1) does not occur, and (2) is addressed at a higher level. For example, the client contacting the PIT API server would store and retrieve JSON constructs (arrays or objects), which are treated as opaque strings by the server implementation. The case (3) would currently be solved by registering distinct properties, thereby hiding their implicit relationship. In all cases, the referential integrity of property values and registry entries should be

guaranteed: If there are specific values from e.g. a controlled vocabulary, they should be dereferenceable, e.g. through the type registry, independent from how many times they occur and how they are encoded.

A recurring issue that needed to be addressed was whether to store type conformance information in PID records (stating conformance explicitly) or to perform conformance queries always on the current state of a PID record. Explicitly stated information would reduce effort required to answer queries and make the interpretation of conformance less ambiguous and independent from particular PIT implementations. However, when a PID record is changed through other channels than the PIT API, it would be impossible to keep the type information correct, and the PIT API would have to deal with potentially contradicting information. Also, compatibility with legacy records would be broken. The final decision was therefore to always perform conformance checks and not record any explicit information in the records.

The prototype does not offer sophisticated authentication and authorization. The overall recommendation for the API is to use HTTP-auth of the application server for basic access control.

6.2 API methods

All methods receive and accept data in JSON encoded format. This is also a minimum requirement on the choice of formats that must be provided by conformant implementations; other formats (e.g. RDF/XML) may be provided additionally. The following is a very brief overview on the major methods of the API; more details can be found in the technical Javadocs.

GET /pid/{identifier}?filter_by_type=…&filter_by_property=…&include_property_names=…

Returns some or all property values from the given PID record. The method supports filtering by type or single properties, and can also inject property meta-information from the type registry such as the property name. The method also checks for conformance of the PID record with the given types. Profiles are supported through filtering by several types.

GET /property/{identifier}
GET /type/{identifier}

Returns meta-information of properties or types from the type registry.

GET /peek/{identifier}

Can be used to determine whether a PID points to a simple object, property or type. This is most useful for services that deal with PIDs from an unknown context.

POST /pid

Assigns a new PID with properties set to given values. The method uses random UUIDs as PID names and will return the created name.

6.3 Servlet implementation and client GUI

The WG has implemented a prototypical servlet implementation and a client GUI in Java based on the Jersey and ZK frameworks. The servlet can be deployed in a Java application server; currently, Apache Tomcat is used. The prototype server offers all essential methods to read PID records, property and type definitions, while the prototypical client GUI provides basic lookup facilities. The prototype relies on the type registry prototype provided by the Data Type Registries WG and a preliminary Handle Server v8 installation courteously provided by CNRI.

The servlet implementation is not optimized for response time and overall performance. The frequent queries to the type registry currently issued form a major performance bottleneck, particularly in case of full property name resolution, as this requires multiple type registry requests to answer just one PID record resolution request. A possible solution is local caching of type registry records, as they are expected to be quite static over significant timespans.

6.4 Registering new types

A Type Registry instance or access to an instance is of course required. CNRI currently runs a prototype service, so before registering new types, please contact the PIT and DTR WGs. To be useful for the PIT implementation, a property/type/profile registered in the registry needs to provide additional metadata fields. Details on this are available in the Javadocs[6]. It may be useful to license new types under a public domain (e.g. Creative Commons CC0) license to enable easy reuse by third parties.

7 Adoption pathways and future work

As applications emerge as part of the adoption process, the long-term goal to come to a coherent set of information types that support a wide range of base-layer cross-disciplinary use cases could be reached.

Adoption could happen at the technical level or the conceptual level. At the technical level, the API could be offered server-side by identifier providers, reducing effort and costs for their customer communities. In an alternative scenario, discipline-specific e-science infrastructures could implement the server-side API on top of standard interfaces offered by their respective identifier providers. This requires that there are resources available at the community-level, which is often underfunded.

Client-side adoption should happen through individual organizations or communities. At the conceptual level, these stakeholders should contribute to the ecosystem of types, preferably in coordination with other interested parties that will eventually lead to a frequently used or advertised set of core types. The first types may however be very community-specific as this gives the best short-term ROI in terms of directly useable information in existing or quickly developed tools and services.

[6] http://smw-rda.esc.rzg.mpg.de/apidocs/rdapit/rest/TypingRESTResource.html

A pragmatic approach is thus to go to the individual communities, work with them on adoption and if there is a continued interest, ask the identifier providers to adopt it on the long-term as they may see their customers' demands.

In the long-term, there should not be a coherent set of types limited to a single identifier system, because by doing so we will not reach a sufficient level of acceptance. On the other hand, the broad-scale adoption of the API will not provide a sufficient long-term return on investment if the respective consumers still have to invest considerable resources on defining their own types due to lack of convergence.

Possible follow-ups to the WG efforts are:

- **Refinement of the data model.** It is unclear whether the data model is both flexible enough to cover multiple disciplines and pragmatic enough to encourage a bottom-up process. The data model may be revised after some first-hand community exposition. This should also include an attempt to include **multi-valued properties**.
- **Enhancement of the API.** Similarly to the data model, tools and services at the community level may need other or more methods. The balance must be kept however so that nothing too community-specific enters the agnostic API.
- **Performance requirements.** Communities may require a certain quality of service from identifier providers for accessing type information. This was briefly discussed at RDA Plenary meetings, but not in sufficient detail. One option may be to include performance requirements in existing service-level agreements (SLAs).
- **PID information types as part of the Data Fabric.** The exact notion of a Data Fabric is currently unclear, but there is rough consensus on PIDs and associated information playing the role of a central component.
- **Association of profiles with prefixes.** To reduce effort and reduce variations in PID records, communities may want to enforce a specific profile on a whole prefix. It is unclear how this will work across PID providers as there are differences in the notion of prefixes (PID namespaces). The PIT API may also be unable to enforce such regulations as it does not restrict write operations.

Future work may also be concerned with the application of identifiers to different kinds of entities, such as organizations or individuals. For this reason, the future developments within the larger RDA Persistent Identifier Interest Group are of relevance. Also, the future evolution of the Type Registry must be observed, both technically at the API level and organizationally on questions of federation, namespaces and sustainability.

The exact relationship between PID records and metadata (and, in particular, existing extensive metadata catalogs) is also not clear at this moment; however there are some boundary conditions and suggestions. PID record information may be more easily retrievable, live through object disappearance and offer more performant access than regular metadata. Synchronization between these information spaces must however be dealt with carefully to avoid unnecessary duplication and conflicts. It may even be deemed beneficial to more prominently expose some information stored in catalogs at the PID level. Such questions may however be properly addressed only by working with adopting communities.

8 Prototype interface impressions

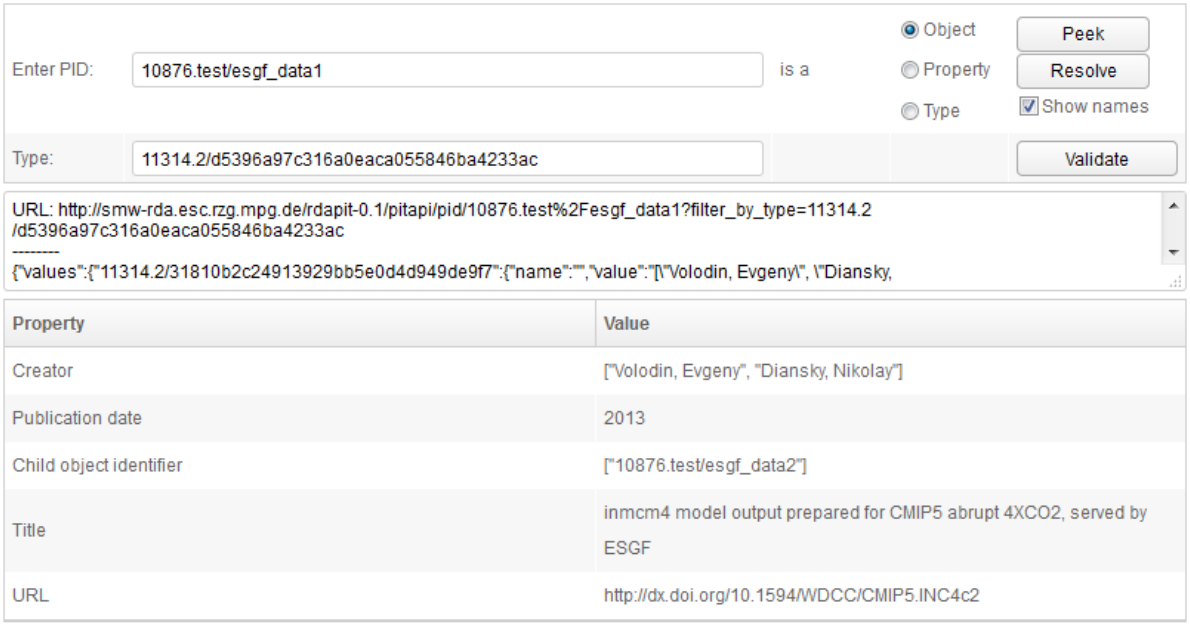

Figure 9: Graphical user interface showing an example object record. All entries are taken through the PIT API from a PID system and a type registry. The individual properties are registered in the type registry; the name (caption) listed there is displayed (as opposed to the registered property PIDs). The record's conformance to the "citation" type (11314.2/d539…) can be verified through the separate validation action.

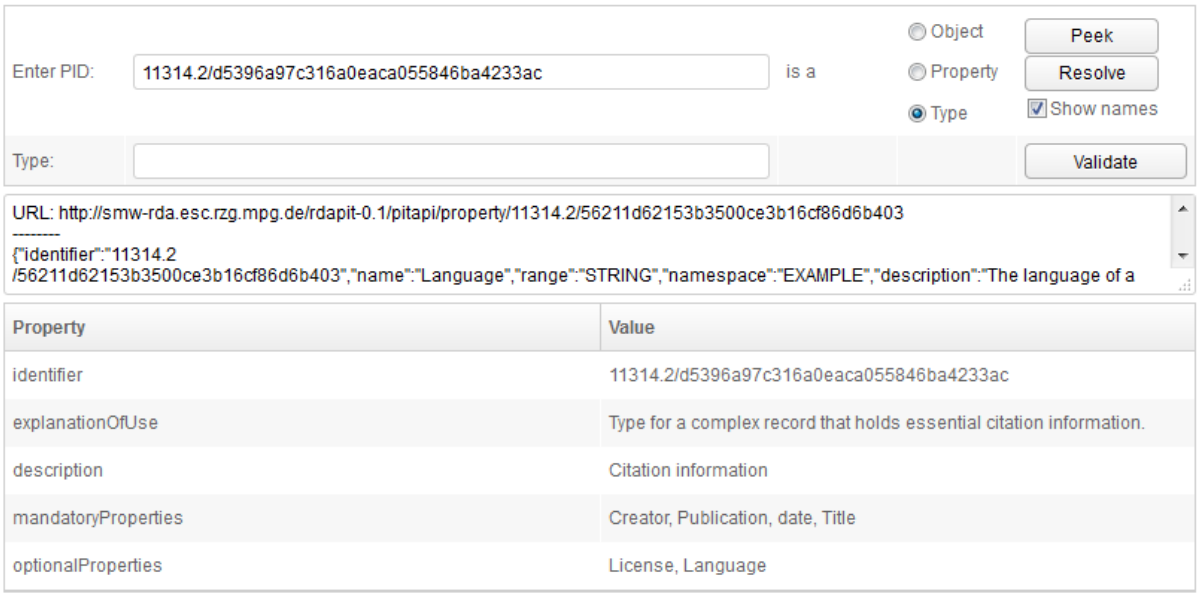

Figure 10: The graphical user interface showing the "citation" type record, offering several mandatory and optional properties. All information is taken via the API from a type registry. The record in figure 9 conforms to this type description.

A Appendix A: Use Cases

This appendix lists the use case documents, which were discussed during the first half of the WG lifetime. They are representative of individual stakeholder use cases and reflect the varying scenarios in which individual communities deemed PIDs to be a possible ingredient.

A.1 Data replication & Data access load leveling

Author: Tobias Weigel

Summary

- the goal is replication, i.e. one item X at a location S should be replicated to another location D
- the context is to move stuff around automatically; either based on rules (iRODS) or through conventional middleware
- therefore, the workflow is non-interactive; all actors are non-humans (for the replication case)
- if there are PIDs for individual replicas, they should be navigable from and to the PID of the master item
- possible existing target infrastructures are EUDAT (FP7 project) and ESGF (Earth System Grid Federation, www.esgf.org)
- the difference between replication and load leveling is that replicated items are not accessible for end-users (they only serve as backups), while load leveling explicitly targets data access at any location

Actors

- Source System (system)
- Broker (system)

- File Replication Service (system)
- Data consumer (human; only for load leveling)

Preconditions

- the item X (and any sub-items) already has an assigned PID pointing to its current location at S
- a replication policy has been configured, which forces the source system to act

Main Flow (replicate)

1. The Source System submits a request for replication of item X at S to destination D. The request contains the PID of X, but not X (the payload data) itself.
2. The Broker determines the structure of X by looking at its PID record.
3. If the structure of X is a collection or hierarchy, the Broker walks all elements; otherwise, it looks at X only.
    1. For each element, the Broker determines the item data type by looking at the PID record.
    2. Given the type, the Broker filters each element to assemble the Replication List. For example, Metadata-type elements may not have to be replicated.
4. The Broker arranges the transfer of each element of the Replication List from S to D.
    1. For each element, it accesses the PID record and retrieves the current storage location
5. All locations are submitted to the File Replication Service and the flow of events is on hold until it terminates.
6. When the File Replication Service process terminates, all new file replica locations are stored in corresponding PID records
    1. This can mean to create new PIDs and reference the original PIDs (and vice versa) or to update each original PID with an extra replica location.
7. The Broker replies to the Source System that replication has completed.

Main Flow (lookup – only valid for load leveling)

1. The data consumer submits a PID to the Broker.
2. The Broker accesses the PID record and determines whether the structure is a collection or hierarchy.
    1. If the structure is a collection/hierarchy, the Broker walks all elements and assembles a list of object locations according to proper measures by choosing from the locations given in the PID record. Metadata-type elements may reside at a single location only.
    2. If the structure is not a collection/hierarchy, the Broker selects a single object location according to proper measures.
3. The Broker replies to the data consumer with the determined object location(s).

A.2 Format obsolescence audit

Report Format Risk and Possible Migration Paths for a Known Resource

Author: Tim DiLauro

Goal: Determine if a known resource or any of its dependencies are at risk for format obsolescence and, if so, produce a report of potential migration paths for the affected resources.

Actors: Requesting Agent (human or system), Auditor (system), Format Registry (system), Preservation Planner (generally a human)

Summary: Over time, file formats become obsolete or unavailable due to changes in available technical environments. In order to avoid loss of access to needed content in these formats, the onset of these events must be detected and strategies developed to migrate this content to formats in supported technical environments.

Pre Conditions:

The resource, its descendents, and their dependencies have PIDs and the PIDs have associated data describing structure (resource type and, for bytestreams, the file format).

Triggers:

Requesting Agent determines that a resource should be evaluated for obsolescence.

Normal Path:

1. The Requesting Agent contacts the Auditor and requests an obsolescence audit, providing the PID of the target resource.
2. The Auditor uses the PID to access the resource's PID metadata.
    a. If the resource is a bytestream, the Auditor adds its PID and format to a list of resources to review.
    b. The Auditor obtains a list of the PIDs of the resource's descendants and their dependencies and recursively processes each of them via step #2.
3. For each unique format on the list, the Auditor queries the Format Registry to determine the obsolescence risk.
    a. If the risk is low enough, items matching that format are removed from the list.
    b. Items remaining on the list are at sufficient risk to warrant further review.
4. For each unique format in the remaining items, the Auditor requests a list of viable migration paths from the Format Registry.
5. The Auditor generates a summary list of at-risk formats with associated migration paths.
6. The Auditor generates a complete list of at-risk resources, their formats, and associated migration paths.
7. The Auditor forwards these lists to the Preservation Planner.
8. The Auditor notifies the Requesting Agent that the process has completed.

Alternate Paths:

- 2a. The PID metadata record for some bytestream resources may not have format information.
- 2b. The PIDs of some or all child or constituent resources are not captured in the PID metadata record: The Auditor would need to dereference the PID to extract the identifiers of these resources, if they are available.
- 3. The Format Registry might not have obsolescence risk information for all formats. The Auditor would need to default to either removing those items from the list, leaving them on the list as-is, or leaving them on the list and noting that risk information was unavailable. This might be driven by some policy or heuristics.

- 4. The Format Registry may not contain migration paths for some formats. The lists should indicate that there is no migration path for such formats.

Post Conditions:

The Preservation Planner has a list of at-risk resources and some possible migration paths for some or all of them.

Notes:

This case does not fully account for a conceptual resource like a Data Item (or Data Object), which might have multiple manifestations (representations) of the same information content as a result of previous format migrations/transformations. For such an object, the Auditor would need to determine if the resource has one or more manifestations that are not at risk, and if so, remove that resource from the list.

A.3 Versioning

Goal: Enable basic version control to improve accountability.

Actors: user (human or system), system (system)

Summary:

Data is stored in a repository and receives a PID. At a later point in time, a new version of the same data is available. When it is stored in the repository, questions are what to do with the old copy and its PID, whether to assign a new PID and if so, how to manage/interlink the two PIDs. PID types may help to establish the link between old and new version PIDs or specify which policies apply in this process.

Subcase 1) Old data "A" must be replaced by new data "B"

Precondition: A PID has been assigned to old data A.

Main flow:

1. The user submits B and the PID to A to the system, requesting a version update
2. The system assigns a PID to B and stores B
3. The system arranges the PIDs of A and B in a way that enables the other use cases (e.g. via the PID record – or external metadata)
4. Optional: The system deletes A and marks the PID to A with a fate flag ("data has been deleted because a new version is available")
5. The system replies to the user with the new PID to B

Postcondition: PIDs for A and B exist. B is kept. Optionally, A is kept.

Subcase 2) User access to latest version

Precondition: A consecutive series of data version has been submitted (using subcase 1) with the newest version being object "Z".

Main flow:

1. User submits PID to A
2. System accesses PID record of A and determines the latest PID to Z
3. System responds to user with Z and the PID of Z

Subcase 3) User access to specific version

Precondition: same as for subcase 2)

Main flow:

1. User submits PID to A
2. System accesses PID record of A. Alternatives may happen:
    1. If the object A is not available, the system replies to the user with an information on the fate of A. Optionally, the PID for B or Z is included.
    2. If the object A is available, the system replies to the user with A. Optionally, the PID of a predecessor version may be returned to enable back tracing.
3. Optional: If a newer version is available, the system may enquire from the user whether the newer version should be returned. In that case, continue with subcase 2).

A.4 Composite objects

Goal:

Browse composite / hierarchical objects through a designated tool.

Actors:

Data provider (human or system), composite tool (system)
Data consumer (human or system)

Summary:

No consensus exists regarding the level of granularity at which PIDs are assigned to data objects. Different usage scenarios require different granularities, and thus PIDs must become hierarchically structurable. If both individual objects and the larger composite receive PIDs, then these implicit relations should be discoverable for humans but also for machine agents that for example copy or analyze objects.

Preconditions:

Each element object already bears a PID. There is no super object with a PID yet.

Main Flow (create composite):

1. The data provider submits a list of element object PIDs to the composite tool.
2. The composite tool analyzes each element.
    1. If there is information on an element that signals it as being in a composition already and that multiple membership is not allowed, the use case terminates with a corresponding failure message to the data provider.
3. The composite tool creates a PID for the composite and returns it to the data provider.

Main Flow (discover composite):

1. The data consumer submits a PID to the composite tool.
2. The composite tool analyzes the PID information.
    1. If the PID refers to a composite, the tool returns a list of all element PIDs to the data consumer.
    2. If the PID refers to an element, the tool returns a list of all super objects to the data consumer (if any). This list may contain only one entry if no multi-membership is allowed.

Special cases:

This use case can be extended to many increasingly complex scenarios. It is now kept intentionally simple.

The current “create” flow of events assumes a case where there is no super object with an assigned PID and where the PID created for the composite does not point to a distinct super object itself. This may not be the case for some communities.

The question of multiple membership may be answered differently for particular communities, thus the event flows take care of both possible scenarios.

Some communities may also have composite objects with a huge number of elements, which may render the discoverability of super objects impractical due to the number of required links and PID operations. Things also get more complicated if an object is simultaneously an element and a super object (for hierarchies with multiple layers).

A.5 Managing data objects and metadata objects in combination

Goal:

Binding distinct data and metadata objects together.

Actors:

Data provider (human or system), binding tool (system)
Data consumer (human or system)

Summary:

If there are distinct data and metadata objects and these are deposited together in a repository, both of them may receive an individual PID. PID types may help by distinguishing data from metadata objects and connect them with each other.

Preconditions:

There is a distinct data and a distinct metadata object and each of them bears an individual PID.

Main Flow (connect):

1. The data provider submits a data PID and a metadata PID to the binding tool in 'bind' mode.
2. The binding tool connects the two objects and returns with a success message.

Main Flow (navigate):

1. The data consumer submits a PID to the binding tool in 'query' mode.
2. The query tool analyzes the PID record.
    1. If the PID points to a data object, the binding tool returns the PID of the data object to the user and a note that this is a data object.
    2. If the PID points to a metadata object, the binding tool returns the PID of the metadata object to the user and a note that this is a metadata object.

A.6 Managing object access permissions

Goal:

Allow or disallow access to a specific file based on criteria such as current time or total number of times the file has been accessed.

Actors: user (human or system), gatekeeper (system)

Summary:

Repository administrators or data owners want to limit data access in order to prevent uncoordinated distribution of preliminary or temporary data.

Preconditions:

- the file has received a PID
- criteria have been defined in the metadata (PID metadata or other)

Triggers:

The user requests access to an object using its PID.

Normal path:

1. The user submits the PID to the gatekeeper.
2. The gatekeeper resolves the PID and collects the relevant metadata.
3. The gatekeeper assembles a list of criteria to check based on the metadata.
4. The gatekeeper decides on each criterion based on contextual information available (e.g. the current time or prior access information deposited as part of the metadata).
5. The gatekeeper either returns the requested data to the user (or redirects to a retrieval service) or responds with a request denial.

A.7 Manage write control

Goal:

Control write access to a specific collection for a specific user.

Actors: user (human), system, manager (human)

Summary:

In a repository that organizes objects in collections, policies exist that demand that users do not modify such collections arbitrarily. For instance, adding or replacing new collection elements must be restricted, and for this purpose, a ticketing system is used that is ultimately based on PIDs.

A) Initial assignment of a ticket that gives a specific user permission to modify a collection.

Precondition: Objects have PIDs assigned

1. Manager requests a ticket PID from the system to give permission to a user for a specific collection under some restrictions
2. The system creates a ticket PID and stores information with it that points both to the data collection and to the user; this also includes restrictions.

B) A user modifies a collection via a previously assigned ticket.

Precondition: A) done.

1. User submits ticket PID to the system with a specified request to write new data to the collection
2. The system accesses ticket PID information and decides based on that record and the current system status:
    1. The remaining write cycles on the ticket have expired (volume, number of files) or the allowed time period is not matching. Use case terminates without changes to data or PIDs. The system responds to user with a warning message stating the reason for denial and possible further steps (try again later, renew ticket, upload new data)
    2. All restrictions match.
        1. The system performs the requested action (add or replace collection elements)
        2. The system modifies the PID information to record the performed action.

A.8 Custom Data Citation

Author: Tobias Weigel

Summary

- The goal is to enable an end-user (author of a paper) to cite a customized collection of data, spanning across DOIs (or any other form of PID) or being more fine granular than them
- This is relevant to probably many disciplines
    - where data from multiple sources are combined to solve a research problem
    - or where an exact specification of which data was used is tedious given existing solutions
- One particular challenge is that the individual data PIDs may belong to different identifier infrastructures, disciplines and data repositories, which will provide further challenges for enabling seamless access

Actors

- Author (human)
- Customizer (system)

- Registry (system)
- Auditor (system)

Pre Conditions

- The data to cite (all collection elements) are already identified through existing PIDs

Main Flow

1. The author submits a set of PIDs to the Customizer
2. The Customizer accesses each PID and verifies that the identified object exists
3. The Customizer registers a PID at the Registry which points to all submitted PIDs (it creates a PID collection)
4. The Customizer answers to the Author with the registered PID

Post Conditions

- The Auditor can verify that upon requesting the custom collection PID and iterating over its elements, all PIDs originally submitted are included and no additional ones (actually, this is another potential use case dealing with search/access/retrieval)

Notes

- Enabling such custom citations may make it harder to track citations. Current tracking solutions would have to be significantly extended to cover this.

A.9 Modifying data

Goal:

Permit or deny modification or replacement of data.

Actors:

Data provider (human or system), infrastructure (system), data consumer (human)

Summary:

In scientific data infrastructures, individual objects are frequently modified and re-distributed for reasons such as error correction or regular recomputation. Such situations should however be accountable to meet user expectations, ease maintenance and ensure a fundamental level of service quality. Replacing or modifying objects should be properly managed.

This use case is strongly related to the versioning case, where references to old data are kept while new data always receives an individual new PID: If data is allowed to be replaced as part of this use case's main flow, the versioning flow may be kicked off.

Pre Conditions:

The data object is available in the infrastructure and bears a PID

Main Flow (submit):

1. The data provider submits a modified data object to the infrastructure with the specific request that this object should replace the existing object
2. The infrastructure verifies that the object can be replaced or modified
3. If the object is flagged as static, the infrastructure responds to the data provider with an error
4. If the object is not flagged as static, the infrastructure replaces the object with the modified object
5. The infrastructure responds to the user with a success message and the object's PID

Main Flow (lookup):

1. The data consumer requests the data object from the infrastructure using its PID
2. The infrastructure provides information about the object including a statement when it was last modified and whether future changes are possible

Special cases:

One special case might be that the modified object is an extension of the existing object (additional items for a collection, new data in a continuous time series). The infrastructure should treat such cases differently to prevent overwrites of existing data and to be able to tell the data consumer that the object is dynamic.

A.10 Provenance tracing

Goal:

Discover the provenance trace of a data object.

Actors:

Data provider (human or system), repository (system)

Investigator (human), provenance discovery tool (system)

Summary:

When a repository receives data and PIDs are available for base or predecessor objects, these PIDs can be included in the request to deposit the data. The repository will then make sure to establish a provenance trace and provide a tool that any user can use to discover the provenance trace for a given object. Most particularly, this may also work across repositories.

Pre Conditions:

All predecessor objects have received a PID. The new object has not received a PID yet.

Main Flow (create link):

1. The data provider submits a new object (pivot) to the repository together with a list of predecessor PIDs.
2. The repository returns to the user with a PID for the new object.

Implementation notes: The repository may solve this by e.g. creating inter-identifier links, each of which points from the pivot object's PID to one of the predecessor PIDs. The repository may also assign links that point from predecessor PIDs to the pivot PID.

Main Flow (discover):

1. The investigator submits a PID to the provenance discovery tool.
2. The tool replies to the user with a list of direct predecessor PIDs for the object with given PID.
3. The user may optionally restart the use case with one of the listed PIDs.

Implementation notes: The tool will follow the inter-identifier links for one level of depth only. The tool resolves the PID and retrieves the interlinks.