OASIS OPEN PROJECTS

Tracked Resource Set Primer Version 1.0

Project Note 01
16 December 2021

This stage:

https://docs.oasis-open-projects.org/oslc-op/trs-primer/v1.0/pn01/trs-primer.html (Authoritative)
https://docs.oasis-open-projects.org/oslc-op/trs-primer/v1.0/pn01/trs-primer.pdf

Previous stage:

N/A

Latest stage:

https://docs.oasis-open-projects.org/oslc-op/trs-primer/v1.0/trs-primer.html (Authoritative)
https://docs.oasis-open-projects.org/oslc-op/trs-primer/v1.0/trs-primer.pdf

Latest editor's draft:

https://oslc-op.github.io/oslc-specs/notes/trs-primer/trs-primer.html

Open Project:

OASIS Open Services for Lifecycle Collaboration (OSLC) OP

Project Chairs:

Jim Amsden (jamsden@us.ibm.com), IBM
Andrii Berezovskyi (andriib@kth.se), KTH

Editor:

David Honey (david.honey@ibm.com), IBM

Related work:

This specification is related to:

- *OSLC Tracked Resource Set Version 3.0. Part 1: Specification*. Edited by Nick Crossley. Latest Stage: https://docs.oasis-open-projects.org/oslc-op/trs/v3.0/tracked-resource-set.html

Abstract:

This primer serves as a guide to the concepts in the TRS 3.0 specification, and through the use of simple examples, explains how a data provider might expose resources in a TRS and how a TRS client might consume a data provider's TRS.

Status:

This is a Non-Standards Track Work Product. The patent provisions of the OASIS IPR Policy do not apply.

This document was last revised or approved by the Project Governing Board of the OASIS Open Services for Lifecycle Collaboration (OSLC) OP on the above date. The level of approval is also listed above. Check the "Latest stage" location noted above for possible later revisions of this document. Any other numbered Versions and other technical work produced by the Open Project are listed at https://open-services.net/about/.

Comments on this work can be provided by opening issues in the project repository or by sending email to the project's public comment list oslc-op@lists.oasis-open-projects.org.

Citation format:

When referencing this specification the following citation format should be used:

[TRS-Primer-v1.0]

Tracked Resource Set Primer Version 1.0. Edited by David Honey. 16 December 2021. OASIS Project Note 01. https://docs.oasis-open-projects.org/oslc-op/trs-primer/v1.0/pn01/trs-primer.html. Latest stage: https://docs.oasis-open-projects.org/oslc-op/trs-primer/v1.0/trs-primer.html.

Notices

Table of Contents

1. Outline
2. Concepts
3. Starting with an empty TRS
4. Adding TRS events
5. Handling large number of events in a change log
6. Ensuring events are exposed with increasing `trs:order` values
7. Patch events
8. Handling a large TRS base
9. How should TRS clients consume a TRS
    9.1 Initial consumption of a TRS
    9.2 Incremental consumption of a TRS
        9.2.1 Incremental update procedure
10. Making clients tolerant of server restore from backup
11. Change log truncation and rebase
12. General guidance for TRS servers
13. General guidance for TRS clients

Appendix A. Acknowledgements

1. Outline

An OSLC Tracked Resource Set (TRS) provides a mechanism for making a set of resources discoverable and for reporting ongoing changes affecting the set. This allows tools to expose a live feed of linked lifecycle data in a way that permits other tools to build, aggregate, and maintain live, searchable information based on that linked data.

TRS Clients can use Tracked Resource Sets to monitor or replicate some of the server's resources, keeping up to date with changes. A single TRS Client could get information from many different Tracked Resource Sets. For example, a client might get information about requirements from a requirement management TRS, and test cases from a test management TRS, and provide query and reporting capabilities to show traceability between requirements and test cases.

This primer serves as a guide to the concepts in the TRS specification, and through the use of simple examples, explains how a data provider might expose resources in a TRS and how a TRS client might consume a data provider's TRS. The specification remains the definitive source, and where this primer differs from the specification, the specification should be taken as the authoritative reference.

2. Concepts

A *Tracked Resource Set* (TRS) is a resource that describes the set of *Tracked Resources* in the TRS. These are defined by a *Base* set of resources plus *Change Log* that is a set of ordered TRS *Events* that apply incrementally relative to the *Base*. The *Base* represents the set of tracked resources at some specific point in time. The ordering of TRS events after the base is specified by a `trs:order` property in each event. Each TRS event has a type which is one of the following:

1. `trs:Creation` signifies that a tracked resource is created.
2. `trs:Modification` signifies that a tracked resource is modified.
3. `trs:Deletion` signifies a tracked resource is deleted.

Example:

```
Base:
    uri1
    uri2

Change Log Events:
    trs:order=1, trs:Creation uri3
    trs:order=2, trs:Modification uri2
    trs:order=3, trs:Creation uri4
    trs:order=4, trs:Deletion uri1
    trs:order=5, trs:Deletion uri4
```

The base contains tracked resources `uri1` and `uri2`. After the point in time represented by the base, the change log describes the following *ordered* sequence of changes:

1. A new tracked resource `uri3` was created.
2. Tracked resource `uri2` was modified.
3. A new tracked resource `uri4` was created.
4. Tracked resource `uri1` was deleted.
5. Tracked resource `uri4` was deleted.

This represents a TRS that currently contains `uri2` and `uri3`. The ordering of events is critical. For example, if the event for the creation of `uri4` had an order of 6, then the final result is that the TRS contains `uri4`, and this is different from the intended result.

Over time, the change log will continue to accumulate TRS events and without some means to limit its size, would become unmanageable and burdensome for servers to persist, and for clients to consume. The TRS specification addresses this through truncating change logs and rebasing. See Change log truncation and rebase for details. After a change log truncation and rebase, the earlier example might become:

```
Base:
    uri2
    uri3
    uri4

Change Log Events:
    trs:order=5, trs:Deletion uri4
```


3. Starting with an empty TRS

A GET of the TRS URI might return:

```
:trs
    a trs:TrackedResourceSet ;
    trs:base :trsBase ;
    trs:changeLog [
        a trs:ChangeLog ;
    ] .
```

This references a TRS base at `:trsBase` and a in-line change log that has no events. A GET of the TRS base may return a *redirect* to the first page of the TRS base. Typically a TRS base is split into pages as described in Handling a large TRS base. If a client receives such a redirect, it should perform a GET on the URI of the first TRS base page specified by that redirect. The response to the GET of the redirected URI might be:

```
:trsBase
    a    ldp:#DirectContainer ;
    trs:cutoffEvent () ;
    ldp:hasMemberRelation ldp:member ;
    ldp:membershipResource :trsBase .
```

That base resource is a *Linked Data Platform Container* with no members (there are no tracked resources in the base), and with no *cutoff event*. The *cutoff event* comes only becomes relevant once a TRS base has been updated to include events from a change log. See Change log truncation and rebase for details.

A client reading the TRS for the first time would:

1. GET the TRS and determine the URI of the TRS base.
2. GET the TRS base and record the tracked resources in the TRS base. In this case, this is an empty set.
3. Look at the change log in the TRS from step 1 and process the events there, starting with the highest `trs:order`. Since there are no events in the change log, there are no tracked resources in the change log.

The result is the client sees a TRS with no member tracked resources.

4. Adding TRS events

Consider that our example data provider adds a tracked resource `:tracked1`. The change log should contain a `trs:Creation` event (or a `trs:Modification` event) for `:tracked1`. While a `trs:Creation` event more accurately describes the creation of a tracked resource, clients should treat an initial `trs:Modification` event equivalently to a creation event.

A GET on `:trs` might return:

```
:trs
    a trs:TrackedResourceSet ;
    trs:base :trsBase ;
    trs:changeLog [
        a trs:ChangeLog ;
        trs:change <urn:example:example.com:2021-02-05T17:39:33.000Z:1> .
    ] .

<urn:example:example.com:2021-02-05T17:39:33.000Z:1>
    a trs:Creation ;
    trs:changed :tracked1 ;
    trs:order "1"^^xsd:integer .
```

There are several things worthy of note:

1. The ordering of the events referenced in the change log is determined by the `trs:order` property of each of the referenced events. It's required that events describing later changes must have unique and higher order values than earlier events. `trs:order` values are not required to be *consecutive*, so a client cannot use the order property to check for missing events.
2. The RDF data type for `trs:order` should be `xsd:integer` (and not `xsd:int`) because, over time, new order values could become arbitrarily large and beyond the range of, for example, a 32-bit integer.
3. The URI of an event in the change log should be stable over time and be guaranteed to be unique. For example, if a TRS gets truncated and/or rebased, existing events in the change log should retain the same URI. It should never be the case that a new event reuses a URI that was used even if the server is restored from a backup. The URI of an event should therefore not be solely based on its `trs:order` value. See Making clients tolerant of server restore from backup for further details.

The TRS base was not changed by this operation.

A client that had earlier read the empty TRS would now:

1. GET the TRS.
2. Look at the change log and process the events there, starting with the highest `trs:order` continuing until the last previously-processed event (in this case none) was encountered. It records the creation of `:tracked1`.
3. Record the URI of the most recently processed event. In this example, that is `urn:example:example.com:2021-02-05T17:39:33.000Z:1`.

The client should therefore see the TRS as having members `:tracked1`.

Now consider that our example data provider adds a tracked resource `:tracked2`.

A GET on `:trs` might return:

```
:trs
    a trs:TrackedResourceSet ;
    trs:base :trsBase ;
    trs:changeLog [
        a trs:ChangeLog ;
        trs:change <urn:example:example.com:2021-02-05T17:39:33.000Z:1> ;
        trs:change <urn:example:example.com:2021-02-05T17:40:12.000Z:2> .
    ] .

<urn:example:example.com:2021-02-05T17:39:33.000Z:1>
    a trs:Creation ;
    trs:changed :tracked1 ;
    trs:order "1"^^xsd:integer .

<urn:example:example.com:2021-02-05T17:40:12.000Z:2>
```



```
a trs:Creation ;
trs:changed :tracked2 ;
trs:order "2"^^xsd:integer .
```

The client that previously read the TRS would:

1. GET the TRS.
2. Look at the change log process the events there, starting with the highest `trs:order` continuing until the last previously-processed event (in this case `urn:example:example.com:2021-02-05T17:39:33.000Z:1`) was encountered. It records the creation of `:tracked2`.
3. Record the URI of the most recently processed event. In this example, that is `urn:example:example.com:2021-02-05T17:40:12.000Z:2`.

The client should therefore see the TRS as having members `:tracked1` and `:tracked2`.

5. Handling large number of events in a change log

Servers should be able to handle a large number of TRS events. Returning all the events in the in-line change log of the TRS resource would not be practical or scaleable. TRS clients should not be expected to handle arbitrarily large responses to a GET of a TRS. The TRS specification describes a mechanism for segmentation of a change log. The change log embedded in the response to a GET of the TRS includes a limited number of the most recent events, and then includes a `trs:previous` referencing a URI that will return the next *segment* of the change log. A GET of that page would also return a change log referencing the next set of older events , and a `trs:previous` referencing a URI that will return the next segment. A client can thus perform a GET of each segment, and process a response whose size is constrained by some implementation-defined limit on the number of events per segment.

Let's imagine that our example data provider adds another tracked resource `:tracked3`. A GET of the TRS might respond with:

```
:trs
    a trs:TrackedResourceSet ;
    trs:base :trsBase ;
    trs:changeLog [
        a trs:ChangeLog ;
        trs:change <urn:example:example.com:2021-02-05T17:40:12.000Z:2> ;
        trs:change <urn:example:example.com:2021-02-05T17:42:55.000Z:3>
        trs:previous <trsChangeLog?before=2>
    ] .

<urn:example:example.com:2021-02-05T17:40:12.000Z:2>
    a trs:Creation ;
    trs:changed :tracked2 ;
    trs:order "2"^^xsd:integer .

<urn:example:example.com:2021-02-05T17:42:55.000Z:3>
    a trs:Creation ;
    trs:changed :tracked3 ;
    trs:order "3"^^xsd:integer .
```

In this case, the embedded change log only includes the most recent 2 events (`trs:order` values 2 and 3). If a client needs to examine older events, it should perform a GET on the previous segment `trsChangeLog?before=2` which might provide:

```
<trsChangeLog?before=2>
    a trs:ChangeLog
    trs:change <urn:example:example.com:2021-02-05T17:39:33.000Z:1> .

<urn:example:example.com:2021-02-05T17:39:33.000Z:1>
    a trs:Creation ;
    trs:changed :tracked1 ;
    trs:order "1"^^xsd:integer .
```

In this way a client can iterate over the change log, getting previous segments until it encounters a change event it has already processed or there are no more segments to process. Change log segments are ordered from the most recent change in the first page, going back in time with each page, with the oldest change in the last page. A change log page must not contain order numbers higher than any in the more recent pages already encountered - that is, each page of events must be strictly older than all previously read pages.

What maximum number of events should be served in the in-lined change log, and in each subsequent segment?

If the maximum number of events per segment is too small, clients will have to make many more GET requests to process events, and if network latency is significant, this could be slower than using larger segments. Typically, TRS clients will poll a TRS server on a regular basis. If the maximum number of events in the first segment is too large, a client will get a TRS with a large embedded change log, of which only the most recent events in the last poll interval will be processed. Ideally, for the first segment, the maximum number of events should be larger than the average number of new events per poll interval, but not so large that the response is too large for a client to process. Subsequent segments might be larger, and tend only to be read by TRS clients processing the TRS for the first time. An implementation should probably have a configurable property for each of the initial change log size and the subsequent segment size. A default value of 1000 events might be an appropriate starting point for both properties. They can be reviewed and adjusted based on event rate per poll interval and the total number of TRS events.

It is required that segmentation of the change log is *stable*. That is, a client that starts processing the change log embedded in the TRS and processes each `trs:previous` segment should not miss any tracked resources between those segments, even if new events are added to the TRS.

One implementation approach for servers is to persist each TRS event as a row in a table of events. This allows servers to handle a very large number of events. A GET on the TRS can get the first segment of events by querying for the most recent *N* events and include those in the change log embedded in the RDF of the TRS. The `trs:previous` can use a URI that includes a query parameter that references the oldest `trs:order` that was included. A GET on that URI can then query for the next *M* events whose `trs:order` is greater than that value. Such queries tend to scale well, and provide the stable segmentation required by clients.

6. Ensuring events are exposed with increasing `trs:order` values

The TRS specification requires that TRS events are exposed with increasing `trs:order` values. Consider a non-compliant TRS server that exposes a change log over time as follows:

```
Time 10.0 seconds
    <event100Uri> trs:order=100
    <event101Uri> trs:order=101

Time 15.0 seconds
    <event100Uri> trs:order=100
    <event101Uri> trs:order=101
    <event103Uri> trs:order=103

Time 20.0 seconds
    <event100Uri> trs:order=100
    <event101Uri> trs:order=101
    <event102Uri> trs:order=102
    <event103Uri> trs:order=103
```

A TRS client might consume this over time as follows:

1. At time 11.0 seconds, GET the TRS and embedded change log. Consume events with orders 100, and 101. Record the *sync point* as `event101Uri`.
2. At time 16.0 seconds, GET the TRS and embedded change log. Consume events after last processed event `event101Uri`, which consists of one event with order 103. Record the *sync point* as `event103Uri`.
3. At time 21.0 seconds, GET the TRS and embedded change log. Consume events after last processed event `event103Uri`, of which there are none.

This could result in a client missing processing the event with `trs:order=102` because that event was exposed in the change log **after** the event with `trs:order=103` was exposed. This is why the specification requires that events must be *exposed with increasing `trs:order`*.

Achieving this requirement is sometimes not straightforward. A server persisting its tracked resources in a relational database may want to use a single atomic transaction to save a resource's changes to the database and to create a TRS event describing those changes. Consider what might happen if the `trs:order` value is allocated within that transaction, either explicitly in code, or being automatically allocated using an auto-incrementing sequence provided by the relational database. If there are two such operations `O1` and `O2` running concurrently, even if the allocation occurs in the order `O1` followed by `O2`, there is no guarantee that the *commits* of those transactions will complete in that order. There is a risk that `O2` might complete commit before `O1`. Many implementations, may use *read committed* mode, which means that data changed in a transaction only becomes visible outside of that transaction once it has been successfully committed. So there is a real risk that the TRS event for `O2` with a higher `trs:order` might be exposed **before** the TRS event for `O1` with a lower `trs:order`.

One approach is to create *private* TRS events within transactions, and then have separate processing after transaction commit that is single-threaded that allocates the `trs:order` value and makes the event *public*.

It is also recommended that for resilience against TRS servers that do not address this issue, TRS clients be tolerant of such TRS server defects. A more tolerant client might record the last *N* ordered URIs of processed events as the *sync point*, and be tolerant of incorrect ordering within a window of *N* events. In our example, for *N*=2 the sequence might be:

1. At time 11.0 seconds, GET the TRS and embedded change log. Consume events with orders 100, and 101. Record the *sync point* as [`event100Uri`, `event101Uri`].
2. At time 16.0 seconds, GET the TRS and embedded change log. Check for any events in between the `trs:order` for `event100Uri` and `event101Uri`. There are none. so process the next event after `event101Uri`, which is `event103Uri`. Record the *sync point* as [`event101Uri`, `event103Uri`].
3. At time 21.0 seconds, GET the TRS and embedded change log. Check for any events in between the `trs:order` for `event101Uri` and `event103Uri`. There is one missed event `event102Uri`. Process events starting with this missed event and then move forwards to process events with higher `trs:order` values.

Note that if missing earlier events are detected due to incorrect order of publication, and those events are `trs:Modification` events with

patches (see Patch events), a client should ignore the patch and fetch the complete resource.

In this way, the TRS client would be tolerant of small time windows in which the `trs:order` values might not be exposed in increasing order.

7. Patch events

Consider a tracked resource whose RDF content is very large, and when modified, the changes to that RDF content represent a small fraction of the overall content. If a TRS server generates a standard `trs:Modification` event, then a client has to fetch the entire content of the changed tracked resource. The TRS specification defines a *patch* mechanism so that a modification event can include the delta of RDF changes from the current state defined by the event and a previous event, which might be another patch, a regular `trs:Modification` or `trs:Creation` event, or an entry in a TRS base page. This reduces the burden on a TRS client in maintaining the content of the tracked resource.

Consider a tracked resource that is a container referencing other resources whose current etag is `15687ds9gha6s7`:

```
@prefix dcterms: <http://purl.org/dc/terms/>.
@prefix ldp: <http://www.w3.org/ns/ldp#>.
<https://a.example.com/container/a1>
   a ldp:BasicContainer;
   dcterms:title "Container A1";
   ldp:member <https://a.example.com/version/s/143>;
   ldp:member <https://a.example.com/version/r/577>;
   ldp:member <https://a.example.com/version/t/033>.
```

The container is updated, replacing member `https://a.example.com/version/r/577` with member `https://a.example.com/version/r/578` giving a resource with etag `285d4h2ffgddd9` and the following RDF content:

```
@prefix dcterms: <http://purl.org/dc/terms/>.
@prefix ldp: <http://www.w3.org/ns/ldp#>.
<https://a.example.com/container/a1>
   a ldp:BasicContainer;
   dcterms:title "Container A1";
   ldp:member <https://a.example.com/version/s/143>;
   ldp:member <https://a.example.com/version/r/578>;
   ldp:member <https://a.example.com/version/t/033>.
```

A TRS server might create a `trs:Modification` event describing this change as follows:

```
@prefix trs: <http://open-services.net/ns/core/trs#>.
@prefix trspatch: <http://open-services.net/ns/core/trspatch#>.
<urn:example:6e8bc430:a.example.com:2014-04-28T17:39:32.000Z:102>
   a trs:Modification;
   trs:changed <https://a.example.com/container/a1>;
   trs:order "102"^^xsd:integer;
   trspatch:beforeEtag "15687ds9gha6s7";
   trspatch:afterEtag "285d4h2ffgddd9";
   trspatch:rdfPatch
     """
      D <https://a.example.com/container/a1> <http://www.w3.org/ns/ldp#member> <https://a.example.com/version/r/577> .
      A <https://a.example.com/container/a1> <http://www.w3.org/ns/ldp#member> <https://a.example.com/version/r/578> .
     """.
```

The event contains a `trspatch:rdfPatch` that specifies:

1. The triple `<https://a.example.com/container/a1> <http://www.w3.org/ns/ldp#member> <https://a.example.com/version/r/577>` is deleted.
2. The triple `<https://a.example.com/container/a1> <http://www.w3.org/ns/ldp#member> <https://a.example.com/version/r/578>` is added.

A TRS patch has to specify the triples (RDF statements) that are to be deleted and/or added. The use of RDF blank nodes in patches is problematic since they are not guaranteed to be stable over time. Hence TRS patches should not be used to represent deltas where the changes involve blank nodes.

Patch events may also reference a `trspatch:createdFrom` as a means of representing a new resource whose RDF is formed from another *created from* resource plus the patch delta. Consider the case where version 2 of some artifact is created from version 1. Version 2 might inherit a significant set of common RDF data from version 1. An application might represent the TRS event for version 2 as a patch specifying a `trspatch:createdFrom` referencing version 1 and the RDF delta between version 2 and version 1.

When there is a chain of patch events, the `beforeEtag` of a patch must match the `afterEtag` of the immediately preceding patch. For

patch events to work reliably, it's important that the patch events are complete, and ordered correctly, and that TRS clients can validate this. This becomes increasingly important as the chain of patch events grows in size. If a TRS client detects that the `beforeEtag` of a patch does not match the `afterEtag` of an immediately preceding patch, it should assume the patch is untrustworthy and fetch the complete the resource, in effect, treating the `trs:Modification` event as if the patch were absent.

For TRS servers, the implementation should consider limiting the length of patch chains. For example, a TRS server might emit a normal `trs:Modification` event after *N* patch events for that tracked resource in order trigger a full fetch of the tracked resource periodically by clients.

8. Handling a large TRS base

Just as with the change log, it's not practical for a GET on the TRS base to return all the members of the base where the number of members is large. Typically, TRS servers implement a *paged base* where a GET on the TRS base returns a redirect to the first base page. Where there are additional pages, the response to the GET of a base page must include an LDP `Link` header of the form `Link: <nextPageUri# >; rel="next"`.

For example, the first GET of the TRS base `:trsBase` might return a redirect to the first base page at `<trsBase/1>`, and a GET from `<trsBase/1>` might return:

```
:trsBase
    a    ldp:#DirectContainer ;
    trs:cutoffEvent () ;
    ldp:hasMemberRelation ldp:member ;
    ldp:membershipResource :trsBase ;
    ldp:member <uri1> , <uri2> , ... <uri1000> .
```

The response might contain the header `Link: <trsBase/2>; rel="next"`. A GET of the next page at `<trsBase/2>` might return:

```
:trsBase
    a    ldp:#DirectContainer ;
    ldp:hasMemberRelation ldp:member ;
    ldp:membershipResource :trsBase ;
    ldp:member <uri1001> , <uri1002> , ... <uri2000> .
```

The response might contain the header `Link: <trsBase/3>; rel="next"`. Note that the 2nd and subsequent pages do not need to include the `trs:cutoffEvent` property.

It's important that the paging of the TRS base is *stable*. A client should be able to iterate over all the base pages and be guaranteed to see all the entries up until the moment that the last page is returned. This means that new entries to the base should only be added to the last page, creating a new page as required. Base page members should only be removed during change log truncation and rebase.

How many members should be included in each TRS base page?

If each base page is small, a TRS client that needs to process the base will have to perform many more GETs, and with significant network latency, this could adversely affect performance. Conversely, a base page that is too large could take too long to construct and return, resulting in socket timeouts, or result in a response that is too large for a client to handle. Ideally the base page size should be configurable. A suggested starting point is 1000 members per base page.

There are several implementation designs that a TRS server might consider:

1. Persist TRS base pages explicitly. Each base page is a persisted entity, and references the members of that page.
2. Persist base members as separate rows in a table and perform paging of the base at run time. In order to achieve stable paging, each member will need to be ordered, and a `trs:order` value or a member creation time might provide an appropriate sort order.

Option 1 is likely to be a more compact representation, and it makes it very easy to serve any base page via a REST service. However, when base members are removed during change log truncation, this is likely to lead to only partially populated, or possibly even empty, base pages. This kind of base page fragmentation can be avoided with option 2. However, paging could become unstable during change log truncation. The deletion of a base member would shift the newer base members up in the query of members. So achieving stable paging may become more complex.

TRS servers might consider providing a way to completely recompute the TRS, creating a new TRS base from scratch. It's important that while such a base reconstruction in progress, the existing base and change log remain viable. A server might achieve this by constructing the new base as private data, temporarily suspending any updates to the active base, and then deleting the old base and exposing the new base in a single atomic transaction. However, there is danger that the new base might include members that were absent from the old base, or have members that are absent that were present in the old base. A TRS client that is running incrementally would not know that the base needed to be reprocessed. One way of avoiding this is for the recompute of the TRS to also clear the TRS change log and use a base `cutoffEvent` of `rdf:nil`. A client should detect the *sync point URI* does not exist in the change log and reprocess the entire TRS.

9. How should TRS clients consume a TRS

TRS clients should maintain a *sync point URI* that records the URI of the latest change event that has been processed. This will be required for incremental consumption.

9.1 Initial consumption of a TRS

1. GET the TRS to retrieve the URI of the TRS base.
2. GET successive pages of the base, processing each member resource as required.
3. Invoke the procedure described in Incremental Update Procedure with the *sync point URI* set to the `trs:cutoffEvent` property on the first page of the based (which could be `rdf:nil`).

9.2 Incremental consumption of a TRS

1. Invoke the the procedure described in Incremental Update Procedure with the *sync point URI* set to the URI of the most recent previously processed change event.

9.2.1 Incremental update procedure

1. GET the TRS resource to get the first segment of the change log.
2. Process events from the newest (highest `trs:order`) to oldest.
3. For `trs:Creation` events and `trs:Modification` events without a *patch*, process the tracked resource. Typically a client will fetch the complete RDF of that tracked resource. Record that tracked resource as processed. If an older event for that tracked resource is encountered, ignore it.
4. For a `` `trs:Modification `` event with a *patch*, record the event for later *patch processing*. This is required because patch events have to be processed in the order oldest to newest.
5. For a `trs:Deletion` event, process the tracked resource. A client might remove any persisted content for that resource. Record that tracked resource was processed. If an older event for that tracked resource is encountered, ignore it.
6. If the event at the *sync point URI* is encountered, the first pass of processing terminates and proceeds to *patch processing*. If the end of the current change log segment is reached, then fetch any next segment and continue processing as above.

At this stage, all of the non-patch events have been processed. If there are any modification events with patches to be processed, for each tracked resource, process the patches ordered from the oldest to the newest. A client might take the RDF of the tracked resource was it was last updated and persisted, and then apply the deletions and additions of each patch. Clients should check the `beforeEtag` of a patch matches the `afterEtag` of any prior patch. If these do not match, a client should fetch the entire tracked resource.

10. Making clients tolerant of server restore from backup

Consider a TRS whose change log contains events for `trs:order` values 1 to 1000. A TRS client might read that TRS and record the last processed event URI as `<event1000Uri>`. On the next poll, the TRS client will be looking for events with a `trs:order` greater than the event `<event1000Uri>`. If the TRS server was restored from a previous day's backup, the restored TRS might only have TRS events for `trs:order` values 1 to 900. If a TRS clients polls that TRS at that time, it should detect that event `<event1000Uri>` does not exist, and reprocess the entire TRS.

However, consider the case where the server created an additional 200 events after restore from backup but before the next poll by a TRS client. If that results in `<event1000Uri>` being used for one of those new events, the clients incremental TRS update procedure is likely to miss processing events. This is why TRS servers should guarantee unique event URIs, even if the `trs:order` value is reused after restore from backup. A TRS event URI must not rely on the `trs:order` alone to achieve uniqueness. Servers might consider using a GUID persisted as data on events, or embedding a date time string with the `trs:order` in the URI.

11. Change log truncation and rebase

TRS servers should not allow a TRS change log to grow without limit. A server should provide a means to *truncate* the change log in a well-behaved manner so that TRS base is updated to reflect new members, and deleted old members are removed from the base. A server might implement an automatic periodic truncation and base update, and/or provide a capability for an administrator to initiate this manually.

Consider a TRS with an empty base, and a change log with 5 TRS events. A GET on the TRS might return:

```
:trs
    a trs:TrackedResourceSet ;
    trs:base :trsBase ;
    trs:changeLog [
        a trs:ChangeLog ;
        trs:change <urn:example:example.com:2021-02-05T17:39:33.000Z:1> ;
        trs:change <urn:example:example.com:2021-02-05T17:40:12.000Z:2> ;
        trs:change <urn:example:example.com:2021-02-05T17:42:55.000Z:3> ;
        trs:change <urn:example:example.com:2021-02-06T11:09:11.000Z:4> ;
        trs:change <urn:example:example.com:2021-02-06T11:17:42.000Z:5> .
    ] .

<urn:example:example.com:2021-02-05T17:39:33.000Z:1>
    a trs:Creation ;
    trs:changed :tracked1 ;
    trs:order "1"^^xsd:integer .

<urn:example:example.com:2021-02-05T17:40:12.000Z:2>
    a trs:Creation ;
    trs:changed :tracked2 ;
    trs:order "2"^^xsd:integer .

<urn:example:example.com:2021-02-05T17:42:55.000Z:3>
    a trs:Deletion;
    trs:changed :tracked1 ;
    trs:order "3"^^xsd:integer .

<urn:example:example.com:2021-02-06T11:09:11.000Z:4>
    a trs:Modification ;
    trs:changed :tracked2 ;
    trs:order "4"^^xsd:integer .

<urn:example:example.com:2021-02-06T11:17:42.000Z:5>
    a trs:Creation ;
    trs:changed :tracked3 ;
    trs:order "5"^^xsd:integer .
```

The TRS base is empty, so a GET might return:

```
:trsBase
    a    ldp:#DirectContainer ;
    trs:cutoffEvent () ;
    ldp:hasMemberRelation ldp:member ;
    ldp:membershipResource :trsBase .
```

When events from the change log are processed to update the base, the TRS base must have a *cutoff event* that references the URI of the event in the change log that was most recently processed for the TRS base. This event referenced by the cutoff should not be processed as part of the change log processing as part of the Initial consumption of a TRS.

A TRS server may later remove some of the events that were processed for the base and that are older than the base cutoff. The change log must never be truncated to the extent that the cutoff event itself disappears from the change log.

So if a TRS server fully processed the events, updated the base, and truncated the change log, the resultant TRS would be as follows:

```
:trs
    a trs:TrackedResourceSet ;
    trs:base :trsBase ;
    trs:changeLog [
```



```
        a trs:ChangeLog ;
        trs:change <urn:example:example.com:2021-02-06T11:17:42.000Z:5> .
    ] .

<urn:example:example.com:2021-02-06T11:17:42.000Z:5>
    a trs:Creation ;
    trs:changed :tracked3 ;
    trs:order "5"^^xsd:integer .
```

and the TRS base would be:

```
:trsBase
    a    ldp:#DirectContainer ;
    trs:cutoffEvent <urn:example:example.com:2021-02-06T11:17:42.000Z:5> ;
    ldp:hasMemberRelation ldp:member ;
    ldp:membershipResource :trsBase ;
    ldp:member :tracked2, :tracked3 .
```

Note that the `trs:cutoffEvent` references the URI of the the event in the change log that was last processed for the base. Tracked resource `:tracked1` is not in the TRS base because the most recent event for it in the change log prior to truncation was a `trs:Deletion`.

In practice, TRS servers should **not** perform such a brutal truncation and rebase. A TRS client may not have processed all the events that were in the change log prior to truncation. In order to provide adequate response to client requests, a TRS server needs to allow those clients sufficient time to read the base, the change log, and process the set of tracked resources. However, the data volumes and timescales involved in TRS processing are likely to vary between servers for different applications. A server representing Amazon transactions might have many events per second, while a server representing exhibits at a museum might have a few events per month. The cost of processing a single event is also likely to vary between applications; reading a new or modified resource with 5 RDF properties will take less time than reading one with 5,000 properties.

For these reasons, the TRS specification does not impose specific constraints over the length of time for which a TRS base must remain readable, nor what the degree of overlap should be between a base and a corresponding change log. A server implementing TRS must consider, and should document, the quality of service it will provide in terms of the size of pages in the base or change log, how long base pages are kept, how long change events are kept, and the minimum period for which change events behind the latest base cutoff are kept.

Related to this is the issue of deletions in a TRS base arising from change log truncation. Consider a TRS client reading a TRS for the first time while a `trs:Deletion` event is being truncated from the change log and the corresponding member is being removed from a base page. A client may have processed the base page before that member was removed, but if, some considerable time later, the event was removed from the change log, the client would not see the deletion. The result would be a client that erroneously records that tracked resource as being in the TRS. To avoid this, it is recommended that truncation and base update is processed in two phases:

1. Events older than *N* days, are processed so that the resources are added or removed from base pages are required. The cutoff event of the base is updated to the URI of the most recent event that was processed for the base. However, those events are **not** deleted from the change log at this time.
2. Events that were processed for base page updates in step 1, are deleted from the change log at least *M* days after they were processed in step 1.

It is recommended that *N* and *M* are configurable properties, and that a server documents the default values of these properties. Suitable default values depend on the frequency of events and how often TRS clients poll the TRS. A suggested starting point is for *N*=7 days and for *M*=14 days. This would keep events in the change log for at least 21 days. Providing that a TRS client can process the base pages and change log in less than 21 days, it will not miss deletions of tracked resources.

12. General guidance for TRS servers

- A TRS server should restrict itself to a small number of TRSs.
- If a TRS server provides multiple TRSs, any single tracked resource should be a member of only one of those TRSs. In other words, each TRS should be a disjoint set of tracked resources. For example, a requirements management application might publish a TRS for requirements resources, and a separate TRS for tracked resources representing users and user groups.
- The members of a TRS should be tracked resources that are control of the server. In the case of a TRS adapter for an existing application, the TRS adapter should only expose that application's resources. In other words, the subjects of a lifecycle tool claims should be its own resources, as opposed to resources of some other lifecycle tool.

Nick: this is a Jim de Rivieres comment with which many of us do not agree. There are valid reasons for a TRS feed for 'foreign' resources, perhaps for some kind of adapter.

- A TRS should expose regular linked lifecycle data resources, and associated resources such as resource shapes, including properties and their referenced allowed values. This allows a TRS client to determine a *type system model* for the data that might be useful for constructing and executing reports.
- A TRS server should report operations on tracked resources as new TRS events within a short time of the changes being persisted in the server. This allows TRS clients to obtain a live feed of changes in nearly real-time. If a server takes longer than a second or two to publish TRS events, the server should document the quality of service it supports so that clients understand the latency in getting events for changes.
- A TRS server should maintain a change log of events for a certain duration, and this should be configurable. A server should document the default duration. This allows a TRS client to be offline for less than that duration and then resume processing of the TRS without missing events.
- A TRS server should provide some means of truncating the change log to limit the number of TRS events a new TRS client has to process.
- TRS servers should create a TRS event for a tracked resource in the same atomic transaction in which the tracked resource itself is created, modified, or deleted.
- TRS servers should use URIs for events that are unique and will not be reused for later events, even if the server is restored from backup.
- TRS servers should limit the maximum size of patches in `trs:Modification` events.
- TRS servers should consider limiting the length of patch event chains, and consider periodically using a `trs:Modification` event without a patch to require that a client fetch the complete resource. This helps to improve the robustness of the TRS for clients.

13. General guidance for TRS clients

- Clients should read the base and change log as quickly as they can, and not delay for extended periods of time between reading the base and the change log, or between reading pages of the base.
- Clients are expected to repeat the incremental update procedure on a regular basis to keep up with changes, perhaps as often as every minute, and at least often enough so that events are not lost by change log truncation. The poll interval should be configurable, and clients should document the default.
- Clients should be defensive against imperfect or untrustworthy TRS servers. A client that is persisting data about tracked resources should place limits on the number of tracked resources, and the data representation size of each tracked resource. For example, if a data provider exposes a tracked resource whose RDF content includes a very large RDF literal, a client should limit the impact of data, either truncating the literal in some fashion, or discarding that data.
- Clients that encounter error conditions in a TRS should report those errors in an appropriate way that allows such errors to be made visible, and for the TRS server development team to investigate the issue.
- Clients should treat `trs:Creation` events in the same way as `trs:Modification` events.
- Clients should consider fetching tracked resources concurrently to improve performance.

Appendix A. Acknowledgements

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Project Governing Board:

James Amsden, IBM (co-chair)
Andrii Berezovskyi, KTH (co-chair)
Axel Reichwein, Koneksys

Techical Steering Committee:

James Amsden, IBM
Andrii Berezovskyi, KTH
Axel Reichwein, Koneksys

Additional Participants:

David Honey

