Tethys R Client

CONTENTS

1 INTRODUCTION

Analysis of Tethys data in R begins with accessing the data. There are several ways to do this:

- Run a query in the web client interface (e.g., http://mytethys.domain:9779/Client[1]).
  Results may be saved in either the R data.tree format or as XML and then imported. Use the Save to functionality to select the format and save it. See chapter 3 for details on working with XML and data.tree.
- Save output from the Data Explorer program as a comma separated value file and load it using R's read_csv function. Data Explorer provides a simplified view of the Tethys data and reduces everything to a table. This reduces the complexity of working with the data, but does not offer as much functionality.
- Use the RClient Tethys class to execute queries with the R programming language (chapter 2). Results are returned as text that can then be parsed using XML libraries (see chapter 3).

The following Comprehensive R Archive Network (CRAN) library packages should be installed:

- data.tree provides a relatively straightforward method for representing the hierarchical data contained in an XML document. We recommend this as a method for accessing Tethys data that has not been preprocessed into a table suitable for a data frame.
- xml2 – A library to work with XML files
- XML – A library to parse XML files

[1] Update this URL for the Tethys server that you are using.

2 USING THE TETHYS R CLIENT

Code for accessing Tethys data from R is available in the RClient subfolder of the Tethys distribution. The R client is preliminary and is not currently set up as an R package. To use it, you must source the tethys.r.

```
> source("path/to/tethys/RClient/tethys.r")
```

This will make the Tethys class available. It is a reference class, so methods are separated from the class instance by a $ sign. To being, create an instance of the Tethys object. You must know the machine identifier for the Tethys server, the port on which it is running, and whether or not the server is running in encrypted mode. Most Tethys users do not turn on encryption as it requires a signed security certificate to work well.

```
>  t <- tethys$new("my.server.gov", port=9779, secure=FALSE)
```

If you are running the server on the same machine as the R process and you have not changed the port or security settings, you can use the default arguments:

```
> t <- tethys$new()
```

To test communication, you can use the ping() method which returns TRUE if we can communicate with the server:

```
>  t$ping()
[1] TRUE
```

Should ping return FALSE, there are several likely causes:

- Tethys may be down. Try to connect from another machine if possible. The easiest way to do this is with the web client as it does not require software installation. Navigate to http://server.machine.name:9779/Client . If you can load the page on your machine, you may have an R issue or the R program may need a firewall exemption. If you cannot, try the same test on a machine that has been able to connect to the server in the past.
- Verify that Tethys is running on the server. Your administrator should be able to check this.

2.1 EXECUTING QUERIES

RClient supports two forms of query, simple query language that is a simple text-based set of constraints and data to return, and XQuery, a standard mechanism for querying XML databases. Simple query language is only capable of a subset of XQuery's functionality, but it is suitable for most users and does not require learning XQuery.

2.1.1 Simple declarative query language

Tethys supports the following methods that are designed to enable users to write queries without having to learn a complicated query language:

- getDeployments – Retrieve information about instrument deployments.
- getCalibration – Retrieve information about instrument calibrations.
- getDetectionEffort – Retrieve information about which taxa we have been looking for, where we have been looking, and how we have been looking.
- getDetections – Retrieve information about what we have actually found.
- getLocalizationEffort – Retrieve information about where and how we have attempted to localize sounds.
- getLocalizations - Retrieve localization information.

All of these rely on a simple syntax that indicates a set of criteria that must be met and optionally a set of information that will be returned in the query. For each type, there is a default set of values that are returned, but these can be customized to contain specific types of data.

The first argument of all of these use the same syntax to specify the criteria being selected. For example, if we wanted to see all information about instruments deployed at 800 m depth or deeper that had a sample rate of greater than or equal to 192 kHz, we could provide the following string to getDeployments:

```
# Throughout these examples, we will assume that variable t contains
# an instance of the Tethys object.
> xml <- t$getDeployments("DeploymentDetails/ElevationInstrument_m <= -800 and
    SampleRate_kHz >= 192")
```

The results could then be processed using the methods described in the section on working with XML. In order to know how to construct a query you first need to have an understanding of the data that are being queried. These can be found by asking to have the list of elements associated with a collection displayed:

```
# Open a schema in a web browser.  Argument should be the root element of a
# collection document:
# Deployment, Calibration, Ensemble, Detections, or Localizations
> t$schema("Deployment")
```

This will open up a table in a web browser that lists the names of elements in the schema, their type, the minimum and maximum number of times they can appear as a child of their parent, and a description of for what the name is used. Descriptions of fields that we consider to be self-describing may not have a description. Valid arguments to the schema method can be found in Table I.

Table I – Collections and the root element used with the schema method.

Collection	Root element	Collection purpose
Calibrations	Calibration	Results of transducer (microphone, hydrophone) calibration.
Deployments	Deployment	Details about instrumentation that has been deployed.
Detections	Detections	Details about passive acoustic monitoring detection effort and results.
Localizations	Localizations	Details about localization effort and results.
Ensembles	Ensemble	Virtual instruments consisting of multiple deployments.

Comparisons of fields and values are based on operators that are common in many languages (Table II) and the **field name must appear on the left-hand side of the comparison**.

Table II – Simple query language comparison operators.

=	equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

Multiple comparisons must be joined by the word "**and**," indicating that all conditions must be true. The "or" keyword is not available in the simple query language[2].

Field names consist of a path of field names separated by forward slashes (/). For example, the deployment sample rate field path is:

Deployment/SamplingDetails/Channel/Sampling/Regimen/SampleRate_kHz.

When a field only appears once in the schema, the path can be omitted and one can just write SampleRate_kHz. This is not always the case. For example, in the deployments collection, Longitude fields exist in multiple places:

Deployment/DeploymentDetails/Longitude,
Deployment/RecoveryDetails/Longitude, and
Deployment/Data/Tracks/Point/Longitude.

Writing constraints such as "Longitude > 30" are not sufficient for the parser to determine which Longitude is desired. Either use the entire path, or a distinct subset of it, e.g., "DeploymentDetails/Longitude > 30". Failure to do so will produce an error.

It is possible to use constraints across schemata. For instance, one might be querying detections, and wish to add a constraint that they be restricted to a specific deployment Site. The Site field can be used in the query. If a field appears in both schemata, it will default to the one in the schema most associated with the query. For example, in the deployments collection, Deployment/DeploymentId is a number associated with the N^{th} deployment at a site or the N^{th} set of deployments. In the detections schema, Detections/DataSource/DeploymentId indicates with which deployment the detections are associated, that is the deployment where Detections/DataSource/DeploymentId = Deployment/DeploymentId. Consequently, if one wishes to filter by deployment's DeploymentId, one must use "Deployment/DeploymentId" in the query. When fields are referenced from multiple collections, constraints are automatically added to the query to ensure that related documents are matched.

If a user simply writes a set of constraints, a default set of return values appropriate to the type of query will be returned. However, this can be overridden by using the **return** keyword and a comma-separated list of

[2] Disjunction (or) is permitted in XQuery, the query language to which the simple query language is translated. However, disjunction is only allowed when a pair of elements share the same common ancestor. Allowing this would require additional complexity that we do not with to introduce into simple queries. If this is really needed, either learn XQuery or use the plan argument in in t$QuerySimple to translate to XQuery and modify the result. (QuerySimple is called by all of the "get" functions described above.)

elements to be returned. For example, suppose we were interested in a list of all deployments in water less than 100 m (instrument elevation > -1, their sample rate, and deployment depth. The following query will accomplish this:

```
> xml <- t$getDeployments('DeploymentDetails/ElevationInstrument_m > -100 return
    Id, SampleRate_kHz, SampleBits, DeploymentDetails/ElevationInstrument_m')
```

As a final example, we ask for which species in the Northern hemisphere did we have effort where detections were reported as binned presence:

```
> xml <- t$getDetectionEffort('Granularity = "binned" and DeploymentDetails/Lati
   tude > 0')
```

Note that even though we queried on detections, we were able to deployment fields to filter where we had detection effort. The XML produced by this looks like this for a query on the demonstration database:

The results look like this:

```
> cat(xml)
<Result>
    <Record>
        <Id>CINMS04C_automatic_UO</Id>
        <DataSource>
            <DeploymentId>CINMS04-C</DeploymentId>
        </DataSource>
        <Start>2008-10-15T00:00:00Z</Start>
        <End>2008-12-04T01:02:30Z</End>
        <Kind>
            <SpeciesId>Odontoceti</SpeciesId>
            <Call>Clicks</Call>
            <Granularity BinSize_m="1.25">binned</Granularity>
        </Kind>
    </Record>
    <Record>
        <Id>CINMS05B_automatic_UO</Id>
        <DataSource>
            <DeploymentId>CINMS05-B</DeploymentId>
        </DataSource>
        <Start>2008-12-04T00:00:00Z</Start>
        <End>2009-02-21T11:14:28.000002Z</End>
        <Kind>
            <SpeciesId>Odontoceti</SpeciesId>
            <Call>Clicks</Call>
            <Granularity BinSize_m="1.25">binned</Granularity>
        </Kind>
    </Record>
    <!-- other records ... -->
</Result>
```

The results from the get query functions are strings. To process them as XML, you will need to use data.tree or one of the XML packages, see the documentation on converting these.

2.1.2 XQuery

Advanced users may wish to use the XQuery facility which requires that queries be formulated using the XQuery language. This provides additional flexibility, but has the overhead of learning the query language. We recommend Walmsley (2006) for an introduction to the language. The Tethys server manual has a short tutorial on XQuery that provides the basics of the query language.

To query Tethys using XQuery, use one of the following two functions:

- t$XQuery(xq, stylesheet)
- t$QueryTethys(xq, stylesheet)

Both functions expect xq to contain an XQuery character string. QueryTethys differs in that it prepends the following lines to your query:

```
declare default element namespace "http://tethys.sdsu.edu/schema/1.0";

import module namespace lib="http://tethys.sdsu.edu/XQueryFns" at "Tethys.xq";
```

These have the effect of placing all of your element names that do not have explicit namespace prefixes (e.g. <xsi:element>) in the Tethys namespace and making Tethys XQuery library functions available. If you are not familiar with namespaces, the QueryTethys method is the query that you probably wish to use. The library functions are described in the Tethys server manual.

3 WORKING WITH XML

R has a variety of mechanisms for working with XML. We recommend either using the XML and xml2 libraries. These libraries may need to be installed the first time that you use them:

```
install.packages("data.tree")

install.packages("XML")

install.packages("xml2")
```

or if you use RStudio you may use the Tools/Install Packages menu option.

Install these libraries and make them active using :

```
library(data.tree)

library(xml2)

library(XML)
```

The xml2 package is the most efficient library. The data.tree offers relatively easy navigation, but it is not efficient for large xml data sets and we are also currently seeing issues with retrieving data from the nodes and are investigating this. For now, we recommend using xml2.

3.1 THE XML/XML2 LIBRARIES

There exist two packages for parsing XML in R. XML is the older package and requires users to manage memory. The xml2 package is a wrapper for a C library and is generally preferred. While either one can be used, we recommend xml2 and will provide xml2 examples.

Read in an XML file. For example, if the output of an effort query was saved as XML file effort.xml, one might read the XML as follows:

```
# Here, we read XML from a file that is in the current
# working directory.  Instead of having a string with
# a filename, we could also have a string returned from the
# R Tethys client and it would also be parsed.
> xml <- read_xml("effort.xml")
```

This creates a set of lists:

```
> xml
```

```
[1] <Record>\n  <Deployment>\n  <Id>CCES10_PTA</Id>\n    <Project>CCES</Project>\n    <Site>PTA</Site>\n ...
 [2] <Record>\n  <Deployment>\n <Id>CCES12_MOB</Id>\n    <Project>CCES</Project>\n    <Site>MOB</Site>\n ...
 [3] <Record>\n  <Deployment>\n <Id>CCES13_CHI</Id>\n    <Project>CCES</Project>\n    <Site>CHI</Site>\n ...
 [4] <Record>\n  <Deployment>\n <Id>CCES14_BCN</Id>\n    <Project>CCES</Project>\n    <Site>BCN</Site>\n ...
 [5] <Record>\n  <Deployment>\n <Id>CCES16_BCN</Id>\n    <Project>CCES</Project>\n    <Site>BCN</Site>\n ...
 [6] <Record>\n  <Deployment>\n <Id>CCES17_BCN</Id>\n    <Project>CCES</Project>\n    <Site>BCN</Site>\n ...
 [7] <Record>\n  <Deployment>\n <Id>CCES18_BCN</Id>\n    <Project>CCES</Project>\n    <Site>BCN</Site>\n ...
 [8] <Record>\n  <Deployment>\n <Id>CCES19_BCN</Id>\n    <Project>CCES</Project>\n    <Site>BCN</Site>\n ...
 [9] <Record>\n  <Deployment>\n <Id>CCES20_BCN</Id>\n    <Project>CCES</Project>\n    <Site>BCN</Site>\n ...
[10] <Record>\n  <Deployment>\n <Id>CCES21_BCN</Id>\n    <Project>CCES</Project>\n    <Site>BCN</Site>\n ...
[11] <Record>\n  <Deployment>\n <Id>CCES22_BCN</Id>\n    <Project>CCES</Project>\n    <Site>BCN</Site>\n ...
[12] <Record>\n  <Deployment>\n <Id>CCES23_BCN</Id>\n    <Project>CCES</Project>\n    <Site>BCN</Site>\n ...
```

XML supports namespaces, a wrapping mechanism that allows us to distinguish the same name when it is defined in different contexts. While Tethys uses namespaces, the RClient removes these by default.

If you overrode the default, they will need to be stripped manually.

```
xml_ns_strip(xml)  # IMPORTANT:  Examples will not function with namespaces
```

To see it as XML, use the XML package:

```
> xml_p <- xmlParse(xml)
> xml_p # display it
```

```
<Result xmlns="http://tethys.sdsu.edu/schema/1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Record>
    <Deployment>
      <Id>CCES10_PTA</Id>
      <Project>CCES</Project>
      <Site>PTA</Site>
      <DeploymentId>10</DeploymentId>
      <DeploymentDetails>
        <Longitude>234.9416</Longitude>
        <Latitude>36.7607</Latitude>
        <TimeStamp>2018-08-22T02:11:00Z</TimeStamp>
        <AudioTimeStamp>2018-08-22T02:24:55Z</AudioTimeStamp>
```

```
        <Vessel>R/V Ruben Lasker</Vessel>
      </DeploymentDetails>
    </Deployment>
    <Detections>
      <Id>CCES010_BW_Detections</Id>
      <Start>2018-08-22T02:24:00Z</Start>
      <End>2018-10-21T23:48:00Z</End>
      <Kind>
        <SpeciesId>Bb1</SpeciesId>
        <Call>Clicks</Call>
        <Granularity>encounter</Granularity>
      </Kind>
      <Kind>
        <SpeciesId>Zc</SpeciesId>
        <Call>Clicks</Call>
        <Granularity>encounter</Granularity>
      </Kind>
      <Kind>
        <SpeciesId Group="BWC">BWC</SpeciesId>
        <Call>Clicks</Call>
        <Granularity>encounter</Granularity>
      </Kind>
      … other species effort omitted for brevity …
      <Algorithm>
        <Method>Analyst detections</Method>
        <Software>Pamguard</Software>
        <Version>2.00.16</Version>
        <Parameters>
          <Click_Viewer_plot_time_m>2</Click_Viewer_plot_time_m>
        </Parameters>
      </Algorithm>
    </Detections>
  </Record>
  ... next Record ...
</Result>
```

Data are queried through XPath expressions, a list of names showing the path through the nesting. For example, if we wanted to know which deployments were in this set of detection effort, we could use:

```
> xml_find_all(xml, "./Record/Deployment/Id")
{xml_nodeset (12)}
 [1] <Id>CCES10_PTA</Id>
 [2] <Id>CCES12_MOB</Id>
 [3] <Id>CCES13_CHI</Id>
 [4] <Id>CCES14_BCN</Id>
 [5] <Id>CCES16_BCN</Id>
 [6] <Id>CCES17_BCN</Id>
```

```
 [7] <Id>CCES18_BCN</Id>
 [8] <Id>CCES19_BCN</Id>
 [9] <Id>CCES20_BCN</Id>
[10] <Id>CCES21_BCN</Id>
[11] <Id>CCES22_BCN</Id>
[12] <Id>CCES23_BCN</Id>
```

where Record/Deployment/Id specifies the path to the deployment identifiers. If we wanted to extract the text from these nodes, we could wrap the call in xml_text:

```
> xml_text(xml_find_all(xml, "./Record/Deployment/Id"))
 [1] "CCES10_PTA" "CCES12_MOB" "CCES13_CHI" "CCES14_BCN" "CCES16_BCN" "CCES17_BC
N" "CCES18_BCN" "CCES19_BCN"
 [9] "CCES20_BCN" "CCES21_BCN" "CCES22_BCN" "CCES23_BCN"
```

Other useful functions are xml_integer() and xml_double for extracting numeric values. Additional information may be found in the documentation for the xml2 library as well as descriptions of XPath. There are numerous resources for learning XPath, Walmsley (2006) covers XPath well in her introduction to the XQuery language. Many examples on the web tend to use the wildcard search operator (//) which let one search for an XML element at any level (e.g., .//Id instead of ./Record/Deployment/Id). These types of searches are inefficient and should be avoided except on very small data.

3.2 The data.tree Library

The data.tree library is a reference class that can interpret hierarchical data and is well suited for working with XML. It contains a function for transforming XML to a data.tree object.

Assuming that the Tethys class is contained in variable tethys, one could run a query and convert it to a data.tree as follows:

```
# Run a query to find out for what, where, and when analysts have
# been looking for.  Assumes that analysts always use the term
# "Analyst detections" for the Detections/Algorithm/Method value.

> xml <- tethys$getDetectionEffort('Algorithm/Method = "Analyst detections"')
> tree <- tethys$xml2tree(xml)  # Convert to a data tree
```

Data trees can also be loaded in the usual R manner. When the web client is used, the results of queries can be saved as a data.tree object and loaded into R with the load command, e.g.:

```
# deployment query result
load("C:/Users/MyUserName/Downloads/deployments.rData")
```

This will result in the variable tree appearing in the R workspace. Let's take a look at the result of an effort query that we made on the demonstration database:

```
> tree
                                                levelName
1  Root
2   °--Result
3       ¦--1
4       ¦   °--Deployment
5       ¦       ¦--Id
6       ¦       ¦--Project
7       ¦       ¦--DeploymentId
8       ¦       ¦--Site
9       ¦       ¦--Channel
10      ¦       ¦   ¦--ChannelNumber
11      ¦       ¦   ¦--SensorNumber
12      ¦       ¦   ¦--Start
13      ¦       ¦   ¦--End
14      ¦       ¦   °--Sampling
15      ¦       ¦       °--Regimen
16      ¦       ¦           ¦--TimeStamp
17      ¦       ¦           ¦--SampleRate_kHz
18      ¦       ¦           °--SampleBits
19      ¦       °--DeploymentDetails
20      ¦           ¦--Longitude
21      ¦           ¦--Latitude
22      ¦           ¦--ElevationInstrument_m
23      ¦           ¦--DepthInstrument_m
24      ¦           ¦--TimeStamp
25      ¦           ¦--AudioTimeStamp
26      ¦           °--Vessel
27      ¦--2
28      ¦   °--Deployment
29      ¦       ¦--Id
30      ¦       ¦--Project
31      ¦       ¦--DeploymentId
32      ¦       ¦--Site
33      ¦       ¦--Channel
34      ¦       ¦   ¦--ChannelNumber
35      ¦       ¦   ¦--SensorNumber
36      ¦       ¦   ¦--Start
37      ¦       ¦   ¦--End
38      ¦       ¦   °--Sampling
39      ¦       ¦       °--Regimen
40      ¦       ¦           ¦--TimeStamp
41      ¦       ¦           ¦--SampleRate_kHz
42      ¦       ¦           °--SampleBits
43      ¦       °--DeploymentDetails
44      ¦           ¦--Longitude
45      ¦           ¦--Latitude
46      ¦           ¦--ElevationInstrument_m
47      ¦           ¦--DepthInstrument_m
48      ¦           ¦--TimeStamp
49      ¦           ¦--AudioTimeStamp
50      ¦           °--Vessel
51      °--3
52          °--Deployment
53              ¦--Id
…
72              °--DeploymentDetails
73                  ¦--Longitude
74                  ¦--Latitude
75                  ¦--ElevationInstrument_m
```

```
76                    ¦--DepthInstrument_m
77                    ¦--TimeStamp
78                    ¦--AudioTimeStamp
79                    °--Vessel
```

The tree variable is an instance of a reference class, and as such uses $ signs to access its methods. One can access nodes by using the children method, or when a name is unique, just typing the name. For instance, to access the second deployment node, we would use

```
> tree$Result$children[[2]]
```

The children method can be abbreviated with single backticks (`):

```
> tree$Result$`2`
```

To navigate to the Start element of the second deployment, we would use:

```
> tree$Result$`2`$Channel$Start
```

Further documentation is available on the CRAN repository.

References

Walmsley, P. (2006). XQuery. Farnham, UK, O'Reilly.