

Package ‘streamConnect’

October 23, 2026

Version 0.1.0

Date 2026-10-22

Encoding UTF-8

Title Connecting Stream Mining Components Using Sockets and Web Services

Description Adds functionality to connect stream mining components from the stream package using sockets and Web services. The package can be used to create distributed work-flows and plumber-based Web services that can be deployed on most common cloud services.

Depends stream (>= 2.0-0)

Imports plumber, callr, httr, readr, stringr, jsonlite

Suggests knitr, httpuv, processx, rmarkdown, testthat (>= 3.0.0), covr

BugReports <https://github.com/mhahsler/streamConnect/issues>

License GPL-3

VignetteBuilder knitr

Config/testthat/edition 3

URL <https://michael.hahsler.net/streamConnect/>

Config/roxygen2/version 8.1.1

NeedsCompilation no

Author Michael Hahsler [aut, cre, cph] (ORCID:
<<https://orcid.org/0000-0003-2716-1405>>)

Maintainer Michael Hahsler <mhahsler@lyle.smu.edu>

Repository CRAN

Date/Publication 2026-10-23 09:10:03 UTC

Contents

DSC_WebService	2
DSD_ReadSocket	3
DSD_ReadWebService	4

publish_DSC_via_WebService	5
publish_DSD_via_Socket	7
publish_DSD_via_WebService	9
retry	11

Index	12
--------------	-----------

DSC_WebService	<i>A DSC Interface for a DSC Running as a Web Service</i>
----------------	---

Description

Provides a DSC front-end for a clusterer running as a web service. The methods `nclusters()`, `get_center()`, `get_weights()` are supported. The request is retried with `httr::RETRY()` if it fails the first time.

Usage

```
DSC_WebService(url, verbose = FALSE, ...)
```

Arguments

<code>url</code>	endpoint URI address in the format <code>http://host:port/<optional_path></code> .
<code>verbose</code>	logical; display connection information.
<code>...</code>	further arguments are passed on to <code>httr::RETRY()</code> . Pass <code>httr::verbose()</code> as parameter config to get detailed connection info.

Value

A `stream::DSC` object.

See Also

Other WebService: `DSD_ReadWebService()`, `publish_DSC_via_WebService()`, `publish_DSD_via_WebService()`
 Other dsc: `publish_DSC_via_WebService()`

Examples

```
# find a free port
port <- httpuv::randomPort()
port

# deploy a clustering process listening for data on the port
rp1 <- publish_DSC_via_WebService("DSC_DBSTREAM(r = .05)", port = port)
rp1

# get a local DSC interface
dsc <- DSC_WebService(paste0("http://localhost", ":", port),
  verbose = TRUE, config = httr::verbose(info = TRUE))
```

```
dsc

# cluster
dsd <- DSD_Gaussians(k = 3, d = 2, noise = 0.05)

update(dsc, dsd, 500)

get_centers(dsc)
get_weights(dsc)

plot(dsc)

# kill the background clustering process.
rp1$kill()
rp1
```

DSD_ReadSocket*A DSD That Reads from a Server Port*

Description

Creates a `DSD_ReadStream` that reads from a port.

Usage

```
DSD_ReadSocket(host = "localhost", port, retry_args = NULL, ...)
```

Arguments

<code>host</code>	hostname.
<code>port</code>	host port.
<code>retry_args</code>	a list with arguments for <code>retry()</code> .
<code>...</code>	further arguments are passed on to <code>stream::DSD_ReadStream()</code> .

Value

A `stream::DSD` object.

See Also

Other Socket: [publish_DSD_via_Socket\(\)](#)

Other dsd: [DSD_ReadWebService\(\)](#), [publish_DSD_via_Socket\(\)](#), [publish_DSD_via_WebService\(\)](#)

Examples

```
# find a free port
port <- httpuv::randomPort()
port

# create a background DSD process sending data to the port
rp1 <- DSD_Gaussians(k = 3, d = 3) %>% publish_DSD_via_Socket(port = port)
rp1

# create a DSD that connects to the socket. Note that we need to
# specify the column names of the stream
dsd <- DSD_ReadSocket(port = port, col.names = c("x", "y", "z", ".class"))
dsd

get_points(dsd, n = 10)

plot(dsd)

close_stream(dsd)

# end the DSD process. Note: that closing the connection above
# may already kill the process.
if (rp1$is_alive()) rp1$kill()
rp1
```

DSD_ReadWebService	<i>A DSD That Reads from a Web Service</i>
--------------------	--

Description

Reads from a web service that publishes an operation called `get_points` which takes a parameter `n` and returns `n` data points in CSV or json format. The request is retried with `httr::RETRY()` if it fails the first time.

Usage

```
DSD_ReadWebService(url, verbose = FALSE, ...)
```

Arguments

<code>url</code>	endpoint URI address in the format <code>http://host:port/<optional_path></code> .
<code>verbose</code>	logical; display connection information.
<code>...</code>	further arguments are passed on to <code>httr::RETRY()</code> . Pass <code>httr::verbose()</code> as parameter config to get detailed connection info.

Value

A `stream::DSD` object.

See Also

Other WebService: [DSC_WebService\(\)](#), [publish_DSC_via_WebService\(\)](#), [publish_DSD_via_WebService\(\)](#)

Other dsd: [DSD_ReadSocket\(\)](#), [publish_DSD_via_Socket\(\)](#), [publish_DSD_via_WebService\(\)](#)

Examples

```
# find a free port
port <- httpuv::randomPort()
port

# create a background DSD process sending data to the port
rp1 <- publish_DSD_via_WebService("DSD_Gaussians(k = 3, d = 3)", port = port)

## use json for the transport layer instead of csv
# rp1 <- publish_DSD_via_WebService("DSD_Gaussians(k = 3, d = 3)",
#                                   port = port, serialize = "json")
rp1

# create a DSD that connects to the web service
dsd <- DSD_ReadWebService(paste0("http://localhost", ":", port))
dsd

get_points(dsd, n = 10)

plot(dsd)

# end the DSD process. Note: that closing the connection above
# may already kill the process.
rp1$kill()
rp1
```

publish_DSC_via_WebService

Publish a Data Stream Clustering Task via a Web Service

Description

Uses the package plumber to publish a data stream task as a web service.

Usage

```
publish_DSC_via_WebService(
  dsc,
  port,
  task_file = NULL,
  serializer = "csv",
  serve = TRUE,
  background = TRUE,
  debug = FALSE
)
```

Arguments

dsc	A character string that creates a DSC.
port	port used to serve the task.
task_file	name of the plumber task script file.
serializer	method used to serialize the data. By default, csv (comma-separated values) is used. Other methods are json and rds (see plumber::serializer_csv).
serve	if TRUE, then a task file is written and a server started, otherwise, only a plumber task file is written.
background	logical; start a background process?
debug	if TRUE, then the service is started locally and a web client is started to explore the interface.

Details

The function writes a plumber task script file and starts the web server to serve the content of the stream using the endpoints

- GET /info
- POST /update requires the data to be uploaded as a file in csv format (see Examples section).
- GET /get_centers with parameter type (see [stream::get_centers\(\)](#)).
- GET /get_weights with parameter type (see [stream::get_weights\(\)](#)).

Supported serializers are csv (default), json, and rds.

APIs generated using plumber can be easily deployed. See: [Hosting](#). By setting a task_file and serve = FALSE a plumber task script file is generated for deployment.

Value

a [processx::process](#) object created with [callr::r_bg\(\)](#) which runs the plumber server in the background. The process can be stopped with [rp\\$kill\(\)](#) or by killing the process using the operating system with the appropriate PID. [rp\\$get_result\(\)](#) can be used to check for errors in the server process (e.g., when it terminates unexpectedly).

See Also

Other WebService: [DSC_WebService\(\)](#), [DSD_ReadWebService\(\)](#), [publish_DSD_via_WebService\(\)](#)

Other dsc: [DSC_WebService\(\)](#)

Examples

```
# find a free port
port <- httpuv::randomPort()
port

# Deploy a clustering process listening for data on the port
rp1 <- publish_DSC_via_WebService("DSC_DBSTREAM(r = .05)", port = port)
rp1
```

```

# look at ? DSC_WebService for a convenient interface.
# Here we show how to connect to the port and send data manually.
library(httr)

# the info verb returns some basic information about the clusterer.
resp <- RETRY("GET", paste0("http://localhost:", port, "/info"))
d <- content(resp, show_col_types = FALSE)
d

# create a local data stream and send it to the clusterer using the update verb.
dsd <- DSD_Gaussians(k = 3, d = 2, noise = 0.05)

tmp <- tempfile()
stream::write_stream(dsd, tmp, n = 500, header = TRUE)
resp <- POST(paste0("http://localhost:", port, "/update"),
  body = list(upload = upload_file(tmp)))
unlink(tmp)
resp

# retrieve the cluster centers using the get_centers verb
resp <- GET(paste0("http://localhost:", port, "/get_centers"))
d <- content(resp, show_col_types = FALSE)
head(d)

plot(dsd, n = 100)
points(d, col = "red", pch = 3, lwd = 3)

# kill the process.
rp1$kill()
rp1

# Debug the interface (run the service and start a web interface)
if (interactive())
  publish_DSC_via_WebService("DSC_DBSTREAM(r = .05)",
    port = port, debug = TRUE)

```

publish_DSD_via_Socket

Publish a Data Stream using a Socket

Description

Creates a socket server connection to send stream data.

Usage

```
publish_DSD_via_Socket(dsd, port, blocksize = 1024L, background = TRUE, ...)
```

Arguments

<code>dsd</code>	A DSD object.
<code>port</code>	port used to serve the DSD.
<code>blocksize</code>	number of data points pushed on the buffer at once.
<code>background</code>	logical; start a background process?
<code>...</code>	further arguments are passed on to <code>socketConnection()</code> .

Details

Creates a server socket with `socketConnection()` and then uses a `stream::write_stream()` to write data to a socket connection. This method does not provide a header for the data.

Value

a `processx::process` object created with `callr::r_bg()` which runs the plumber server in the background. The process can be stopped with `rp$kill()` or by killing the process using the operating system with the appropriate PID. `rp$get_result()` can be used to check for errors in the server process (e.g., when it terminates unexpectedly).

See Also

Other Socket: `DSD_ReadSocket()`

Other dsd: `DSD_ReadSocket()`, `DSD_ReadWebService()`, `publish_DSD_via_WebService()`

Examples

```
# find a free port
port <- httpuv::randomPort()
port

# create a background DSD process sending data to the port
rp1 <- DSD_Gaussians(k = 3, d = 3) %>% publish_DSD_via_Socket(port = port)
rp1

# connect to the port (retry waits for the socket to establish)
con <- retry(socketConnection(port = port, open = 'r'))
dsd <- retry(DSD_ReadStream(con, col.names = c("x", "y", "z", ".class")))

get_points(dsd, n = 10)

plot(dsd)

# close connection
close_stream(dsd)

# end the DSD process. Note: that closing the connection above
# may already kill the process.
rp1$kill()
rp1
```

`publish_DSD_via_WebService`*Publish a Data Stream via a Web Service*

Description

Uses the package plumber to publish a data stream as a web service.

Usage

```
publish_DSD_via_WebService(  
  dsd,  
  port,  
  task_file = NULL,  
  serializer = "csv",  
  serve = TRUE,  
  background = TRUE,  
  debug = FALSE  
)
```

Arguments

<code>dsd</code>	A character string that creates a DSD.
<code>port</code>	port used to serve the DSD.
<code>task_file</code>	name of the plumber task script file.
<code>serializer</code>	method used to serialize the data. By default, csv (comma-separated values) is used. Other methods are json and rds (see plumber::serializer_csv).
<code>serve</code>	if TRUE, then a task file is written and a server started, otherwise, only a plumber task file is written.
<code>background</code>	logical; start a background process?
<code>debug</code>	if TRUE, then the service is started locally and a web client is started to explore the interface.

Details

The function writes a plumber task script file and starts the web server to serve the content of the stream using the endpoints

- `http://localhost:port/get_points?n=100` and
- `http://localhost:port/info`.

APIs generated using plumber can be easily deployed. See: [Hosting](#). By setting a `task_file` and `serve = FALSE` a plumber task script file is generated for deployment.

A convenient reader for stream data over web services is available as [DSD_ReadWebService](#).

Value

a `processx::process` object created with `callr::r_bg()` which runs the plumber server in the background. The process can be stopped with `rp$kill()` or by killing the process using the operating system with the appropriate PID. `rp$get_result()` can be used to check for errors in the server process (e.g., when it terminates unexpectedly).

See Also

Other WebService: `DSC_WebService()`, `DSD_ReadWebService()`, `publish_DSC_via_WebService()`

Other dsd: `DSD_ReadSocket()`, `DSD_ReadWebService()`, `publish_DSD_via_Socket()`

Examples

```
# find a free port
port <- httpuv::randomPort()
port

# create a background DSD process sending data to the port
rp1 <- publish_DSD_via_WebService("DSD_Gaussians(k = 3, d = 3)", port = port)
rp1

# connect to the port and read manually. See DSD_ReadWebService for
# a more convenient way to connect to the WebService in R.
library("httr")

# we use RETRY to give the server time to spin up
resp <- RETRY("GET", paste0("http://localhost:", port, "/info"))
d <- content(resp, show_col_types = FALSE)
d

# example: Get 100 points and plot them
resp <- GET(paste0("http://localhost:", port, "/get_points?n=100"))
d <- content(resp, show_col_types = FALSE)
head(d)

dsd <- DSD_Memory(d)
dsd
plot(dsd, n = -1)

# end the DSD process. Note: that closing the connection above
# may already kill the process.
rp1$kill()
rp1

# Publish using json

rp2 <- publish_DSD_via_WebService("DSD_Gaussians(k = 3, d = 3)",
  port = port, serializer = "json")
rp2

# connect to the port and read
```

```
# we use RETRY to give the server time to spin up
resp <- RETRY("GET", paste0("http://localhost:", port, "/info"))
content(resp, as = "text")

resp <- GET(paste0("http://localhost:", port, "/get_points?n=5"))
content(resp, as = "text")

# cleanup
rp2$kill()
rp2

# Debug the interface (run the service and start a web interface)
if (interactive())
  publish_DSD_via_WebService("DSD_Gaussians(k = 3, d = 3)", port = port,
    debug = TRUE)
```

 retry

Retry an Expression that Fails

Description

Retries an expression that fails. This is mainly used when establishing a connection.

Usage

```
retry(f, times = 5, wait = 1, verbose = FALSE, operation = NULL)
```

Arguments

f	expression
times	integer; number of times
wait	number of seconds to wait in between tries.
verbose	logical; show progress and errors.
operation	name of the operation used in the error message.

Value

the result of the expression f

Examples

```
retry(1)
```

Index

- * **Socket**
 - DSD_ReadSocket, [3](#)
 - publish_DSD_via_Socket, [7](#)
- * **WebService**
 - DSC_WebService, [2](#)
 - DSD_ReadWebService, [4](#)
 - publish_DSC_via_WebService, [5](#)
 - publish_DSD_via_WebService, [9](#)
- * **dsc**
 - DSC_WebService, [2](#)
 - publish_DSC_via_WebService, [5](#)
- * **dsd**
 - DSD_ReadSocket, [3](#)
 - DSD_ReadWebService, [4](#)
 - publish_DSD_via_Socket, [7](#)
 - publish_DSD_via_WebService, [9](#)

callr::r_bg(), [6](#), [8](#), [10](#)

DSC_WebService, [2](#)

DSC_WebService(), [5](#), [6](#), [10](#)

DSD_ReadSocket, [3](#)

DSD_ReadSocket(), [5](#), [8](#), [10](#)

DSD_ReadWebService, [4](#), [9](#)

DSD_ReadWebService(), [2](#), [3](#), [6](#), [8](#), [10](#)

httr::RETRY(), [2](#), [4](#)

httr::verbose(), [2](#), [4](#)

plumber::serializer_csv, [6](#), [9](#)

processx::process, [6](#), [8](#), [10](#)

publish_DSC_via_WebService, [5](#)

publish_DSC_via_WebService(), [2](#), [5](#), [10](#)

publish_DSD_via_Socket, [7](#)

publish_DSD_via_Socket(), [3](#), [5](#), [10](#)

publish_DSD_via_WebService, [9](#)

publish_DSD_via_WebService(), [2](#), [3](#), [5](#), [6](#), [8](#)

retry, [11](#)

retry(), [3](#)

socketConnection(), [8](#)

stream::DSC, [2](#)

stream::DSD, [3](#), [4](#)

stream::DSD_ReadStream(), [3](#)

stream::get_centers(), [6](#)

stream::get_weights(), [6](#)

stream::write_stream(), [8](#)