LineSender

You could use the LineSender program to provide sample input from a file to Odysseus.

The source code can be found here: https://git.swl.informatik.uni-oldenburg.de/projects/ODYDATGEN/repos/linesender/browse

A runnable jar can be downloaded here: https://odysseus.informatik.uni-oldenburg.de/download/linesender/linesender.jar

Use

```
java -jar linesender.jar
```

to see a list of parameters:

```
Usage: LineSender [-hrsV] [-d=<delay>] [-p=<port>] <path>
Sends a csv file to a tcp socket connection
      <path>            The file whose checksum to calculate.
  -d, --delay=<delay>   Delay ms between each line sent.
  -h, --help            Show this help message and exit.
  -p, --port=<port>     The server port.
  -r, --repeat          Repeat the file when ended. Default is false.
  -s, --skip            Skip the first line in csv, e.g. because of table
                          header. Default is false.
  -V, --version         Print version information and exit.
```

This program reads a CSV file and sends each CSV file via TCP to a connected TCP client. For each new connected client, the reading starts from the beginning of the file. If you set the repeat flag, the content of the CSV file is sent again and again to simulate a never ending input stream. You could limit the data rate with the delay flag.

Example

If you have the following CSV file:

```
a;1;1.0
b;2;2.0
c;3;3.0
d;4;4.0
e;5;5.0
```

and you have started the sender on port 9876. You could use the following Odysseus query to access the content:

```
#PARSER PQL
#ADDQUERY
in = ACCESS({
          transport = 'tcpclient',
          source = 'source_test',
          datahandler = 'tuple',
          wrapper = 'GenericPush',
          protocol = 'SimpleCSV',
          options = [['host','localhost'],['port','9876'],['delimiter',';']],
          schema=[
            ['A', 'String'],
            ['B', 'Integer'],
            ['C', 'Double']
          ]
        }
      )
```

Remark: If you use repeat, you need to assure at the client side, that the timestamps are increasing. In the above example this is simply done by using system time but when using application time (i.e. providing timestamp datatypes) you will need to increase the timestamps manually,

```
in_prep = STATEMAP({
              keepinput = true,
              expressions = [
                ['!isNull(__last_1.B) && __last_1.B > B','nextRound']
              ]
            },
            in
          )

in_ts = TIMESTAMP({
            start = 'B+condcounter(nextRound)*10'
          },
          in_prep
        )
```

Here, we first check, if the new start timestamp is lower than the old one. This means, the input file has started from the beginning and nextRound is set to true.

Then the TIMESTAMP operator creates timestamps from the B attribute values and increases the concounter each time, nextRound is true. Here we use 10 as a maximum value.