IBM InfoSphere Streams 3.2: SPL Programming

Lab Guide 1: Standard Operators and SPL Expression Language

November 2013

Document Version: 3.0
Update Date: November 11, 2013
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE OF CONTENTS</td>
<td>2</td>
</tr>
<tr>
<td>INTRODUCTION – SPL PROGRAMMING</td>
<td>1</td>
</tr>
<tr>
<td>1.1 GOAL AND OBJECTIVES</td>
<td>1</td>
</tr>
<tr>
<td>1.2 PREREQUISITES</td>
<td>1</td>
</tr>
<tr>
<td>1.3 OVERVIEW</td>
<td>2</td>
</tr>
<tr>
<td>1.4 LAB ENVIRONMENT</td>
<td>3</td>
</tr>
<tr>
<td>LAB 1 HELLO WORLD</td>
<td>4</td>
</tr>
<tr>
<td>1.1 GETTING STARTED</td>
<td>4</td>
</tr>
<tr>
<td>1.2 CREATING, COMPILING, AND RUNNING THE HELLO WORLD SPL APPLICATION</td>
<td>5</td>
</tr>
<tr>
<td>1.3 RUN INFOSPHERE STREAMS STUDIO USING A NEW WORKSPACE</td>
<td>8</td>
</tr>
<tr>
<td>1.4 IMPORT, BUILD, AND RUN THE HELLO WORLD APPLICATION</td>
<td>9</td>
</tr>
<tr>
<td>1.5 GENERATE A MAKEFILE FOR THE COMMAND LINE VERSION OF HELLO WORLD</td>
<td>10</td>
</tr>
<tr>
<td>LAB 2 EXPRESSION LANGUAGE, FUNCTIONS, TYPES, AND TOOLKITS</td>
<td>13</td>
</tr>
<tr>
<td>2.1 CREATE THE STREAMSTK.IP TOOLKIT PROJECT AND NAMESPACE</td>
<td>13</td>
</tr>
<tr>
<td>2.2 CREATE THE FLOWREC TUPLE TYPE</td>
<td>14</td>
</tr>
<tr>
<td>2.3 CREATE A FUNCTION TO GENERATE A RANDOM INTEGER</td>
<td>16</td>
</tr>
<tr>
<td>2.4 CREATE A RANDOM INTEGER BETWEEN SPECIFIED LIMITS</td>
<td>17</td>
</tr>
<tr>
<td>2.5 WRITE A FUNCTION TO GENERATE AN IP ADDRESS AS AN INTEGER</td>
<td>17</td>
</tr>
<tr>
<td>2.6 VIEW THE TOOLKIT INFORMATION</td>
<td>18</td>
</tr>
<tr>
<td>2.7 CREATE AN APPLICATION TO TEST THE TOOLKIT</td>
<td>19</td>
</tr>
<tr>
<td>2.8 LAB 2 SOLUTION</td>
<td>22</td>
</tr>
<tr>
<td>LAB 3 SPL ADAPTERS: TCPSOURCE AND TCPSINK</td>
<td>24</td>
</tr>
<tr>
<td>3.1 DUPLICATE THE NETBEACON PROJECT</td>
<td>24</td>
</tr>
<tr>
<td>3.2 REFACTOR THE NETGEN PROJECT</td>
<td>25</td>
</tr>
<tr>
<td>3.3 TURN THE TEST APPLICATION INTO A SIMULATOR</td>
<td>25</td>
</tr>
<tr>
<td>3.4 LAUNCH AND TEST THE NETGEN STANDALONE APPLICATION</td>
<td>26</td>
</tr>
<tr>
<td>3.5 CREATE THE NETRECEIVE PROJECT BY DUPLICATING THE NETBEACON PROJECT</td>
<td>27</td>
</tr>
<tr>
<td>3.6 REPLACE THE BEACON OPERATOR WITH A TCPSOURCE OPERATOR</td>
<td>28</td>
</tr>
<tr>
<td>3.7 TEST THE NETRECEIVE PROGRAM IN STANDALONE MODE</td>
<td>28</td>
</tr>
<tr>
<td>3.8 CREATE A FUNCTION TO FORMAT IP ADDRESSES (.XXX.XXX.XXX.XXX)</td>
<td>29</td>
</tr>
<tr>
<td>3.9 ENHANCE NETRECEIVE TO PRINT HUMAN-READABLE IP ADDRESSES</td>
<td>30</td>
</tr>
<tr>
<td>3.10 LAB 3 FINAL SOLUTION</td>
<td>31</td>
</tr>
<tr>
<td>LAB 4 SPL AGGREGATION AND SORTING</td>
<td>33</td>
</tr>
<tr>
<td>4.1 CREATE THE NETSUMMARY PROJECT BY DUPLICATING THE NETRECEIVE PROJECT</td>
<td>33</td>
</tr>
<tr>
<td>4.2 AGGREGATE THE STREAMING DATA IN MEMORY AND PRODUCE A FLOW OF SOURCE IPS WITH STATISTICS OVER A PERIOD OF TIME</td>
<td>34</td>
</tr>
<tr>
<td>4.3 SORT THE SUMMARY RECORDS BY IP ADDRESS</td>
<td>35</td>
</tr>
<tr>
<td>4.4 FIND THE TOP 5 SOURCE IP ADDRESSES BY BYTES PRODUCED</td>
<td>36</td>
</tr>
<tr>
<td>4.5 COMPARE AND LAUNCH THE APPLICATION IN DISTRIBUTED MODE</td>
<td>38</td>
</tr>
<tr>
<td>4.6 REMOVE THE “SPEED LIMIT” FROM THE NETGEN PROGRAM</td>
<td>39</td>
</tr>
<tr>
<td>4.7 VIEW THE CONSOLE LOG</td>
<td>40</td>
</tr>
<tr>
<td>4.8 SAMPLE THE TUPLE DATA</td>
<td>41</td>
</tr>
<tr>
<td>4.9 LAB 4 FINAL SOLUTION</td>
<td>42</td>
</tr>
<tr>
<td>4.10 LAB 4 DISCUSSION TOPIC</td>
<td>44</td>
</tr>
<tr>
<td>END OF SPL LAB</td>
<td>44</td>
</tr>
</tbody>
</table>
Introduction – SPL Programming

The Streams Processing Language (SPL) is the Streams applications composition and flow language. Key features of the language include:

- **Stream-centric, operator-based language** – Stream processing declarative language supporting incremental development and composition, dynamic connectivity, and rich expression language.

- **Growing base of toolkits and operators** – Extensible through the addition of toolkits and operators, able to support arbitrary data streams at high volume. Toolkits include: Internet, Database connectivity, Text Analytics, HDFS Adapters, OpenCV graphics, and many more.

- **User-defined operators** – API included to extend language and toolkits. Supports reusing existing analytic code written in C++ or Java. Write new operators and toolkits to encapsulate domain-specific logic, data types, and adapters.

1.1 Goal and Objectives

The goal of this SPL Programming lab is to provide a solid foundation of SPL, including:

- compiler usage
- the expression language
- standard operators
- data types
- stream processing concepts such as windows and punctuation
- toolkit development

The objectives of this lab (along with the accompanying presentations) are to:

- Use a pseudo-real world problem space to explore the SPL language and stream computing
- Utilize best practices and an iterative development approach to developing SPL applications
- Demonstrate the power of the SPL expression language, which can reduce the need to drop into C/C++ or Java to develop native functions

1.2 Prerequisites

This lab was designed to follow the Streams 3 Introductory Hands-On Lab, which demonstrates many of the features of the Streams Studio IDE. This lab assumes that students have a basic understanding of the IDE and are able to navigate it.
1.3 Overview

The purpose of this hands-on lab is to provide a practical approach to learning SPL through a series of hands-on exercises, which build upon one another.

![Diagram of SPL Programming Lab overview](image)

**Figure 1 – SPL Programming Lab overview**

The lab is based on a fairly simple network flow data (NetFlow) example.

Figure 1, above, provides a graphical overview of the SPL artifacts that you will develop through this lab. They include:

- An SPL Toolkit: streamstk.ip
- A NetFlow-like data generator that can run as a standalone Linux executable
- A NetFlow Summary SPL Application, streaming data from a socket

The lab is broken into 4 parts:

**Lab 1** – Hello World: a review of SPL program creation, compilation and execution in both the GUI and command line environments.

**Lab 2** – SPL Expression Language, Functions, Types and Toolkits: Build an SPL toolkit of data types and SPL functions to facilitate development of more complex applications and promote reusability.
Lab 3 – SPL Adapters: TCPSource and TCPSink. Use the SPL language to develop a standalone data generator and a Streams application to receive streaming data over a socket connection.

Lab 4 – Aggregation and Sort: Use the standard operators to develop a NetFlow summarization analytic and see how fast it can run on the lab hardware.

1.4 Lab Environment

This lab does not require a specific VMware image. The labs in this guide can be accomplished on any Streams environment meeting the following minimum requirements:

- IBM InfoSphere Streams 3.x
- The instructions in this lab are based on version 3.0.0.0. There may be minor changes to the Streams Studio steps in 3.1 and later versions.
- Red Hat Enterprise Linux (RHEL) 5.3 or later, CentOS Linux 6.1 or later, SuSE Linux Enterprise Server (SLES) 11.2 or later
- Streams Studio or Eclipse 3.6.2 with InfoSphere Streams Studio plug-ins installed

![Note]

There are a few steps in the lab that refer to specific usernames and the host name of the lab image. Please modify these as appropriate for your environment.

If this lab is being taught using the same VMware Image that was used for the InfoSphere Streams 3 Introductory Hands-On Lab, then the following usernames and passwords are included:

| Table 1. Streams 3.x lab – virtual machine information |
|-----------------|-------------------|
| **Parameter**   | **Value**         |
| Host name       | streamslab or streamtrial or bigdata (it does not matter) |
| User and administrator ID | streamsadmin |
| User password   | passw0rd (password with a zero for the O) |
| root password   | passw0rd |
Lab 1  Hello World

This lab will introduce you to SPL compilation and execution from the command line (outside the Streams Studio IDE). The use of standalone SPL applications will be introduced, providing a very fast and effective method for developing and testing SPL application functionality without requiring a Streams instance.

One could say that when it comes to programming, everything starts with "Hello World". This is even true for SPL programming. This simple program is a great way to verify Streams installations, verify developer environments, learn individual operators and functions, and unit-test your own SPL functions, custom operators, composite operators, native functions, and primitive operators.

1.1  Getting started

You will be using the streamsadmin account for this lab. After login you will be presented with a desktop containing several icons. Open up a terminal window and adjust position and size as you see fit.

1. Login to the VMware image

   User: Streams Administrator (streamsadmin)
   Password: passw0rd (password with a zero for the O)
1. Right-click on the desktop and choose **Open in Terminal**, or use the **Terminal launcher** in the top panel.

2. Creating, compiling, and running the HelloWorld.spl application

Create and change to a directory named **spllab**.

```
$ mkdir spllab
$ cd spllab
```

Create and change to a directory named **HelloWorld**.

```
$ mkdir HelloWorld
$ cd HelloWorld
```

3. Use **vi**, **emacs**, or **gedit** to create a new file named **HelloWorld.spl**.

   **Note**

   **vi** is a console-based visual editor; if you don’t already know how to use it, you probably want to stick with **emacs** or **gedit**. **emacs** is the GNU Emacs editor, and like **vi**, if you have not used it before, you should probably use **gedit**. **gedit** is a more normal text editing program, using menus and control keystrokes to cut/copy/paste. **vi** and **emacs** support SPL syntax highlighting.

   This lab assumes **gedit**. By appending ampersand (“&”) to the command line, you start the editor in its own window and keep the current terminal window’s command line available for further (Streams) commands.

```
$ gedit HelloWorld.spl &
```
The code for HelloWorld.spl is shown below. (The syntax highlighting is actually from vi with the default color scheme—just to illustrate.)

```plaintext
composite HelloWorld
{
    graph
        stream<rstring message> Hi = Beacon()
        {
            param iterations: 1u;
            output Hi: message = "Hello, world!";
        }
    
        () as Sink = Custom(Hi)
        {
            logic onTuple Hi: printStringLn(message);
        }
}
```

Save your source file, ensuring it is named HelloWorld.spl.

The SPL compiler supports two types of application builds: **Standalone** and **Distributed**. Standalone creates a Linux executable program that can run as a single process on a single machine, without requiring a running InfoSphere Streams instance. Use the `-T` flag to specify the Standalone target.

4. **Compile** the HelloWorld.spl application in standalone mode.

```bash
$ sc -T -M HelloWorld
```

Creating the types.
Creating the functions.
Creating the operators.
Creating the processing elements.
The standalone application is being created.
Creating the application model.
Building the binaries.
[CXX-type] tuple<rstring message>
[CXX-operator] Hi
[CXX-operator] Sink
[CXX-pe] pe HelloWorld-a
[LD-pe] pe HelloWorld-a
[CXX-standalone] standalone
[LD-standalone] standalone
[LN-standalone] standalone

If you do not receive this output, or an error is reported, check your code for errors and recompile.

5. **Display the tree view** of directories and files created by a successful Streams compile.

```bash
$ tree
```

4. **Compile** the HelloWorld.spl application in standalone mode.

```bash
$ sc -T -M HelloWorld
```

Creating the types.
Creating the functions.
Creating the operators.
Creating the processing elements.
The standalone application is being created.
Creating the application model.
Building the binaries.
[CXX-type] tuple<rstring message>
[CXX-operator] Hi
[CXX-operator] Sink
[CXX-pe] pe HelloWorld-a
[LD-pe] pe HelloWorld-a
[CXX-standalone] standalone
[LD-standalone] standalone
[LN-standalone] standalone

If you do not receive this output, or an error is reported, check your code for errors and recompile.

5. **Display the tree view** of directories and files created by a successful Streams compile.

```bash
$ tree
```
data
└── HelloWorld.spl

output
    └── bin
        ├── HelloWorld
        │    └── standalone
        │    └── HelloWorld-a.dpe
        │    └── standalone
        │        └── HelloWorld-a.dpe
        │        └── operator
        │            └── Hi.dep
        │            └── Sink.dep
        │        └── ldOptions
        │            └── Makefile
        │                └── operator
        │                    └── Hi.d
        │                    └── Hi.o
        │                    └── Sink.d
        │                    └── Sink.o
        │        └── pe
        │            └── BeJwz8siMNwnOzMuOBwARYQNr.o
        │        └── standalone
        │            └── standalone.o
        │        └── type
        │            └── BeJwrMSw2z@0tLk5MTwJAGG4EA1.o
        └── etc
            └── adl  >  ../HelloWorld.adl
                └── HelloWorld.adl

src
    └── operator
        ├── Hi.cpp
        ├── Hi.h
        ├── Hi.xml
        ├── Sink.cpp
        ├── Sink.h
        └── Sink.xml
        └── pe
            └── BeJwz8siMNwnOzMuOBwARYQNr.cpp
            └── BeJwz8siMNwnOzMuOBwARYQNr.h
            └── BeJwz8siMNwnOzMuOBwARYQNr.xml
        └── standalone
            └── standalone.cpp
            └── standalone.h
            └── standalone.xml
        └── type
            └── BeJwrMSw2z@0tLk5MTwJAGG4EA1.cpp
            └── BeJwrMSw2z@0tLk5MTwJAGG4EA1.h
            └── BeJwrMSw2z@0tLk5MTwJAGG4EA1.xml

    └── toolkit.xml

16 directories, 35 files
Notice the number of files and directories that were generated by the Streams compiler. Do not let this intimidate you. At this time, the files in reverse print are the only ones you need to interact with. In the future, however, you will find it useful to understand some of the directories and artifacts created by the compiler. The src directory is particularly interesting. This is the location where Streams places all of the generated C++ files that result from SPL compilation.

The executable program created by the SPL compiler is named output/bin/standalone. Also note that there is a symbolic link file created called output/bin/HelloWorld, which points to the standalone executable. You can execute the program by specifying either standalone or HelloWorld.

6. Run the standalone program

$ output/bin/standalone

This returns:

Hello, world!

1.3 Run InfoSphere Streams Studio using a new workspace

InfoSphere Streams Studio supports multiple workspaces. This allows you to separate major projects that you are working on. In addition, it reduces the clutter in your project explorer tab. It is not necessary to use separate workspaces; however, many Streams developers find this a useful practice.

1. Double-click the Streams Studio desktop launcher.

In the Workspace Launcher, create a new workspace by typing a new name in an existing path, such as /home/streamsadmin/Workspaces/workspace6, and click OK.

Hint

If in a prior lab you had checked Use this as the default and do not ask again in the Workspace Launcher, you will not see that dialog again upon starting Streams Studio. In that case, use File > Switch Workspace > Other... to specify a new workspace.

2. Close the Task Launcher for Big Data view.

3. The InfoSphere Streams perspective should be the current perspective. If it is not, use Window > Open Perspective > Other... in the top Eclipse menu, and choose InfoSphere Streams.

4. Right click on the InfoSphere Streams button and choose Reset. Click OK.

Your Streams Studio layout returns to the default layout. This can come in handy if you remove views and tabs by mistake.
1.4 Import, build, and run the HelloWorld application

In addition to being able to create new Streams applications from within Streams Studio, you can import existing applications as a new project. The import will make a copy of the entire application directory and place it under the control of Streams Studio within the workspace directory on the file system.

1. Import the HelloWorld project.

   a. In the top Eclipse menu, choose File > Import...
   b. In the Import dialog, select InfoSphere Streams Studio > SPL Project; click Next >.
   c. Click Browse…; In the file browser, click streamsadmin (under Places) and select spllab in the list on the right. Click OK.
   d. HelloWorld is the only available project; it is automatically selected. Click Finish.

Note that the Console view reports that the project build was completed, but you do not see any output from the SPL compiler. There is a simple explanation for this.

Streams Studio can compile SPL applications automatically, but this requires that each main composite have at least one Build Configuration. Build configurations specify the compiler and execution options for the main composite. By default, a project that was compiled from the command line and is imported into Streams Studio does not have any build configurations defined.

5. Define a build configuration.

   a. In the Project Explorer, expand the HelloWorld project.
   b. Under HelloWorld, expand <default_namespace>.
   c. Right-click on the HelloWorld main composite and choose New > Standalone Build. In the dialog that pops up, accept all defaults and click OK.

Notice that the Compiler Invocation section at the bottom shows the command-line version of how Streams Studio will compile the main composite. It includes a few additional options for output and data directories because Streams Studio supports having both standalone and distributed builds available concurrently.

The application is automatically built (because the Eclipse option Project > Build Automatically is checked by default. See the output in the Console view. The Project Explorer shows that the HelloWorld main composite has a default build configuration: HelloWorld [Build: Standalone].

   d. Expand the HelloWorld main composite.
Notice that the **Standalone** build configuration is marked as **Active**. Each main composite can have multiple build configurations defined, but only one can be active at any one time. It is typical for main composites to have both a standalone and a distributed build configuration.

**6. Launch the HelloWorld application.**

**a.** In the **Project Explorer**, right-click the **HelloWorld** main composite and choose **Launch**.

**b.** In the **Edit Configuration** dialog, click **Apply**, and then **Continue**.

**Hint**
The **Edit Configuration** dialog allows you to configure the runtime settings for the main composite. If you want to suppress it when you launch in the future, scroll down to the bottom of the window and uncheck **Always prompt** before you click **Apply**.

The application runs quickly; the **Console** view shows the output of the program:

![Console view](image)

1.5 **Generate a Makefile for the command line version of HelloWorld**

Not everyone develops Streams applications using the Streams Studio IDE. As you have seen earlier in this lab, you can edit, compile, and run applications from the Linux command line. To improve productivity and consistency, the Linux make command can be used to reduce typing, always specify the right compiler directives, and only recompile what’s needed. Here is a compiler option that you only need to use once to generate a Makefile for your application. Once it is created, you can edit the Makefile as needed.

**Warning**
If you tell the SPL compiler to generate the Makefile after you have already edited it, the file will be overwritten and your changes will be lost.

Switch back to the **Terminal** window from sections 1.1 and 1.2 or open a new Terminal window and change directory to `/spllab/HelloWorld`.

**1. Compile** the HelloWorld.spl application, with the `-m` flag to generate a Makefile.

```
$ sc -m -T -M HelloWorld
```

This returns:

Creating the SPL Makefile.
Note
The application is not compiled, but the -T flag (for standalone build) must still be specified to control the Makefile generation.

____2. Display the generated Makefile.

$ more Makefile

```
.PHONY: all standalone distributed clean
SPLC_FLAGS ?= -a
SPLC = $(STREAMS_INSTALL)/bin/sc
SPL_CMD_ARGS ?=
SPL_MAIN_COMPOSITE = HelloWorld

all: standalone
standalone:
   $(SPLC) $(SPLC_FLAGS) -T -M $(SPL_MAIN_COMPOSITE) $(SPL_CMD_ARGS)
distributed:
   $(SPLC) $(SPLC_FLAGS) -M $(SPL_MAIN_COMPOSITE) $(SPL_CMD_ARGS)
clean:
   $(SPLC) $(SPLC_FLAGS) -C -M $(SPL_MAIN_COMPOSITE)
```

Notice that the default make directive (all) is set to **standalone**. In addition, note that **clean** and **distributed** directives were also created.

If you edit a generated Makefile, place compiler directives that affect both standalone and distributed compiles on the **SPLC_FLAGS** line. Notice the -a flag was included to provide optimized code as a best practice.

____3. Use the **Makefile** to clean out the generated files from the previous compile

$ make clean

This returns:

```
/opt/streams/InfoSphereStreams/bin/sc -a -C -M HelloWorld
```

____4. **List** the contents of the directory using the Linux **ls** command. Notice that the generated files and directories have all been removed. Cleaning an SPL application before checking it into source code control or exporting a project prevents carrying around artifacts that can be automatically generated, thus saving space.

$ ls

This returns:
__5. **Compile** the HelloWorld application using the Makefile and the default directive of standalone.

$ make

```
/opt/ibm/InfoSphereStreams/bin/sc -a -T -M HelloWorld
Creating types...
Creating functions...
Creating operators...
Creating PEs...
Creating standalone app...
Creating application model...
Building binaries...
  [CXX-type] tuple<rstring message>
  [CXX-operator] Hi
  [CXX-operator] Sink
  [CXX-pe] pe HelloWorld-a
  [LD-pe] pe HelloWorld-a
  [CXX-standalone] standalone
  [LD-standalone] standalone
  [LN-standalone] standalone
```
Lab 2 Expression Language, Functions, Types, and Toolkits

This lab provides an opportunity to work closely with the SPL expression language. In addition, we begin to consider organization and reuse of SPL artifacts through the use of toolkits. There are several toolkits provided with InfoSphere Streams; in this lab, however, we will create our own toolkit to hold SPL types and functions that we will reuse in later labs.

The primary goal in this lab is to build a set of functions that allow us to use the Beacon operator to generate random IP flow records: to build a simulator. This is a common technique when you are building Streams applications and do not have the actual source of data available in your development environment. It allows you to test functionality as well as workload levels and performance issues.

Lab 1 guided you through command line development and Streams Studio development in both text and graphical modes. Beginning with this lab and continuing through the remaining labs, it will be your choice whether to use the command line or Streams Studio. The instructions will be written for use within Streams Studio, but the tasks can just as easily be accomplished through the command line.

2.1 Create the streamstk.ip toolkit project and namespace

Organize your reusable types and functions in a toolkit with a namespace of streamstk.ip. Name the toolkit streamstk.ip as well.

Note
It is best practice to name the toolkit the same as the top-level namespace provided by the toolkit.

1. Create the toolkit project.

   a. Right-click in the Project Explorer and choose New > Project....
   b. In the New Project dialog, select InfoSphere Streams Studio > SPL Project.
   c. Keep Use default location checked. Click Next.
   d. Set the project name to streamstk.ip; click Next.
   e. Keep the defaults for Toolkit name, Toolkit version, and Required Streams version; for Description, enter anything you like, such as “Toolkit for Internet Protocol and Network Flow Information”.
   f. Under Dependencies, uncheck Toolkit Locations (this toolkit does not depend on other toolkits). Click Finish.

Before adding any artifacts to the toolkit, create the highest-level namespace: streamstk.ip. (In any toolkit, there may be additional, nested namespaces).

2. Create the streamstk.ip namespace.
__a. In the Project Explorer, expand the streamstk.ip project.
__b. Right-click on the streamstk.ip project; choose New > SPL Namespace.
__c. In the New Namespace dialog, set Namespace to streamstk.ip.
__d. Leave Folder path set to the default of streamstk.ip/streamstk.ip; click Finish.

Note
The Folder path defaults to using a single directory name for the entire namespace. The other option is to use separate directories for the streamstk portion of the namespace and the ip portion. This is not necessary unless you organize your artifacts at both levels. With multiple sub-namespaces, it is useful to use a combination of the two styles; e.g., streamstk.ip/streamstk.ip/adapters/netflow.

The streamstk.ip namespace is now listed within the streamstk.ip project.

2.2 Create the flowRec tuple type

One of the most commonly reused kind of SPL artifact in Streams is the tuple type. This can be used to create variables, define stream schemas, and even define specific attributes in stream schemas.

In this section you will create a simple type representing the type of information that can be found in the Cisco NetFlow or IPFIX specifications. This kind of data is commonly available through commercial network routers and can be used to characterize and monitor commercial networks. The structure of the data used in this lab is described in Table 2, below.

Table 2. Schema for net flow data

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>sourceIP</td>
<td>uint32</td>
<td>IP Address of the process sending the data. Use integer representation, not octet strings, for faster processing.</td>
</tr>
<tr>
<td>sourcePort</td>
<td>uint16</td>
<td>TCP/IP port at the source end of the connection</td>
</tr>
<tr>
<td>destIP</td>
<td>uint32</td>
<td>IP Address of the receiving process</td>
</tr>
<tr>
<td>destPort</td>
<td>uint16</td>
<td>TCP/IP Port at the receiving end of the connection</td>
</tr>
<tr>
<td>packets</td>
<td>uint64</td>
<td>Number of TCP/IP packets transmitted during the flow represented by this flow record</td>
</tr>
</tbody>
</table>

1 See, for example, [http://en.wikipedia.org/wiki/IP_Flow_Information_Export](http://en.wikipedia.org/wiki/IP_Flow_Information_Export)
| bytes | uint64 | Number of bytes transmitted during the flow represented by this flow record |

### Note
This lab does not implement the entire NetFlow specification, and it does not connect to an actual router feed of NetFlow information. The examples in this lab are meant to teach SPL programming using a pseudo-real world scenario, without overcomplicating the labs.

1. Create the `flowRec` type.
   
   a. Right-click on the `streamstk.ip` namespace and choose **New > SPL Source File**.
   
   b. In the **New SPL Source File** dialog, uncheck **Generate Main Composite**; you do not need a main composite because you are only creating type definitions.
   
   c. Set **File name** to `Types.spl`.
   
   d. Click **Finish**.

The `Types.spl` file is opened in the graphical editor, but this view is not useful because you are not designing a composite operator or main composite (application).

2. Switch to the text editor: **right-click** anywhere in the graphical editor and choose **Open with SPL Editor**. **Close** the graphical editor.

### Note
No object appears in the toolkit project at this time, because you have not defined any yet. You can, however, find the (empty but for the namespace) source file you’ve just created under the Resources section of the project listing, in the `streamstk.ip` folder.

3. In the editor, **enter** this type definition below the namespace declaration:

```plaintext
type flowRec = tuple <
    uint32 sourceIP,
    uint16 sourcePort,
    uint32 destIP,
    uint16 destPort,
    uint64 packets,
    uint64 bytes
>
```

2.3 Create a function to generate a random integer

SPL comes with a large library of functions in the standard toolkit; however, most of these are for generic purposes, and for manipulating the SPL data types. In this lab, you write your own functions using the SPL language. While you could write them within your application, they become reusable when you create them in a separate toolkit.

1. Create an additional SPL source file within the streamstk.ip toolkit, using the same procedure as in Section 2.2, Step 0 Keep Generate Main Composite unchecked, and use File name: Functions.spl.

The random number generator function that comes with streams is defined as:

```plaintext
public stateful float64 random()
```

This returns a random floating-point number between zero (inclusive) and one (exclusive). The data in the flow records of this lab consists of integers, and thus requires a function that generates a random integer. Model this function after the C rand() function, which generates an integer between 0 and MAX_RANDOM. Here, generate random numbers from the full range of a half word (16 bits: 0 to 65535).

2. Create a function to generate a random 32-bit unsigned integer as described above. Declare the function public because it needs to be available for use outside this namespace (streamstk.ip). In addition, the function must be declared stateful because its output is not deterministic due to the use of another stateful function within (random()).

For consistency with the examples and solutions throughout the lab, use this function prototype:

```plaintext
public stateful uint32 splRand() {...}
```

Enter the code for the function into the Functions.spl file, and remember that SPL is strongly typed and requires explicit casts.

```plaintext
// C-like rand function
// spl_rand(): Returns a uint32 between 0 and 65535
public stateful uint32 spl_rand()
{
    uint32 maxrand = 0x0000ffffu; // hex for 65535
    return (uint32)(random() * ((float64)maxrand + 1.0));
}
```
__3. **Save the file** to have the SPL compiler run a syntax check on your code. Any errors will be reported in the **SPL Build Console** view.

### 2.4 Generate a random integer between specified limits

The `spl_rand()` function from the previous step is a good start, but this application requires a limited range of random integers:

\[
\text{Min} \leq x \leq \text{Max}
\]

A common approach to this is the formula

\[
x = \text{Min} + r \mod (\text{Max} - \text{Min} + 1)
\]

where \(r\) is a random number within the maximum range for the data type (\(\mod\) is the modulo operator).

__1. Write a second function in Functions.spl that uses the function `spl_rand()` from Section 2.3 to satisfy this requirement.

```spl
// spl_rand(min,max): uint32 between min and max (max <= 65535)
public stateful uint32 spl_rand(uint32 min, uint32 max)
{
    return min + spl_rand() % ((max - min) + 1u);
}
```

Save the file.

---

**Note**

Remember: number literals must be specified with their length and type. Example: 150 is a signed 32-bit integer (int32) in SPL. The literal for an **unsigned** 32-bit integer requires a `u` suffix (150u).

### 2.5 Write a function to generate an IP address as an integer

To round out the set of functions needed to generate a simulated net flow, create one to generate **random IP addresses**. To cover the entire IP space, you could just generate random integers between 0 and 4294967295 (the full 32-bit range). The problem with this approach is that it is not realistic and you may never see the same IP address twice for quite some time. This would make for a rather boring summarization of net traffic.

The function you need should allow you to **mask the 4 octets** of the IP address (e.g., 192.168.1.1) so you can limit the range of IP addresses you generate.

The most familiar way to think about masking IP addresses is to use **hexadecimal** notation. SPL supports the use of hexadecimal notation for integers, where the data type size is inferred from the number of hex digits used. For example: 0xff would be interpreted as an int8 (-1), 0xa8u would be interpreted as a uint8 (168). The function in question requires uint32 variables and parameters, but the mask cannot exceed 255, so its hexadecimal representation needs to be zero-extended, as in: `0x000000ffu`.
Another performance gain in programming is the use of *shift operators* ($\ll$, $\gg$) and *bitwise operators*: *and* ($\&$), *or* ($|$), and *xor* ($^\$). SPL provides these operators, which helps create a very efficient IP address generator using masks.

**Note**  
These functions are not perfect, in that they cannot generate octets of 0 at this time. This may be fixed later.

```plaintext
1. **Type** or *copy/paste* the following functions into the “Functions.spl” file; **save** the file.

```plaintext
// Non-public function to generate a random octet based on  
// an ip mask. The mask is shifted depending on which octet  
// (0,1,2,3) is being generated.  
**stateful uint32** random_octet(uint32 index, uint32 ipmask)  
{  
    uint32 mask = (ipmask >> (8u * index)) & 0x000000ffu;  
    if (mask > 0u)  
        return spl_rand() % mask;  
    else  
        return 0u;  
}

// Generate random IP address.  
// Allow a mask of bits for each octet to control the  
// range of IPs to be generated.  
// Example: mask = 0x0101ffff would allow for the range:  
// 1.1.1.1 - 1.1.255.255  
**public stateful uint32** random_ip(uint32 ipmask)  
{  
    uint32 ip4 = 1u + random_octet(3u,ipmask);  
    uint32 ip3 = 1u + random_octet(2u,ipmask);  
    uint32 ip2 = 1u + random_octet(1u,ipmask);  
    uint32 ip1 = 1u + random_octet(0u,ipmask);  
    return ((ip4<<24) | (ip3<<16) | (ip2<<8) | (ip1));
}
```

### 2.6 View the toolkit information

You have added a type definition and several functions to your toolkit. In practice, you will most likely discover the need for these functions as you develop your applications. Having the toolkit available from the start makes the decision where to create artifacts easy. As a matter of good practice, every major application or set of applications should have at least one toolkit to go with it. This approach promotes reusability and organization.

A toolkit requires an index, that is, a file containing toolkit information such as a list of artifacts and their descriptions and details. When the toolkit is indexed, the index is stored in a file named *toolkit.xml*. If you look at the *toolkit.xml* file for our toolkit you should recognize the items contained.
__1. In the **Project Explorer**, expand the **Resources** folder within the **streamstk.ip** project.

   __a. Double-click on the file named toolkit.xml to open and review it.

      Never make changes to the toolkit.xml file manually.

      Notice that the .spl files (Types.spl and Functions.spl) are listed, as well as each type and function you created.

   __b. **Close** the file when you are finished looking at it.

2.7 **Create an application to test the toolkit**

You have done a lot of work so far, and in practice even this work would have been performed incrementally. For example, after creating the flowRec type, you could have created a simple program to generate default tuples of that type. In addition, you would normally test each function individually. While developing this lab, the author iterated back and forth between the toolkit and test applications quite a bit. It is just not practical to lay the lab out that way.

In this step, you begin to create a standalone SPL application that operates in a very similar manner to the **HelloWorld** application from 0. Instead of generating a single message, it generates a random NetFlow tuple. Use the **Beacon** operator and the **output** clause to make use of the functions you have created.

Before you can use your toolkit, **streamstk.ip**, in the new project, you have to make it aware of the toolkit by declaring its dependency.

__1. Create a new SPL Project named **NetBeacon**.

   __a. In the **New SPL Project** dialog, enter **Project name** NetBeacon; click **Next**.

   __b. In the **Dependencies** field, uncheck Toolkit Locations.

   __c. Scroll down in the **Dependencies** field; expand Workspace Projects.

   __d. Check streamstk.ip.1.0.0; click **Finish**.

__2. Create a new **Main Composite** in the new **NetBeacon** project.

   __a. In the **New Main Composite** dialog, accept the **default** namespace (blank).

   __b. Name the main composite **NetBeacon**.
__c.  Click **Finish**.

__d.  The NetBeacon*.spl* file opens in the graphical editor. Open it in the **SPL Editor**.

In addition to declaring the toolkit dependency, you can add a use directive to the top of the source file so that you do not have to fully qualify the types and functions when you use them.

__3.  Add the line

```plaintext
use streamstk.ip::*;
```

to the top of the file so you have access to all artifacts in the streamstk.ip namespace.

__4.  Code the **NetBeacon** main composite to create a small stream of flowRec tuples and print them to stdout. Actual code for the main composite is in this guide, at the end of this lab. You can complete the exercise by a combination of any of the following approaches:

- Code/Test/Debug your application without looking at the solution
- Code your application, and then review the solution before or during testing
- Type or copy/paste the solution
- Use the command line, Streams Studio in text mode, or graphical mode.

__a.  In the **NetBeacon** main composite, add a graph clause

__b.  Invoke a **Beacon** operator to generate flowRec tuples.

- The output stream name is *GenFlow*.
- Its schema is that defined by the flowRec tuple type defined in the toolkit.
- Use a parameter to limit the number of tuples produced to 5.
- Use the functions you created earlier to generate tuples that are initialized to random values according to the specifications in Table 3, below.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
</table>
| sourceIP | Random IP address limited to the range 1.1.1.1 – 1.1.1.255  
**Hint**: Use a mask of 0x010101ffu |
| sourcePort | Random uint16 between 1 and 9999  
**Hint**: `spl_rand(min,max)` returns a uint32, so you may need to cast the results to get a uint16 |
| destIP | Random IP address limited to the range 1.1.1.1 – 1.1.1.255  
**Hint**: Use the mask of 0x010101ffu |
<table>
<thead>
<tr>
<th>destPort</th>
<th>Random uint16 between 1 and 9999</th>
</tr>
</thead>
<tbody>
<tr>
<td>packets</td>
<td>Random uint64 between 100 and 200</td>
</tr>
<tr>
<td>bytes</td>
<td>Random uint64 between 1000 and 1500</td>
</tr>
</tbody>
</table>

5. **Invoke a Custom** operator to print each flowRec tuple produced by the **Beacon**.
   
   **a.** Take as input the GenFlow stream.
   
   **b.** Use the logic clause to print each tuple that arrives.

<table>
<thead>
<tr>
<th>Hints</th>
</tr>
</thead>
<tbody>
<tr>
<td>The standard function println(&lt;tuple name&gt;) can print the entire tuple, like .toString() in java. The logic clause, the stream name specified in the onTuple label can be referenced as a tuple object.</td>
</tr>
</tbody>
</table>

6. **Save the file** and check for syntax errors in the SPL Build Console view.

   Warning messages (CDISP0079W) are generated about side effects of stateful functions. Since our implementation is a random number generator, potential side effects are not an issue.

   Streams Studio automatically creates a distributed build configuration for each new SPL main composite. For simple testing, as we saw in 0, it is often useful to use a standalone configuration.

7. **Create a standalone** build configuration for the **NetBeacon** main composite.

   **a.** In the **Project Explorer**, expand the **NetBeacon** project, and `<default_namespace>`.
   
   **b.** Right-click on the **NetBeacon** main composite and choose **New > Standalone Build**.
   
   **c.** In the **NetBeacon – Standalone** dialog, check **Show toolkit path list**. Notice that the **Compiler invocation** section now shows the –t option with the path to the streamstk.ip toolkit.
   
   **d.** Click **OK**.

   The **NetBeacon** main composite is automatically recompiled (assuming the option **Project > Build Automatically** is still selected).

8. **See the output in the Console view.** Notice that the compiler command line is still building the distributed version. This is because the **Distributed** build configuration is set as the **Active** build configuration; switch to the **Standalone** build. (It’s possible to build and launch a configuration that is not the Active one, but in this case Standalone should be made Active.)

   **a.** In the **Project Explorer**, expand the **NetBeacon** main composite.
   
   **b.** Right-click on the **Standalone** build configuration and choose **Set active**.
The **NetBeacon** main composite is now recompiled in standalone mode.

__9. Launch the **NetBeacon** standalone application.

  _a._ In the **Project Explorer**, right-click on the **NetBeacon** main composite and choose **Launch**.

  _b._ In the **Edit Configuration** dialog, click **Apply** and then **Continue**.

The application runs quickly; the **Console** view shows the output of the program.

![Console output](image)

Notice the output format of tuples using the `println()` function. This function is very useful for debugging tuple data and composite data types.

Although this is not actual NetFlow data, it should be satisfactory for us to develop additional functionality.

__10. Close the **NetBeacon.spl** editor view.__

### 2.8 Lab 2 Solution

**NetBeacon.spl**

```plaintext
composite NetBeacon {
  graph
  stream<flowRec> GenFlow = Beacon() {
    param
      iterations : 5u;
    output
      GenFlow : 
        sourceIP = random_ip(0x010101ffu),
        sourcePort = (uint16) spl_rand(1u, 9999u),
        destIP = random_ip(0x010101ffu),
        destPort = (uint16) spl_rand(1u, 9999u),
        packets = (uint64) spl_rand(100u, 200u),
        bytes = (uint64) spl_rand(1000u, 1500u);
    }
  }
  ()as BeaconSink = Custom(GenFlow as BeaconStream) {
    logic
  }
}```
IMPLEMENTATION BEST PRACTICE: The use of an input stream alias for operator implementations (e.g. as `BeaconStream` in `BeaconSink` above) allows future iterations of the program to change which stream is processed without having to edit the `onTuple` portion of the Custom operator or any fully-qualified tuple field names in the logic. A new stream can be inserted by just changing the input stream name (e.g. `GenFlow` in the example above) to the name of the new stream. The invocation body can continue to refer to the alias.
Lab 3   SPL Adapters: TCPSource and TCPSink

This lab gives you an opportunity to become familiar with the TCPSource and TCPSink operators. In addition, it demonstrates a common development and test strategy.

File-based test feeds for streaming applications are great for deterministic testing, functional testing, and of course when the production data itself is file-based. They do not always scale well, and often require restarts and time consuming file manipulation to drive a large volume of data or data at a high velocity.

This lab introduces a unique way of building a streaming data generator using a standalone SPL application. You will take the NetBeacon application and turn it into a streaming data test feed that passes data using a Linux TCP/IP socket. In the previous lab you wrote SPL functions that let you generate test data similar to what actual data would look like, while still being random.

In addition, you will go through the process of duplicating an SPL Project in Streams Studio and making the changes that are required to avoid conflicts within the same workspace.

Note
The solution to the lab steps that require SPL programming can be found at the end of this chapter.

3.1 Duplicate the NetBeacon Project

While it is possible to duplicate an SPL project in Streams Studio, the refactoring of main composites and files is not automatic.

1. **Copy** the NetBeacon project.
   a. In the **Project Explorer**, collapse (but do not close) your projects.
   b. Right-click on the NetBeacon project and choose **Copy**.
   c. Right-click on an open area within the Project Explorer and choose **Paste**.
   d. In the **Copy Project** dialog, change the **Project name** to NetGen and click **OK**.

Notice that a white-on-red "x" appears on the NetGen project and the toolkit name (in gray) is still NetBeacon.
3.2 Refactor the NetGen Project

When you duplicate a project in Streams Studio, there are several artifacts that need to be refactored, including:

- The toolkit name (this is maintained in the info.xml file)
- The main composite name
- The main composite SPL filename if it is the same name as the main composite. (This is not necessary, but if that is your standard, it is good practice.)

2. Change the toolkit name.
   a. In the Project Explorer, right-click on the NetGen project; choose Edit Toolkit Information....
   b. In the Toolkit NetBeacon dialog, change the Toolkit name to NetGen.
   c. Click OK.

3. Change the main composite SPL file name.
   a. Expand the NetGen project, and the Resources folder within that.
   b. Right click on NetBeacon.spl and choose Rename....
   c. In the Rename Resource dialog, set the New name to NetGen.spl. Click OK.
   d. Collapse the Resources folder.

4. Change the name of the main composite.
   a. Double-click on the NetBeacon main composite within the NetGen project to open the file containing the main composite.
   b. Change the name of the main composite to NetGen and save the file.

   Notice that the Project Explorer is updated to reflect the new name.

5. Review the SPL Build Console view to check that the code compiled. Even though this is the NetBeacon code, it shows that the refactoring is complete.

3.3 Turn the test application into a simulator

The reason for copying the NetBeacon project is that it already has most of the code you need in its main composite. The key to the streaming test driver is the Beacon operator that generates random flow data. In this step you replace the Custom operator that writes the tuples to stdout with a TCPSink operator that streams tuples to any process that connects via TCP.
Note
The client process that connects to the TCPSink operator does not need to be a streams application. You will see that later in the lab. The following instructions assume that you’re using the SPL text editor in Streams Studio.

_1._ **Comment out** the Custom operator invocation. A quick way to do this is to highlight the entire invocation, right-click and choose **Toggle Comment**; or type Ctrl+/.

_2._ Add a TCPSink operator to send out each flowRec tuple produced by the Beacon operator.
   - Take as input the GenFlow stream.
   - Perform the role of a TCP server.
   - Listen on port 5555 for incoming connections.
   - Allow infinite retries to accept connections after they are broken.
   - Produce data using the default csv format.
   - Flush the network buffer after every 10 tuples.

Note
Frequent buffer flushing would not necessarily be done in production, but it helps provide more responsive testing feedback.

_3._ **Save** your program and check for syntax errors in the SPL Build Console.

During the initial testing of the Beacon operator to generate flow data it was important to just create a small sample set so you could validate the format and accuracy of the data generation functions. Now use the Beacon to generate streaming data, but adjust it to generate data continuously, producing 10 tuples per second. Later on, you will “open it up” and let it produce data as fast as it can, but for now, you just need to make sure data is flowing over the TCP socket.

_4._ Modify the Beacon operator.
   - Eliminate any limit on the number of tuples produced.
   - Produce a tuple every 0.1 seconds.
   - Delay for 1 second before producing the first tuple.
   - Save your program and check for syntax errors in the SPL Build console view.

### 3.4  Launch and test the NetGen standalone application

Because you copied the NetBeacon project, the NetGen main composite already has a Standalone build configuration, designated as the Active (default) build.

You do not need a custom-made data consuming application to test the new program. All you need is the netcat utility (nc command) that is included with most Linux distributions (but needs to be installed separately in RHEL 6).
1. **Launch** the NetGen main composite as a standalone application.

The console output for the NetGen application is displayed, but no messages show up. This is normal. The only time the application prints a message is when an exception occurs. If you coded the TCPSink properly, you will be able to start and stop downstream applications without breaking the NetGen program.

2. Open a new Terminal window by using the desktop or panel launcher, or by right-clicking on the desktop and choosing Open in Terminal.

3. Start the netcat program as a client connecting to port 5555 on this host (e.g., streamslab). The -v (verbose) option gives feedback about the status of the connection.

   $ nc -v streamslab 5555

   Connection to streamtraining 5555 port [tcp/personal-agent] succeeded!
   16843063,5976,16843061,8368,101,1047
   16843249,8510,16843177,15,112,1001
   16843040,9263,16843193,2193,156,1256
   16843119,4446,16843125,9880,161,1209
   16843194,4477,16843139,8958,160,1085
   16843026,1193,16843030,4443,105,1300
   ...

4. Use the keystroke <Ctrl>-C to cancel the netcat program.

5. **Start** and **stop** netcat several times.

You can leave the NetGen standalone program running while you perform the second part of this lab. If you do need to cancel it, use the red square on the NetGen Console view title bar.

### 3.5 Create the NetReceive project by duplicating the NetBeacon project

Now that you have tested half of the solution, you need to create the streams application that will ingest and process the flow data produced by the NetGen application. As with NetGen, rather than starting from scratch it is a good idea to start from a similar application. The original NetBeacon application used a Custom operator to display the data it received on stdout. As an incremental step toward building a more advanced Streams program, it is always a good idea to verify that you can attach to your data sources and receive data.

In this section you will perform the same steps as you did to create the NetGen application in order to create the NetReceive application. Refer back to sections 3.1 and 3.2 for more detailed instructions as necessary.

1. **Duplicate** (copy/paste) the NetBeacon project. Name the new project: NetReceive.

2. **Refactor** the NetReceive project to remove conflicts and use a main composite name that matches the project name.


__a.  Change the toolkit name in the info.xml file to NetReceive.
__b.  Change the name of the SPL file containing the main composite to NetReceive.spl.
__c.  Change the name of the main composite to NetReceive.

3.  Verify that the application compiles (even though the code is still that of NetBeacon).

3.6  Replace the Beacon operator with a TCPSource operator

Rather than generating its own data, the NetReceive application is to receive data on a TCP socket. The first step in transforming the original NetBeacon application is to replace the Beacon operator with a TCPSource operator. Configure the TCPSource operator as a TCP client, attaching to an open socket which produces flow data.

__1.  Comment out the Beacon operator invocation.
__2.  Add a TCPSource operator to connect to the TCP Socket provided by the NetGen application and emit each tuple to the existing Custom operator.
   __a.  Call the output stream GenFlow.
   __b.  Use the flowRec tuple type from the toolkit as the schema for GenFlow.
   __c.  Assign the role of a TCP client.
   __d.  Connect to port 5555 on your current host ("localhost").
   __e.  Allow infinite retries to connect to the given port and host if the connection is broken.
   __f.  Use the csv format (which is the default).
__3.  Save your program and check for syntax errors in the SPL Build console view.

3.7  Test the NetReceive program in standalone mode

Eventually you will return to running SPL applications in a Streams instance. For now, simple functional testing and initial connectivity testing is easier in standalone mode because you can send output to stdout and see it in the console.

__1.  Ensure that the NetGen standalone program is running.
2. Launch the **NetReceive** application in standalone mode.

3. Notice that the `flowRec` tuples start appearing in the **Console** view.

### Note

By default, Streams Studio attempts to bring the console with the most recent output to the front. In the case of NetReceive, it is continuously printing output, making it difficult to bring the NetGen output to the front.

To allow you to switch between consoles while live data is being displayed, press the **Pin Console** toolbar button on the Console view title bar.

4. **Cancel** the **NetGen** application and **restart** it. What happens to **NetReceive**?

### 3.8 Create a function to format IP addresses (*xxx.xxx.xxx.xxx*)

Until now, you have only viewed generated IP addresses as integers. The integer version is recommended for passing, manipulating, and computing data within Streams applications. Numeric operations are a lot faster than string operations.

For humans, however, we sometimes need the data displayed in a format we can understand. In this step you use a simple function to convert an IP address from integer to string. You don't have to write it yourself because it uses bit masking and shifting and that is not the point of this lab. It will force you to make a change in the Custom operator, however, which is using the `println()` function to print the entire incoming tuple; to make use of the new formatting function, it will have to do something else instead.

You can create the function in either your application program or the `streamstk.ip` toolkit. It makes the most sense for it to be in the toolkit, so put it there for now.

1. **Open** the `streamstk.ip` toolkit project if it is not already open.

2. **Right-click** on any of the **functions** in the toolkit and choose **Open With > SPL Editor** to open the `Functions.spl` file.

3. Type or **copy/paste** the following function at the end of the `Functions.spl` file:

   ```spl
   //Convert a uint32 ip address to dot notation
   public rstring ipToString(uint32 ip)
   {
       return (rstring)((ip >> 24) & 0x000000ffu) + "." +
               (rstring)((ip >> 16) & 0x000000ffu) + "." +
               (rstring)((ip >>  8) & 0x000000ffu) + "." +
               (rstring)(ip       & 0x000000ffu) ;
   }
   ``

4. **Save** the file.

This automatically indexes the `streamstk.ip` toolkit, which makes the new function available for use in the **NetReceive** program.
3.9 Enhance NetReceive to print human-readable IP addresses

The current invocation of Custom in the NetReceive application uses println() to print the entire tuple followed by a newline. There is another function, print(), which can print an SPL data type without a newline. In addition, the standard toolkit includes two functions for printing rstrings without the quotes: printString() and printStringLn() will print with and without a newline, respectively.

5. **Change** the Custom operator to print attributes of the incoming flowRec tuples on a single line, using the ipToString() function.

The exact format is up to you, but here is a sample format you can use as a guide:


You have the option of writing the code directly in the body of the Custom operator's logic onTuple clause, or creating an SPL function to encapsulate the printing, and call the function from the onTuple clause.

6. **Save** your program and correct any syntax errors.

7. **Cancel** the currently running version of NetReceive (if it is still running).

8. **Launch** your new version.

9. Identify errors in your output, fix your code, save, and relaunch.

Your final output may look like this:

---

10. **Terminate** the NetReceive job.
3.10 Lab 3 Final Solutions

NetGen.spl

```spl
use streamstk.ip::* ;

composite NetGen {
  graph
    stream<flowRec> GenFlow = Beacon(){
      param
        initDelay : 1.0;
        //iterations : 5u ;
        period : 0.1;
      output
        GenFlow : sourceIP = random_ip(0x010101ffu),
                   sourcePort = (uint16)spl_rand(1u, 9999u),
                   destIP = random_ip(0x010101ffu),
                   destPort = (uint16)spl_rand(1u, 9999u),
                   packets = (uint64)spl_rand(100u, 200u),
                   bytes = (uint64)spl_rand(1000u, 1500u);
    }
  ()as NetGen = TCPSink(GenFlow){
    param
      role : server ;
      port : "5555" ;
      flush : 10u ;
  }
}```
use streamstk.ip::*;

// Function to print a flowRec
// Could have also been coded to just return a string
void printFlowRec(flowRec rec)
{
    printString("[srcIP:" + IPIntToString(rec.sourceIP));
    printString(", srcP:" + (rstring) rec.sourcePort);
    printString(", destIP:" + IPIntToString(rec.destIP));
    printString(", destP:" + (rstring) rec.destPort);
    printString(", pkts:" + (rstring) rec.packets);
    printStringLn(" , bytes:" + (rstring) rec.bytes + "]");
}

composite NetReceive {
    graph
        stream<flowRec> GenFlow = TCPSource()
        {
            param
                role : client;
                address : "streamtraining";
                port : "5555";
        }

        ()as BeaconSink = Custom(GenFlow as BeaconStream)
        {
            logic
                onTuple BeaconStream : {
                    // println(BeaconStream);
                    // println(BeaconStream);
                    printFlowRec(BeaconStream);
                }
        }
    }
Lab 4  SPL Aggregation and Sorting

In this lab you will build your first streaming analytics in SPL. The previous labs have concentrated mostly on the plumbing that is required when you first begin to work with a new type of data in Streams. In addition, you have seen some best practices for organizing artifacts into toolkits, and for building a streaming test driver.

The goal in this lab is to process the streaming flowRec data to find the top five IP addresses that have been the source of the most IP traffic. Unlike in systems that operate on stored data, there is no end to the data that streaming into your application, and yet at any one time you need to look at more than just the current tuple. For that reason you will apply the stream processing concept of a window. Your application will produce the top five IP addresses for each window of data processed. This produces a continuous stream of top-5 results, which, due to window-based aggregation, flows at a lower and more manageable rate than the raw NetFlow input data.

The flow of data through this analytic will be as follows:

![NetFlow Summary]

One goal of this design is to reduce the amount of data flowing through the system as early in the flow as possible, which improves performance.

You will remove the “speed limit” you placed on the NetGen application and let it produce data as fast as it can.

Finally, you will utilize the Streams Instance and a distributed build configuration so that you can monitor the data flow using Streams Studio’s Instance Graph.

4.1  Create the NetSummary project by duplicating the NetReceive project

You should be getting really good at using an existing SPL project as a starting point for new work. The new NetSummary project will use the data streaming from NetGen.

Note
In this lab, you will copy the NetReceive program because that was a step along the way as you incrementally add functionality. In real work, when you have multiple analytical applications that need access to the streaming data, you would create an ingest application that receives the socket data and use the Export operator to make the data available to other SPL applications running in the same instance. Consider that a challenge problem for the end of the day.

__1.  Duplicate (copy/paste) the NetReceive project. Name the new project: NetSummary.
Refactor the NetSummary project to remove conflicts and use a main composite name that matches the project name.

b. Change the toolkit name in the info.xml file to NetSummary.

c. Change the name of the SPL file containing the main composite to NetSummary.spl.

d. Change the name of the main composite to NetSummary.

e. Verify that the application compiles (even though the code is still that of NetReceive).

4.2 Aggregate the streaming data in memory and produce a flow of Source IPs with statistics over a period of time

Take flowRec tuples as input and produce a stream of summary records for each source IP address. Table 4, below, describes the summary data that should be produced for each unique Source IP address received during a window of time.

<table>
<thead>
<tr>
<th>Summary Attribute</th>
<th>Description</th>
<th>Aggregation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ip (uint32)</td>
<td>Source IP from flowRecs</td>
<td>None: Any(sourceIp)</td>
</tr>
<tr>
<td>numFlows (int32)</td>
<td>Number of flowRec tuples accumulated in this period of time</td>
<td>Count the number of flowRecs being aggregated: Count()</td>
</tr>
<tr>
<td>sumPackets (uint64)</td>
<td>Total number of packets sent by the current Source IP in this period</td>
<td>Sum the packets attribute from the flowRec tuples: Sum(packets)</td>
</tr>
<tr>
<td>sumBytes (uint64)</td>
<td>Total number of bytes sent by the current Source IP in this period</td>
<td>Sum the bytes attribute from the flowRec tuples: Sum(bytes)</td>
</tr>
</tbody>
</table>

**Hint**
You can describe each attribute of the schema being produced by the Aggregate operator in the declaration of the output stream, or you can define a tuple type and use that. The choice is yours.

1. Add an Aggregate operator to produce the summary tuples described in Table 4, above.
   a. Consume the GenFlow stream from the TCPSource.
   b. Output a stream as specified in Table 4, above.
c. Use a **tumbling window** that holds **30 seconds** of input data.

d. **Group** the results by `sourceIP`,

No need to worry about `flowRec` attributes that are not aggregated or passed through to the output stream.

### 2. Modify the **Custom** operator to print the tuples that are emitted by the **Aggregate** operator.

Either code it directly in the `onTuple` clause or write an SPL function to encapsulate the logic.

### 3. Save your file and correct any errors.

### 4. Run your application in **standalone** mode and verify the results in the **Console**.

#### Note
Because the `Aggregate` operator has a 30-second window, the application does not produce any output until 30 seconds have passed. After that, it continues to present a list of the unique source IP addresses with summarization statistics every 30 seconds.

### 5. **Cancel** the application.

#### 4.3 Sort the summary records by IP address

The summary tuples are not in order. To make it easier to produce a list of the top five source IPs you will need to put the tuples in order.

In this step you use the **Sort** operator to sort the summary records produced by the **Aggregate** operator. This will make it easier to grab the top five (or top however-many) IP addresses. We will take advantage of the **punctuation-generating** feature of the **Aggregate** operator to identify when a batch of data to be sorted has arrived and is complete.

1. **Add a** **Sort** operator to do the following:
   - **Receive** the output stream from the **Aggregate** operator.
   - **Output** a stream with the same schema as the input stream.
   - Use a **tumbling window** that holds all of the data until **window punctuation** is received on the input stream (`punct()`).
   - **Sort** the output in **descending** order by the `sumBytes` attribute.

#### Hint
As a shortcut to output schema declaration, you can use the name of the input stream in the schema definition of the output stream (e.g., `stream <InStream> OutStream = Sort(InStream) {...}`).

2. **Modify** the **Custom** operator to print the tuples that are emitted by the **Sort** operator.
This should only require a change to the input stream, but that single change can cause a change in the onTuple clause and print statements if they all refer to the incoming stream name.

<table>
<thead>
<tr>
<th>Hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is why it is a good idea to use an input stream alias, even for operators with only a single input stream. The alias can be used in the body of the operator invocation, so a change to the input stream name does not have this ripple effect.</td>
</tr>
</tbody>
</table>

3. Save your file and correct any errors.

4. Run your application in standalone mode and verify the results in the Console. The behavior and timing of output should be the same as in the previous version; however, the data should now appear sorted by size within each batch.

5. Try sorting your output by other attributes as well (e.g., ip); be sure to change it back to sorting by sumBytes before moving on.

6. Cancel the job.

4.4 Find the top 5 source IP addresses by bytes produced

As with most programming assignments, there are multiple ways to accomplish a task. In this section you will use a new instance of the Custom operator to find the first 5 tuples that arrive from the Sort. These should be the 5 IPs that were the source of the most bytes during the observation period (30 seconds in this case).

Use the Streams expression language along with a unique feature of the Custom operator. The Custom operator is the only operator that can use the submit() function to send tuples to one of its output ports.

This is a challenging project for anyone learning SPL. The solution is not complicated, but thinking in terms of tuples and punctuation is new to many.

The following pseudocode describes the logic flow you are being asked to implement.

**Pseudocode: Emit the first n tuples from each batch of arriving tuples**

A batch is made up of all incoming tuples until a window punctuation marker arrives, signaling the end of a batch.

Initialize a constant, n, that represents how many tuples you want to emit from each batch that arrives.

Define a mutable counter, i, count tuples as they arrive. Initialize it to 0 (zero).

When a tuple arrives:

If the counter is within the range 0 (inclusive) to n (exclusive):
Call submit() to send the tuple to the output stream
Increment the counter

If it is not:

You could increment the counter, but this is not necessary, because once the counter = n, the prior condition will never be met. Incrementing the counter no longer serves a purpose.

When a punctuation marker arrives, this signals the end of the batch
Reset the counter to 0
Call submit() to send the punctuation to the output stream. This allows downstream operators to be aware of our batches of n tuples.

__1. Add a Custom operator to implement the pseudocode above.
   ● Receive the output stream from the Sort operator
   ● Output a stream with the same schema as the input stream
   ● Use the logic clause with the onTuple and onPunct labels to implement the logic
   ● There is a system constant that represents window punctuation: Sys.WindowMarker

__2. Modify the previously existing Custom operator to print the tuples coming from the new Custom operator
   Once again, this should only require a change to the input stream name

__3. Modify the previously existing Custom operator to print a string each time it receives a window punctuation marker.
   Use the logic onPunct clause to print the string "*** Window Punctuation ***"
   This lets you verify that the upstream Custom operator is in fact forwarding punctuation.

__4. Save the file and correct any errors.

__5. Run the application in standalone mode and verify the results in the Console. The output should look similar to the picture below.
6. Stop the application.

7. Unpin your Console views if they were pinned.

4.5 Compile and launch the application in distributed mode

You should already have an instance (LabInstance); verify that it exists and is running. If necessary, start it (using the Streams Instances Manager, the streamtool command, or the Streams Explorer view in Streams Studio).

Note
If you shut down and restarted the Virtual Machine in which you are doing this lab without stopping the instance first, it will be in an undefined state. In that case, perform a forced stop:

- Stop > check Force the instance to stop > OK (Instances Manager)
- streamtool stopinstance --force (command)
- right-click > Stop Instance --force (Streams Studio)

and then restart it.

1. Show the Instance Graph: In the Streams Explorer, right-click and choose Show Instance Graph.

2. Make the distributed build configuration Active.
   a. In the Project Explorer, expand the NetSummary main composite.
   b. Right-click on the Distributed build configuration and choose Set Active. Streams automatically rebuilds the application in distributed mode.
   c. View the compiler output in the SPL Build Console view.

3. Launch the job in your private instance.
   a. Right-click on the NetSummary main composite and choose Launch Active Build Config.
   b. In the Edit Configuration dialog, Click Apply; click Continue. The application now runs as a job in your private instance.
4. Make the **Instance Graph** tab visible to view the job graphically.

5. Select **Flow under 100 [nTuples]** in the **Color Schemes** panel on the right. You should see the following (colors will change over time):

![Instance Graph Tab Visibility](image)

Notice that the highest tuple rate is in the GenFlow (**TCPSource**) and Summaries (**Aggregate**) operators. This is what you would expect, as the downstream rate is much reduced by aggregation.

6. Hover over the GenFlow (**TCPSource**) operator and notice that the tuple rate is 10/sec. You can also note the amount of memory being used by the **Aggregate** operator (~600 kB)

Leave the **NetSummary** job running.

### 4.6 Remove the “speed limit” from the NetGen program

Up until this point you have been doing incremental development and functional testing of the **NetSummary** application. In this section you will let the **NetGen** standalone program generate data as fast as it can and see how it affects the performance metrics of your application.

#### Note

Depending on the platform where you are running this lab, your performance metrics may vary greatly from the examples in this lab guide.

7. In the **NetGen Console** view, **stop** the **NetGen** program by clicking the stop button in the title bar. You can also remove this launch from the Console by clicking the Remove Launch button.

8. Remove the speed restriction.

   a. Open the **NetGen** main composite in the SPL editor.

   b. Comment out the **period** parameter in the **Beacon** operator.

   c. Save the file. Notice that a simple change like that is a very fast recompile for the Streams compiler.

9. Run the **NetGen** program in standalone mode.
_10. View the **Instance Graph**, and hover over the **TCPSource** and **Aggregate** operators.

Incoming **tuple rates** are now much higher than 10/sec. On this author’s laptop, the flow averaged 60,000 tuples per second.

Every 30 seconds the **Aggregate** operator has a full window and emits the summary tuples. The memory size of the Aggregate operator on the lab development machine was ~600 kB. Note that this is the same as it was at the slower rate. Since a tumbling window is being used, and only simple sums and counts, the Aggregate operator can use an efficient method to calculate the results, which only keeps intermediate results in memory, not the incoming tuples. This makes memory consumption independent of the tuple rate.

### 4.7 View the Console Log

Because the **NetSummary** application is running in the Streams instance and not as a standalone program, the output produced by the **Custom** operator at the end of the processing flow is redirected to a console log file in:

```
/tmp/streams.LabInstance@streamsadmin/jobs/<nj>/pec_pe<npe>.stdouterr
```

(where `<nj>` is the job number and `<npe>` is the highest PE number).

Rather than using a Terminal window to view this file, use Streams Studio to pull the log back from whatever host it is running on and display it in the Console view.

__1. Right-click on the **BeaconSink** Custom operator and choose **Show Log > Show PE Console**. Click OK once the progress dialog has been replaced by the **Get PE consoles** notification.  

__2. View the log in the **Console**. Scroll to the bottom to see the new, higher numbers.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is a point-in-time snapshot, not a live, scrolling view.</td>
</tr>
</tbody>
</table>

If the job was running when **NetGen** was producing 10 tuples per second, you will find that the top IPs were reporting on average 4-6 flow records (the `numFlows` attribute) per batch. After using the **NetGen** at full speed, they may average 7000+ flow records in the 30-second window.

Flow data before removing the period parameter:

```
[IP:1.1.1.29, numFlows:4, sumPackets:554, sumBytes:5594]  
[IP:1.1.1.204, numFlows:4, sumPackets:476, sumBytes:4893]  
"*** Window Punctuation ***"  
```
### 4.8 Sample the Tuple Data

Streams 3 added a new feature to view live tuple data.

1. Right-click on the stream between the **Aggregate** and the **Sort** operator, and choose **Show Data**. In the **Data Visualization settings** dialog, click **OK** to select the defaults (all tuple fields, default collection rate).

2. A Properties view appears showing the data in a table; resize and move the view as needed.

<table>
<thead>
<tr>
<th>Time</th>
<th>ip</th>
<th>numFlows</th>
<th>sumPackets</th>
<th>sumBytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>7/26/13 6:44:07 PM PDT</td>
<td>16843227</td>
<td>7569</td>
<td>1133553</td>
<td>9457991</td>
</tr>
<tr>
<td>7/26/13 6:45:57 PM PDT</td>
<td>16843226</td>
<td>7521</td>
<td>1130510</td>
<td>9411606</td>
</tr>
<tr>
<td>7/26/13 6:45:57 PM PDT</td>
<td>16843225</td>
<td>7630</td>
<td>1146488</td>
<td>9532533</td>
</tr>
<tr>
<td>7/26/13 6:45:57 PM PDT</td>
<td>16843224</td>
<td>7548</td>
<td>1134789</td>
<td>9436649</td>
</tr>
<tr>
<td>7/26/13 6:45:57 PM PDT</td>
<td>16843223</td>
<td>7612</td>
<td>1135668</td>
<td>9518256</td>
</tr>
<tr>
<td>7/26/13 6:45:57 PM PDT</td>
<td>16843222</td>
<td>7490</td>
<td>1124963</td>
<td>9348748</td>
</tr>
<tr>
<td>7/26/13 6:45:57 PM PDT</td>
<td>16843221</td>
<td>7581</td>
<td>1138977</td>
<td>9468070</td>
</tr>
<tr>
<td>7/26/13 6:45:57 PM PDT</td>
<td>16843220</td>
<td>7610</td>
<td>1138654</td>
<td>9513335</td>
</tr>
<tr>
<td>7/26/13 6:45:57 PM PDT</td>
<td>16843219</td>
<td>7482</td>
<td>1120093</td>
<td>9368076</td>
</tr>
</tbody>
</table>

This view refreshes every 3 seconds (by default) and shows a sample of ten tuples. It can be quite handy when debugging an application, or just to get familiar with the data.
3. Click the **Stop** (Stop the Data Visualization session) button in the Properties view’s toolbar and close the view.

With the **NetGen** data generator running flat out, it’s interesting to see the resource consumption by the Streams application and services. Use the **System Monitor** (in RHEL 6, Applications > System Tools > System Monitor) or the Linux `top` command in a Terminal window to explore.

```
top - 18:54:35 up 22:10,  2 users,  load average: 2.36, 2.54, 2.19
Tasks: 177 total,   1 running, 176 sleeping,  0 stopped,  0 zombie
Cpu(s): 41.9%us, 20.6%sy,  0.0%ni, 29.0%id,  0.0%wa,  0.0%hi,  8.5%si,
        0.0%st
Mem:   1918812k total, 1656484k used,  262328k free,  38052k buffers
Swap:  4063224k total,    31044k used, 4032180k free,  414724k cached
```

<table>
<thead>
<tr>
<th>PID</th>
<th>USER</th>
<th>PR</th>
<th>NI</th>
<th>VIRT</th>
<th>RES</th>
<th>SHR</th>
<th>%CPU</th>
<th>%MEM</th>
<th>TIME+</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>1771</td>
<td>streamsa</td>
<td>20</td>
<td>0</td>
<td>765m</td>
<td>31m</td>
<td>20m</td>
<td>54.2</td>
<td>1.7</td>
<td>22:38.28</td>
<td>streams-pec</td>
</tr>
<tr>
<td>1762</td>
<td>streamsa</td>
<td>20</td>
<td>0</td>
<td>1379m</td>
<td>43m</td>
<td>21m</td>
<td>49.2</td>
<td>2.3</td>
<td>20:26.42</td>
<td>streams-pec</td>
</tr>
<tr>
<td>6840</td>
<td>streamsa</td>
<td>20</td>
<td>0</td>
<td>413m</td>
<td>27m</td>
<td>19m</td>
<td>32.6</td>
<td>1.5</td>
<td>14:27.88</td>
<td>standalone</td>
</tr>
<tr>
<td>9699</td>
<td>streamsa</td>
<td>20</td>
<td>0</td>
<td>2637m</td>
<td>412m</td>
<td>32m</td>
<td>4.3</td>
<td>22.0</td>
<td></td>
<td>java</td>
</tr>
<tr>
<td>3643</td>
<td>streamsa</td>
<td>20</td>
<td>0</td>
<td>1848m</td>
<td>395m</td>
<td>28m</td>
<td>0.7</td>
<td>21.1</td>
<td>1:14.48</td>
<td>java</td>
</tr>
<tr>
<td>1950</td>
<td>informix</td>
<td>10</td>
<td>-10</td>
<td>290m</td>
<td>44m</td>
<td>44m</td>
<td>0.3</td>
<td>2.4</td>
<td>03:39.00</td>
<td>oninit</td>
</tr>
<tr>
<td>2440</td>
<td>streamsa</td>
<td>20</td>
<td>0</td>
<td>431m</td>
<td>17m</td>
<td>22m</td>
<td>0.3</td>
<td>1.7</td>
<td>1:05:55</td>
<td>streams-aas</td>
</tr>
<tr>
<td>3321</td>
<td>streamsa</td>
<td>20</td>
<td>0</td>
<td>751m</td>
<td>31m</td>
<td>22m</td>
<td>0.3</td>
<td>1.7</td>
<td>1:06:69</td>
<td>streams-srm</td>
</tr>
<tr>
<td>3340</td>
<td>streamsa</td>
<td>20</td>
<td>0</td>
<td>1326m</td>
<td>32m</td>
<td>22m</td>
<td>0.3</td>
<td>1.7</td>
<td>11:18:26</td>
<td>streams-srm</td>
</tr>
<tr>
<td>1</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>19352</td>
<td>1344</td>
<td>1112</td>
<td>0.0</td>
<td>0.1</td>
<td>00:35.2</td>
<td>init</td>
</tr>
<tr>
<td>2</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>00:00.01</td>
<td>kthreadd</td>
</tr>
</tbody>
</table>

4. **Cancel** the **NetSummary** job: **Right-click** in the main composite in the Instance Graph and choose **Cancel Job**. You can also use the **Streams Explorer** view to cancel the job.

5. **Stop** the **NetGen** standalone program.

### 4.9 Lab 4 Final Solutions

NetSummary.spl

```plaintext
use streamstk.ip::*;
type summaryRec = tuple<
  uint32 ip,
  int32 numFlows,
  uint64 sumPackets,
  uint64 sumBytes>;

rstring summaryRecToString(summaryRec rec){
  return "[ip:" + IPIntToString(rec.ip) + ", numFlows:" + (rstring)rec.numFlows + ", sumPackets:" + (rstring)rec.sumPackets + 
```
void printFlowRec(flowRec rec){
    printString("[srcIP:" + IPIntToString(rec.sourceIP));
    printString("", srcP:" + (rstring)rec.sourcePort);
    printString("", destIP:" + IPIntToString(rec.destIP));
    printString("", destP:" + (rstring)rec.destPort);
    printString("", pkts:" + (rstring)rec.packets);
    printStringLn("", bytes:" + (rstring)rec.bytes + "]");
}

composite NetSummary {
    graph
        stream<flowRec> GenFlow = TCPSource(){
            param
                role : client;
                address : "streamtraining";
                port : "5555";
        }

        stream<summaryRec> SourceAgg = Aggregate(GenFlow){
            window
                GenFlow : tumbling, time(30.0);
            param
                groupBy : sourceIP;
            output
                SourceAgg : ip = Any(sourceIP),
                numFlows = Count(),
                sumPackets = Sum(packets),
                sumBytes = Sum(bytes);
        }

        stream<SourceAgg> SourceAggSorted = Sort(SourceAgg){
            window
                SourceAgg : tumbling, punct();
            param
                sortBy : sumBytes;
                order : descending;
        }

        stream<SourceAgg> SourceAggTopN = Custom(SourceAggSorted){
            logic
                state : {
                    int32 topN = 5;
                    mutable int32 i = 0;
                }
            onTuple SourceAggSorted : {
                if(i < topN){
                    submit(SourceAggSorted, SourceAggTopN);
                    i ++;
                }
            }
            onPunct SourceAggSorted : {
                i = 0;
                submit(Sys.WindowMarker, SourceAggTopN);
            }
        }
4.10 Lab 4 Discussion Topic

Rather than using a Custom operator to identify the first 5 summary tuples, here is a solution that uses the Filter operator:

```sasl
(stream<SourceAgg> Top5Sources = Filter(SourceAggSorted) {  
    logic  
        state : {  
            int32 topN = 5;  
            mutable int32 i = 0;  
        }  
        onPunct SourceAggSorted : {  
            i=0;  
        }  
        param  
            filter : i++ < topN;  
    }
```