Connector User's Manual

Jose Melchor
Computing and Software Systems, University of Washington, Bothell

Table of Contents

About Connector

Connector is a set of tools designed to help programs retrieve remote data. The main objective of Connector is to help grid/cloud applications obtain data residing outside of the grid/cloud. In order to achieve this, Connector hides all channels of communication from the point of view of the developer. Instead, developers feel like data is local to the running application. The current version of Connector is able to provide data from FTP, SFTP, and HTTP servers.

Connector also provides redirection of data to users' local machines. To achieve this, a GUI and a Web Server is provided so that remote applications may redirect output, input, error, and graphics to users.

Another key aspect of the Connector project is to provide data analysis applications running in the cloud with real-time data. Since sensor networks reside outside of the cloud, a simple way to obtain data was necessary. Sensor Server allows a sensor network to communicate with cloud running applications.

Connector Toolkit Components

Connector API

This is a collection of classes that provides developers with the necessary tools to manage I/O and graphics. Java-like classes are provided such as FileInputStream, FileOutputStream, Frame, and Graphics. This API needs a configuration file to map all files with their remote locations. To do this, a file map must be declared in the directory of the application using this API.

File Map

The file map uses a quadratic notation specifying the name of the file, URL, retrieving interval, and type of read (*name, URL, [interval, extract]*). The following explains these four fields:

1. *Name* of the file used in the program. This is also the name of the file at the remote location.
2. *URL* of the remote location where the file is stored. This field must also include the user's account name and password. Also, the path of the file must be declared.
3. *Interval* is the amount in minutes a program will access a web site to retrieve data. This field is only for data read from http servers.
4. *Extract* tells the program that only text data should be retrieved from web sites. If this field is missing form http server lines, then the entire html code will be retrieved.

The file map also needs two other fields to allow programs know where a GUI or Web Server is and which port to use. To do this, the first two lines of the file map must include the IP address of the machine running the GUI/Web server and the port to be used for the connection. The first two lines should start with a # sign followed by the –i and –p options.

The following is an example of how to construct a file map:

```
# -i        ip_address
# -p        port
file1.txt   sftp://account:password@dante.u.washington.edu/SensorGrid
sensor1     ftp://account:password@hercules.uwb.edu:55555
master      ftp://account:password@hercules.uwb.edu:55555    1
file2.txt   ftp://account:password@ftp.tripod.com/SensorGrid/Temperatures
file3.txt   http://www.weather.com/today/Bothell+WA+98011    5    extract
```

Notice that for Sensor server reads, a port should be provided. This is the default port used by Sensor server. Interval is allowed for Sensor server reads.

Something to keep in mind is the way a URL is written. The URL should contain the path of the file location. However, the path should include only the file's directory but not the name of the file itself. The file name is considered to be the first token of the line, so is not needed to restate its name. An error would occur if the file name were to be included in the URL.

Connector API's Classes

This section is an overview of the classes and functions provided by the Connector API.

Connector Class – Allows a remote/local application to use standard input, output, and error using a Connector-GUI. The Connector object first connects to a Connector Daemon in order to transfer data from and to the Connector GUI. Once a connection is established, interaction may begin.

Constructor Summary	
Connector() /** This constructor has not been fully implemented. No \tmp file defined **/ Default Constructor - Instantiates a ConnectorDaemon object using the local file "tmp\Connector.config" to set up the arguments required when booting ConnectorDaemon.	
Connector(String filename) Filename Constructor - Instantiates a ConnectorDaemon object using the file defined as a parameter in this constructor.	
Connector(String filename, boolean needDaemonOnly, int port) Boolean Constructor - Instantiates a ConnectorDaemon object using the file defined as a parameter. Such file is used to set up the arguments required when booting a ConnectorDaemon. A connection to a Connector-GUI is dependent on the boolean argument. If the boolean argument is true, a Connector-GUI connection is established (this is just like using a filename constructor). However, if the boolean argument is false, this Connector object will only instantiate a ConnectorDaemon and the out(), in(), and err() funtions will be disabled. This constructor is necessary for applications that do not require a Connector-GUI in order to function.	
Connector(String filename, int port) Port Constructor -Instantiates a ConnectorDaemon object using the file defined as a parameter in this constructor.	

Method Summary	
void **close**()	Closes all data streams used and terminates the ConnectorPanel and the ConnectorDaemon associated with the object.
void **err**(String)	Sends an error string to a ConnectorPanel.
protected void **finalize**()	Ensures that the close method of this Connector object is called when there are no more references to it.
String **in**()	Retrieves a string from a ConnectorPanel.
void **out**(String)	Sends a string to a ConnetorPanel
void **run**()	Reads the filename defined in the constructor and boots a connectorDaemon.

ConnectorDaemon Class - Daemon process that manages I/O operations from applications. A ConnectorDaemon starts automatically once a Connector object is instantiated. Every I/O request made is handled in a separate thread. ConnectorDaemon accepts requests from Connector, FileInputStream, FileOutputStream, Frame, and Graphics classes. So in a sense, this class is a helper class in charge of setting up all data channel connections and data retrievals.

FileInputStream Class- With this class a user application can open, read, and close remote files. FileInputStream establishes a socket connection to the local Connector daemon process within its constructor, sends "FileInputStream" and the remote file name from which to read, reads data through the socket from Connector, and forward the data to its application.

Constructor Summary	
FileInputStream(String name) Main constructor. The filename is the remote file needed to retrieve.	
FileInputStream(String name, int port) Initializes communication with the Connector daemon through a specified port number.	

Method Summary	
int **available**()	Returns an estimate of the number of remaining bytes that can be read (or skipped over) from this input stream without blocking by the next invocation of a method for this input stream.
void **close**()	Closes this file input stream and releases any system resources associated with the stream.
protected void **finalize**()	Ensures that the close method of this file input stream is called when there are no more references to it.
int **read**()	Reads a byte of data from this input stream.
int **read**(byte[] b)	Reads up to b.length bytes of data from this input stream into an array of bytes.
int **read**(byte[]b, int off, int len)	Reads up to len bytes of data from this input stream into an array of bytes.
long **skip**(long n)	Skips over and discards n bytes of data from the input stream.

FileOutputStream Class - With this class a user application can open, write, and close a remote file. FileOutputStream establishes a socket connection to the local ConnectorDaemon process within its constructor, sends "FileOutputStream" and the remote file name from which to write, and writes data through the socket from Connector. No acknowledgment is received if data was successfully stored at the remote server.

Constructor Summary	
FileOutputStream(String name) Main constructor. The name of the remote file is passed as the parameter.	
FileOutputStream(String name, int port) Initializes communication with Connector Daemon through an specified port number.	

Method Summary	
void **close**()	Closes this file output stream and releases any system resources associated with this stream.
protected void **finalize**()	Cleans up the connection to the file, and ensures that the close method of this file output stream is called when there are no more references to this stream.
void **write**(byte[] b)	Writes b.length bytes from the specified byte array to this file output stream.
void **write**(byte[] b, int off, int len)	Writes len bytes from the specified byte array starting at offset off to this file output stream.
void **write**(int b)	Writes the specified byte to this file output stream and overwrites the remote file specified in the constructor.

DataPacket class - This is a wrapper class intended to work mainly within FileInputStream and FileOutputStream classes. This class provides the object required in the Connector Daemon to send either actual byte data or exceptions to the extended Connector's FileInputStream or FileOutputStream classes.

Frame class – This class allows remote applications to draw graphics local and/or where Connector-GUI is running.

Constructor Summary	
Frame(String title) Construct a new instance of a Frame at the GUI's location. With this constructor the application will work as a X client and the GUI as an X server.	
Frame(String format, int seconds, String fileLoc) Constructs a new instance of a Frame at the location defined by the user. With this constructor an image will be sent to the user as often as the user specifies. The image is sent using a format provided by the user (jpeg, tiff, or png). Every time an image is sent to the GUI, the GUI pops a frame to display it if necessary.	

Method Summary	
void close()	Closes all connections to ConnectorDaemon
Graphics getGraphics()	Creates a graphics context for this frame.
java.awt.Insets getInsets()	Determines the insets of this container, which indicate the size of the container's border.
void setLocation(int x, int y)	Moves frame to a new location.
void setResizable(boolean resizable)	Sets whether this frame is resizable by the user.
void setSize(int width, int height)	Resizes this component so that it has width width and height height.
void setVisible(boolean b)	Shows or hides this Window depending on the value of parameter b.

Graphics Class – This class allows to draw inside a Connector's Frame.

Constructor Summary	
Graphics() Constructor allowing Graphics as an X client.	
Graphics(BufferedImage image) Constructor allowing Graphics to display on request by the user.	

Method Summary	
void fill3DRect(int x, int y, int width, int height, boolean, raised)	Paints a 3-D highlighted rectangle filled with the current color.
void fillRect(int x, int y, int width, int height)	Fills the specified rectangle.
void setColor(Color c)	Sets this graphics context's current color to the specified color.

Connector-GUI

Connector-GUI's functionality resides in the ConnectorPanel.java. This is the front-end application required to reroute standard input, output, and error to a machine local to the user. This ConnectorPanel connects to ta ConnectorDaemon to manage I/O operations launched by Connector, FileInputStream, and FileOutputStream objects. Also, ConnectorPanel recognizes messages from the Frame and Graphics classes so that it can render graphics local/remote in relation to the user. In addition to rerouting standard input, output, error and graphics from a remote machine to a local user machine, ConnectorPanel class can also interact with VikingX sensors through the FTP Sensor Server.

Connector-Web Server

This part of the project is still a prototype and very limited in terms of design and functionality. Using the Netbeans IDE, servlets have been implemented to be able to connect using TCP sockets to Connector Daemons. Since http protocol does not keep permanent connections, servlets are used to maintain Java sockets bound to Connector daemons in the background. Whenever a user retrieves data by pushing a button in the web browser, then the servlet grabs data from the TCP socket.

At this moment output, input, and error from the applications happens without trouble. Redirecting graphics work by sending images and repacking them in the web server. However, X client capabilities are not possible. Also, functionality to manage Sensor Server has not been implemented.

Sensor Server

Sensor Server was implemented in order to read information from VikingX sensors to remote applications. This is an FTP server that allows applications to retrieve sensor information as if data was contained in files. In order for this to work, remote applications need to use Connector.FileInputStream.

At the VikingX directory another two files must exist in order for Sensor Server to work properly. Those files are:

files.txt: This file defines the IP and port of the GUI (ConnectorPanel) to which the Sensor Server will be connected (if necessary) and, also, defines the URL of the remote file containing "sensor-map".

users.txt: This file includes all the allowed user names and their passwords. Each line must have one user name and a password only. Example:

Jmelchor 54321
dslab 12345

Sensor map

To manage sensors, Sensor Server uses a "sensor map" in order to keep a list of sensor names and sensor IP/Mac addresses available. Such list is not stored local to the Sensor Server but it is accessed from a remote FTP server. The sensor map is retrieved using Connector.FileInputStream and updated using Connector.FileOutputStream. For that reason, "files.txt" located in the VikingX directory must include the URL where the actual sensor map file is stored.

The sensor file map has the following format:

add [IP/Mac address] [sensor name]

Example:

```
add  192.168.15.3  sensor1
add  0x082be4      sensor2
add  all           master
```

Using master as a sensor name is a special case. When master is used in an application or when setting up handlers, it means that sensor information will come from all available sensors in the file map.

Design and Architecture

Any application using the **Connector API** needs to always declare a Connector object before declaring any other Connector API objects or calling any functions from them. The reason for this requirement is that a Connector object spawns a Connector Daemon that runs in the background. This daemon thread serves all Connector API objects by spawning thread for each operation performed.

When a Connector object is instantiated, first a Connector Daemon thread is spawned. Once a daemon thread is up and running, the connector object signals the daemon to start a thread that handles communications with a Connector-GUI. This thread is alive until the Connector object calls the close() function. The daemon thread also starts a thread that keeps listening for heartbeat messages from the Connector GUI. This thread ensures that GUI is able to figure out if an application moved/failed (see figure 1 to see the format of a program using the Connector API).

Figure 1 Simple application using Connector API

```
import Connector.*;       // Import Connector Package
import java.util.Scanner;

public class TemperatureRecording {
  public static void main ( String[] args ) {
    // Initialize a Connector daemon thread
    Connector System = new Connector("file.map");
    // forward the message to a remote thread
    System.out("Recording...");
    // Connect to an orchard sensor
    FileInputStream in = new FileInputStream("sensor1");
    Scanner input = new Scanner( in );
    // Connect to ftp.tripod.com
    FileOutputStream out=new FileOutputStream("file2");
    DataOutputStream.output=new DataOutputStream(out);
    while( input.hasNextLine() ){
      //Transfer data from the orchard sensor to ftp.tripod.com
      output.writeUFT( input.nextLine( ) );
    }
    System.close( );    // Terminate the Connector Daemon
} }
```

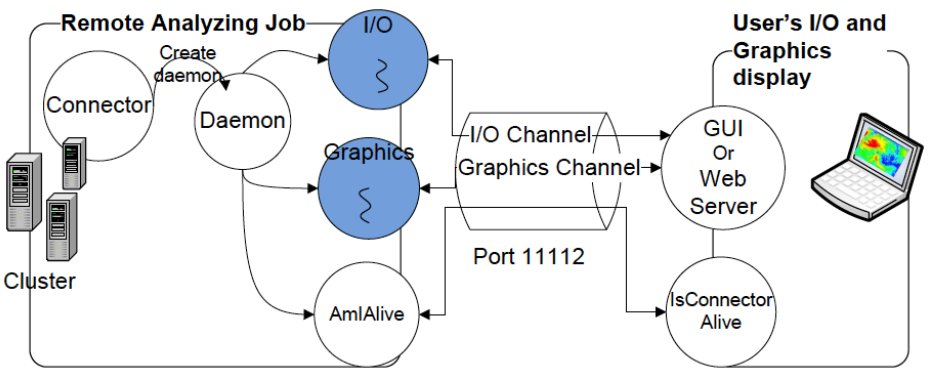
1) Start a daemon with the file map

2) Start using Connector API to I/O with GUI or perform File I/O for data analysis.

3) Close the connector

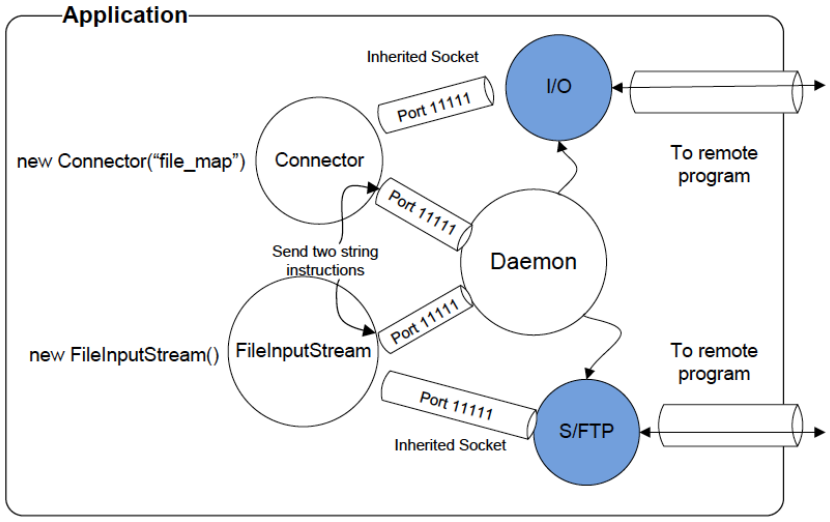
After the daemon thread and connections to GUI are established, then the user application can start the interaction with remote files. Figure 2 shows how threads would communicate with a GUI. Notice that the graphics thread is only to illustrate that such thread would have its own channel once created (however, no graphics thread is instantiated with the Connector object).

Figure 2 Connector bound to GUI



As mentioned before, Connector API objects use daemon threads to perform their jobs. In order to communicate with the daemon thread, Connector API objects use inner process communication through sockets. Each object communicates using port 11111 as default to send instructions to the running Connector daemon. The daemon thread needs two messages to be able to spawn an independent thread to handle the request of any API object. For example, FileInputStream first establishes a socket connection with the daemon thread and then sends two strings messages. One message is "FileInputStream" and the other is the name of the file (see figure 3). Some API objects only require one message for the daemon to spawn a new thread while others require more than two. The important thing is that no matter what, the daemon expects two messages coming from API objects.

Figure 3 Communicating with daemon

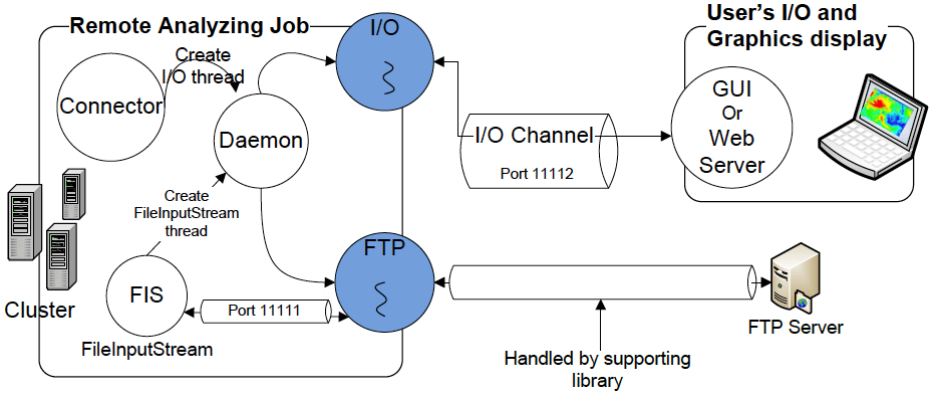


Support libraries are used to manage connections with FTP, SFTP, and HTTP servers. The libraries used are:

- For FTP, Apache Commons Net
- For SFTP, Jsch
- For HTTP, Htmlparser.

These libraries create independent connections to servers as necessary. In other words, connections are abstracted and transparent from the point of view of the connector daemon thread (see figure 4).

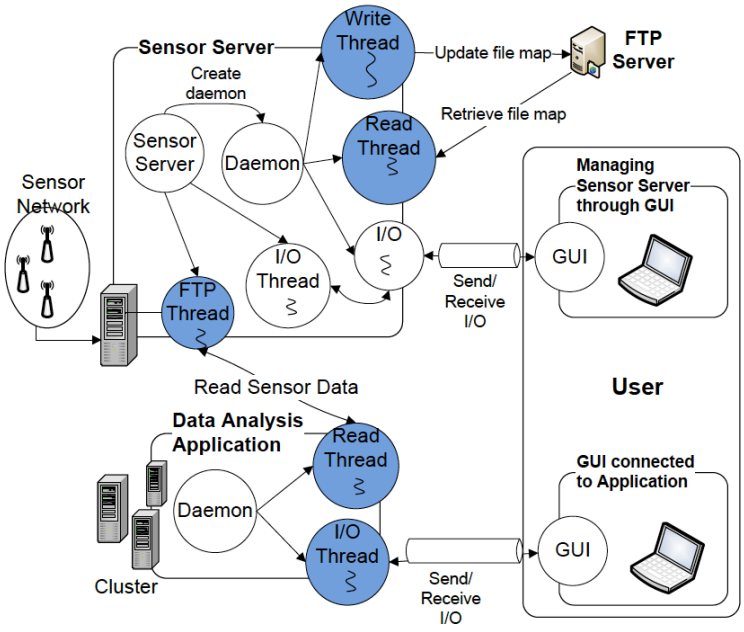
Figure 4 Connections using support libraries



Sensor Server is an FTP server. However, when replying to FTP clients, the server reads from the VikingX sensor network at the DSLab. The Hercules computing node at the DSLab contains the master sensor where all other sensors send information. Using a program we retrieve data from sensors and then send the data to FTP clients as if it was a file. At this moment the protocol of the sensor network is being reviewed and the Sensor Server will need to accommodate for any changes about the way data from sensors is obtained.

Sensors can be managed using a Connector-GUI. To achieve this, Sensor Server uses a Connector object to use input and output. Instructions are exchanged between the GUI and the server. Sensor Server recognizes instructions so that changes on sensors can be performed (more information on how to manage sensors can be found in the How to Run section). Figure 5 provides a view of how Sensor Server interacts with other components of the Connector system.

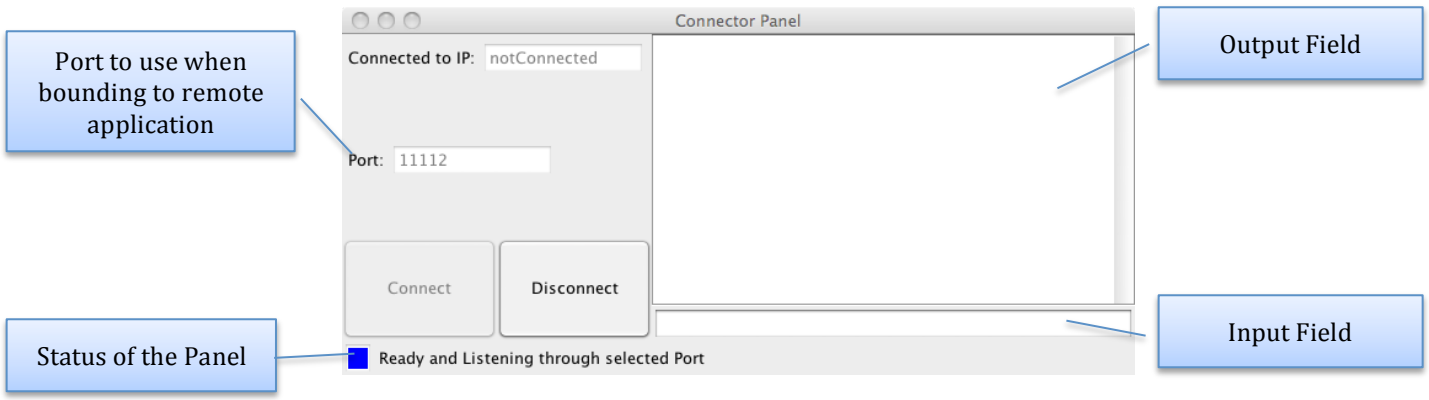
Figure 5 Visualization of Sensor Server



How to Run

Connector-GUI

To start the GUI, run the following command: $ **java ConnectorPanel**
Below is a snapshot of GUI's components.



Connector-API

Programs using the Connector API need to use the following commands while at the same directory as the application.

java –cp Connector.jar:. [*program-name*] [*arguments*]

The directory of the application needs to contain the "file map" for the connector object.

Sensor Server

Sensor Server allows a Connector GUI to interact directly with it. Such interaction makes possible the modification/addition of sensor names and sensor IPs. Also, Sensor Server can set conditions to monitor sensors based on commands defined from the GUI. These tasks can be achieved by following these steps:

1. Start a GUI using this command
   - **java ConnectorPanel**
2. Sensor Server connects to Connector GUI using port number 11113, so set the port field to 11113. Then press the Connect button.
3. Run Sensor Server (located in the VikingX directory) with a GUI option, like this:
   - **java -cp Connector.jar:. SensorServer -g**
4. Once a connection is established, then commands from the GUI can be sent to Sensor Server. Remember that the GUI's IP need to be set in the "file map" located at the VikingX directory in order for the connection to be successful.

The following is the structure of commands that can be used to alter sensors and place event handlers to monitor sensors:

```
- To add a sensor, type:          add    [ip] [sensorName]
- To delete a sensor, type:       delete [ip/sensorName]
- Event handler commands:         detect [sensorName] [logical operator][temperature][absolute/relative]
- Remove handler conditions:      detect [sensorName]
```

Examples:

```
add      192.168.15.3      sensor1
delete   192.168.15.3
delete   sensor2
detect   master <= 32.0    absolute
detect   sensor1 >= 0.2    relative
detect   sensor1
detect   master
```

Examples

Examples can be found at folder dslab's folder of Test programs (dslab/SensorGrid/Connector/TestPrograms). Most programs are very simple and follow the format stated in the "How to Run" section. The programs have been modified throughout development to test different cases. However, there are programs to test each Connector API object.

Known Issues

By far the main issue with Connector is the Web Server. As mentioned before, the web server functionality is at minimal stage, not to mention design of the web site. However, the framework is set up and the construction should not be too complicated. Also, the skeleton of the web server should follow the format of the GUI to ease construction and connectivity with Connector daemon.

There are a few issues that need to be fully tested. One of those issues is the handling of exceptions at the daemon/application side. At this point connector wraps exceptions within a DataPacket object and sends it to the GUI for display, but I have not tested this.

Another issue is that sometimes when an application is killed, the GUI goes back to listening state (as it is suppose to do) but then when another application starts the GUI is not able awake and display the application's requests unless the GUI is disconnected and reconnected again.

Surely more bugs will pop up as the Connector system keeps growing. However, the author is willing provide support whenever more information is desired.