Available on CMS information server **CMS IN-2008/005**

8 February 2008

GUTS - A Generalized Unilateral Transfer Server

Stefan Spanier and Matt Hollingsworth

Abstract

GUTS is a framework for transferring data from one node to another in a simple, fast, and secure manner. It is based on JAVA. A sender can be easily adopted to front-end data acquisition and send data objects to several consumer clients or clients that simply broadcast to a network of other clients. The transfer is only in one direction; there is very minimal communication between the server and the client. This allows for both security and for speed, and limits the scope of required coding. The package has been tested for reliability and security and is in use for the beam radiation monitoring in CMS.

1 Introduction

GUTS is a framework for transferring data from one node to another in a simple, fast, and secure manner. It is designed specifically to solve the problem of getting data from a data producer to a data consumer. The transfer is only one way; there is very minimal communication between the server and the client. This allows for both security and for speed, but limits the scope of the library. The strengths of GUTS lie in its simplicity of use and its speed. It is, therefore, very good at getting data from an instrument to another computer that is interested in knowing the status or current data of that instrument–it is tailored to monitoring.

The package was designed out of the needs for the beam radiation monitoring for the CMS detector [1], where the data acquisition is performed in the LHC technical network. A stringent requirement by CMS to the beam monitoring is that data are transferred redundantly into live displays with the same update rate of 1 Hz used by the LHC and with an interrupt of no longer than 2 seconds.

The GUTS Java based package allows a direct feed of Java data objects from the front-end DAQ crate PC across different networks and different platforms of multiple clients at the same front-end clock rate, with a buffer scheme and timeout. The default client number is set to 20. It practically depends only on the processor, the frequency of transfer and the available memory how the scheme can be configured. The present version is based on Java 1.6, and utilizes the java.io.* object input/output API [2].

2 General Description

2.1 Server Client Scheme

The server is controlled by the primary source - in case of BRM by the DAQ class within the CMW [3] framework. As soon as data are handed to the server it packages them into the user defined data object [1]. It then iterates over a list of local listeners such as a file logger and a list of connected clients. Clients can be started independently of the server and will keep listening until explicitly shut down. The reconnect trial rate is 1Hz chosen as user default, but can be set substantially higher or lower. A connected client is practically a mailbox instantiated on demand of a client at the server side: the server keeps a buffer (by default 10 packets deep) and a buffer counter. At the very next acquisition cycle the server drops the packet for the client and increments the counter. (The buffer is in fact physically present on the server side, but also on the client side and administered by TCP/IP). The client is responsible to pick up the mail and send a zero back (the only information that is send toward the server and accepted there) to acknowledge the receipt. On receipt the packet is removed from the buffer.

The frequency at which the client checks the presence of the packet is substantially higher than the feed rate (implemented in the Java i/o library). For the implementation scheme it is important that the frequency at which the client reads the packets is higher than the rate at which the packets are dropped into the mailbox. The 10 deep packet buffer will smooth fluctuations and connection drops. When the buffer counter reaches 10 it starts a timeout that by default is set to 3 seconds; if the client did not reconnect or demand packets within this time interval the client is kicked off from the list and needs to explicitly reconnect and open a new mailbox. Logging on the server side (with a local application) will ensure that the information is not lost, though. For local applications and further clients the primary client assumes the role of the server - the server class is practically reused to implement the client.

Notice, that since any real Java object can be sent using this scheme (including for example DIP [4] objects or CMW [3] objects) it can be used to bridge and probably speed up existing transfer schemes, or just introduce redundant lines for safety. A plot representing the components and an example of a more complex broadcasting scheme is displayed in Fig. 1.

2.2 Security

First of all, GUTS is protected by the standard security measures of a network connection through ports (firewall, iptable configuration - allow only registered IP, user etc.). The Java data objects are unique to the server and the client. The object if intercepted can only be interpreted through the data object class.

Furthermore, the transfer occurs only in one direction: if a client tries to write toward a previous stage client or server the connection is dropped (kicked) - only a zero as acknowledgment of a packet reception is accepted. Data

[1] If the primary data is a physical Java object it can be just pushed through.

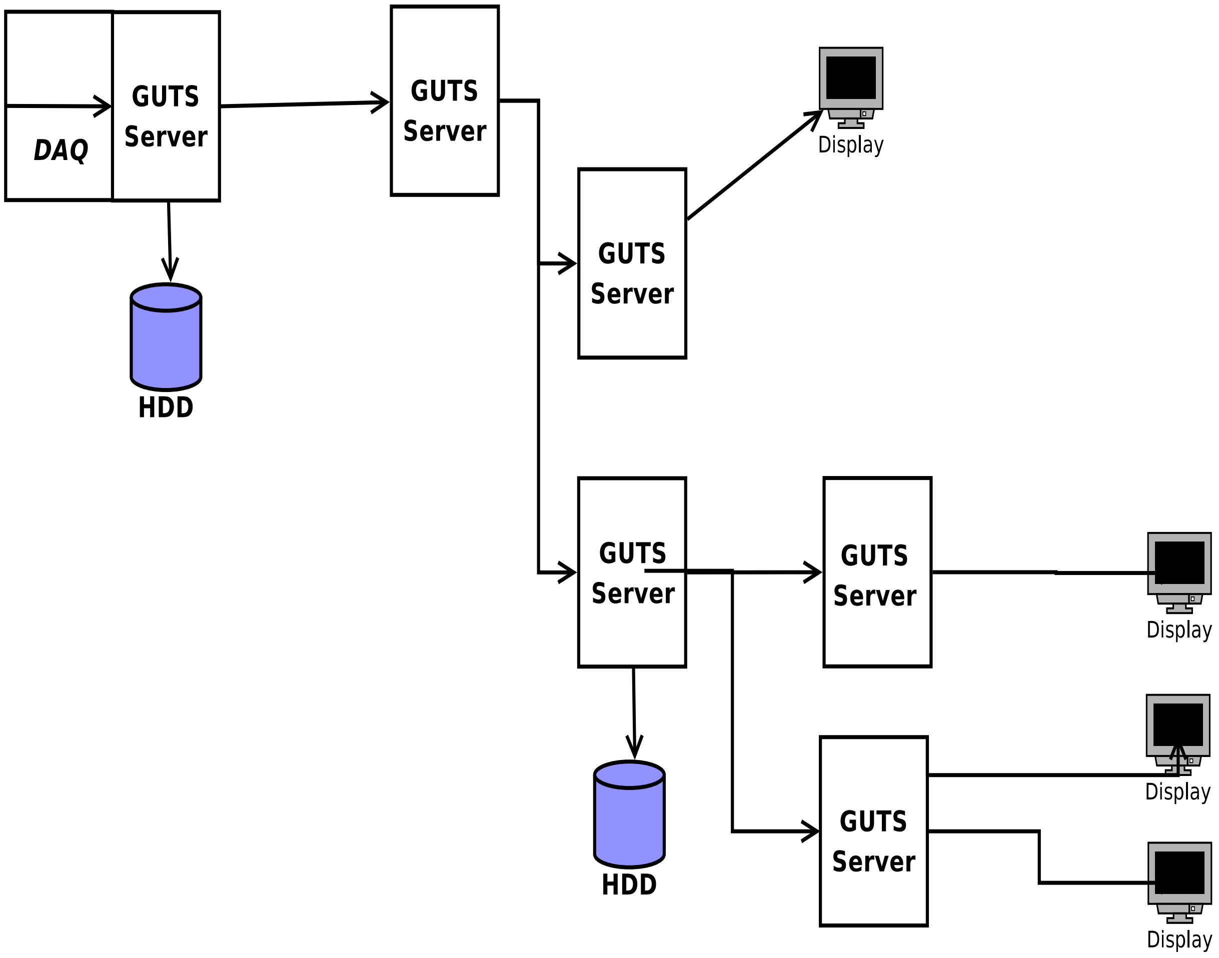

Figure 1: *Symbolic description of the GUTS server and client, left, and a hypothetical scheme linking several clients to a front-end server.*

Event	Server reaction	client reaction
Frontend crate goes offline	send data packet with 'crate offline' flag	alarm if offline longer than 2s
Frontend publisher goes offline	same as above	same as above
Server PC offline	none	raises alarm after 2s
corrupt data in server	none	none
corrupt data at client	md5 checksum	md5 checksum error output
client execution stops	after 2nd write kick off	user has to restart it
server execution stops	needs restart	sounds alarm after 2s

Table 1: *Evaluation of different exceptional situations to ensure robustness of the server client scheme.*

integrity is checked by creating and evaluating a md5 checksum sent with the packet by each server or serving client. The client is responsible for evaluating and taking action (an exception is thrown). In case of a checksum violation by default the client generates a log message indicating a checksum problem. This means that a local client application like a display will probably set data values explicitly to exceptional values (e.g. -99 if positive integer ADC values are expected) on suspect of corruption to indicate the problem and a shifter is informed by the log-message.

2.3 Reliability

The code is written in Java and has been tested over several months (since Fall 2006). Stress tests have been performed (see below). The defaults such as number of clients per server are very conservative choices that have been tested extensively. The GUTS scheme provides error messaging and error handling for several situations in the clients: If the server is shutdown properly or the server terminates a client connection, the client gets a last packet beforehand with the message 'you got kicked'. Upon crash of the server or CTRL-C (improper shutdown) the client will generate an io exception and generate the message 'connection dropped' - it will try to reconnect every 1s (chosen as default). The same happens if a cable connection is broken. Front end errors are propagated: if the FE times out a message packet is sent by the server and displayed by the client: 'instrument timed out'; for an explicit disconnect the message is 'instrument disconnected'. In both cases the server will try to reconnect continuously. If a client did not pick its packet for 10 cycles the server (or client that assumes the server function) will send a 'missed packet' message ones to inform this client that from now on the box is not filled anymore. If the client acts and reads at least one packet, the buffer updated.

2.4 Reliability Tests

Several exceptional situations have been tested with a server and client scenario. They are listed in Table 1.

We identify a safe regime of operation as the result of several stress tests with a broad range of choices of parameter settings: the available RAM limits the product $N_{client} \times N_{buffer} \times S_{packet}$, with N_{client} the number of clients, N_{buffer} the buffer depth per client on the server, and S_{packet} the size of a data packet. Our conservative default recommendation is $N_{client} \leq 20$, $N_{buffer} = 10$, and packet size not exceeding 2 MByte. Other choices are possible but were not established here - e.g. we were able to send 150 packets of 2 MByte size per second, but for small sized packets (2 bytes) the minimum spacing between packets should stay above 50 ms to keep the receiver frequency (given by the network latency and the client acknowledgment time) above the server sending frequency. Up to 1000 clients have been connected to a single server or serving client, and up to 7 chained clients in a GUTS network have been exercised.

2.5 Client Installation

A monitor client within the GUTS framework connecting to an existing front-end server (such as the GUTS server running in the LHC environment connecting to BRM detectors within CMW) is distributed as a .jar collection. The computer must run on SL3 or newer or Windows XP, 2003 or newer. To run the software requires

- Java 6

To run it on a local machine anywhere in the Internet requires the knowledge of the ip-address and the open port on the server machine or on the machine that mirrors the original port.

For development work of and within the GUTS framework the package is distributed from a CVS repository or as binary jar distribution in form of a .tar. Additional dependencies are:

- log4j [5]

The project is maintained within Eclipse [6]. Tools to build the software are available from there. For the production of the same .jar that is distributed as a binary release an Eclipse plugin called FatJar (http://fjep.sourceforge.net/) is needed. The plugin creates one jar out of multiple jars to reduce dependencies.

Without the use of Eclipse three pieces of software are needed to build GUTS from source:

- Apache Ant (http://ant.apache.org/),
- “Headless” Buckminster (http://www.eclipse.org/buckminster/), and
- One-Jar (http://one-jar.sourceforge.net/).

The download, installation, and integration of all of these external dependencies are discussed in http://hep.phys.utk.edu/wiki/GUTS.

2.6 Documentation

In addition to this manual, there are examples, tutorials, and other resources available at http://hep.phys.utk.edu/wiki/GUTS. Please check here first, as the Wiki will always have updated links to all the relevant resources.

3 Coding

3.1 Example Server

Let’s begin with an example. This example establishes a server that produces (generates) random numbers, packages them into a class, and sends them to any connected client. The client then receives this data and prints it to the console.

```
import edu.utk.phys.guts.DataServer;
import edu.utk.phys.guts.DataClient;
import edu.utk.phys.guts.DataPacket;
import edu.utk.phys.guts.exceptions.*;
import edu.utk.phys.guts.event.DataEventGenerator;

import java.util.Date;
import java.util.Random;
import java.io.IOException;

public class SimplestExample {

   /**
    * A basic class for containing our data
    * This is step #1!
    */
   private static class OurData implements java.io.Serializable {
      private int i = 0;

      OurData() {
         i = new Random().nextInt();
      }

      public String toString() {
         String str = new Integer(this.i).toString();
         return str;
      }
   }
```

```
/**
 * A simple class for generating data that is sent to connected clients.
 *
 * This is step #2!
 * @author matt
 *
 */
private static class SimpleEventGenerator extends DataEventGenerator
      implements Runnable {
   private boolean isStarted;

   @Override
   public void run() {

      // Sends a date to the client every second
      for (int i = 0; i < 10; i++) {
         OurData ourData = new OurData();
         this.fireEvent(new DataPacket(ourData));
         try {
            Thread.sleep(1000);
         } catch (InterruptedException e) {
            // do nothing
         }
      }

      // When it's done, stop the server
      this.getDataServer().stop();

   }

   /**
    * Starts the server by forking a new thread to throw the events
    */
   public void start() {
      if (!isStarted) {
         Thread t = new Thread(this);
         t.start();
      }
   }

   /**
    * The stop function does nothing since the data generation will stop
    * itself (although we could easily control this with a flag in the
    * thread).
    */
   public void stop() {
      // No stopping this train
   }

   public boolean isStarted() {
      return isStarted;
   }
}

public static void main(String[] args) {

   // Create the data server that will listen on port 4000
   DataServer server = new DataServer(4000); //Step 3.0
   server.setName("Simple␣Example␣Server");

   // Create the object that will generate events for the server
   SimpleEventGenerator generator = new SimpleEventGenerator(); //Step 3.66
```

```
        // Register the event generator
        server.registerDataEventGenerator(generator); //Step #3 complete!

        // Create the object that will get the data from the server
        DataClient client = new DataClient("localhost", 4000); //Step #4

        // Start data generation and data client
        try {
            //Step #5
            server.start();
            client.connect();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (GutsException e) {
            // do nothing
        }

        // Get the data from the server and print it to the console
        while (true) {
            try {
                System.out.println(client.getDataPacket());
            } catch (IOException e) {
                break;
            } catch (ServerException e) {
                // do nothing
            } catch (MissedPacketException e) {
                // do nothing
            }
        }

    }

}
```

For more examples such as this, see the examples/ directory of the GUTS source distribution.

3.2 Example Explained

We will now walk through the example step-by-step, focusing first on the server logic, and then explain how the client works.

In the example, we did five things:

1. Create a class to represent the data that we want the clients to receive
2. Create a class that produces our data (a Date object). This class must inherit from DataEventGenerator.
3. Instantiate a DataServer and associate it with our data production class
4. Instantiate a DataClient to get data from our server
5. Start both the server and the client

The implementation-specific code for a user of the GUTS package will fall primarily into steps 1 and 2. It is here that you are gathering and packaging the data that you wish to send to the clients of your server–instrument data, the time, a status report, etc. The GUTS package sends serialized POJO's (Plain Old Java Objects that only need to implement java.io.Serializable), so all you must do is make a class to contain the data, fill up an instance of that class, and call fireEvent(DataPacket) your class that extends DataEventGenerator. Since the DataEventGenerator is what client programmers will be working with the most, I will give a full-fledged discussion of it and the DataPacket in the following section.

After the data-containing POJO and the DataEventGenerator that packages data into said POJO are written, the GUTS framework deals with all of the logic necessary to send the data to the clients. All the client programmer must do to send the data that the DataEventGenerator is producing is

1. Instantiate a DataServer
2. Call setName on the DataServer to give it a name. While this step is not necessary, it is recommended because it makes some management features available that will be discussed later on.
3. Register an instance of your DataEventGenerator with the new DataServer
4. Start the server (by calling start() on the DataServer instance)

When all these steps are complete, clients may connect to the interface(s) and port that you specified during the construction of the DataServer and receive the data produced by your DataEventGenerator.

Receiving data requires that the class that is being sent by the DataServer to be available in the client's CLASSPATH (otherwise, you will get a ClassNotFoundException), and to :

1. Instantiate a DataClient
2. Provide the DataClient instance with the hostname and port of the GUTS server to which you are connecting 2)
3. Connect the DataClient to a GUTS server using the DataClient.connect() method
4. Repeatedly call getDataPacket() method of the DataClient to receive each packet

The base features of the GUTS package are described in more detail below.

3.3 The DataEventGenerator

The DataEventGenerator is the class that produces and packages the data that you want to send to clients. In particular, it puts your POJO instance inside a DataPacket, and calls fireEvent(DataPacket) with the new DataPacket. The DataPacket serves as a container for your data; it has data such as the date when the server sent the data, the name of the server that sent it, etc.

The DataEventGenerator supports two types of listeners:

- Network clients connected to the DataServer that is associated with the DataEventGenerator
- A class that implements the edu.utk.phys.guts.event.DataEventListener interface

Events are dispersed among the listeners by calling one of the fireEvent() or fireException() overloads. These overloads are categorized into the functions that fire "good data" events, and the functions that fire "error" events. Error events simply mean that you form an ErrorPacket (a child class of DataPacket) as opposed to a normal DataPacket. ErrorPackets hold exception information, and are handled differently by the DataClients (see Section 3.5 for more information). fireException() is shorthand for fireEvent(ErrorPacket), where the ErrorPacket was constructed using the ErrorPacket(Throwable) constructor.

You associate a DataServer with a DataEventGenerator by calling:
DataServer.registerDataEventGenerator(DataEventGenerator) on a DataServer instance. You may only associate a single instance of a DataEventGenerator with one DataServer (although you may associate multiple DataEventGenerators with one DataServer). Once this is done, DataEventGenerator.fireEvent(DataPacket) will send the DataPacket to all clients that are connected to the associated DataServer.

Side note: This works internally by querying the associated DataServer for its list of clients using
DataServer.getClientList(), and calling sendData(DataPacket) on each of the clients in the list.

2) You may use either the constructor or the setHostname()/setPort() functions

You are not limited to sending your data only to network clients; you can register any class that implements DataEventListener as a listener to the DataPackets produced by your DataEventGenerator. For example, you may want to write the data that you are producing to disk as an ASCII file, in addition to sending the data to network clients. In order to use a class that implements DateEventListener, you must register the implementing class by using the registerDataEventListener(DataEventListener) method of the corresponding DataEventGenerator. The class implementing the DataEventListener interface will then receive the same DataPackets that any network client that is connected to the associated DataServer are receiving (except it is wrapped up in a DataEvent).

You may use your DataEventGenrator exclusively for network clients, exclusively for local DataEventListeners, or a mixture of both; it is up to you. See the last section in this chapter for a number of examples on how to use these features.

3.4 DataEventGenerator Implementation Notes

You should always make sure that your DataEventGenerator's start(), stop(), and isStarted() functions do not block! In the the start function of example 2.1's DataEventGenerator, a thread was forked to produce the DataPackets. By forking a thread, we allow the function to return immediately. This must be done due to the way that the DataServer is implemented. The DataServer maintains a list of all associated DataEventGenerators. Whenever DataServer.start() is called, it iterates through each of its associated DataEventGenerators and calls start() on each one. Hence, the DataEventGenerators must be able to execute independently of each other; if one start() function blocks while the DataServer is iterating over its list, the other DataEventGenerators will never be started.

3.5 The DataClient

The basic way to get data from a GUTS server is by using a DataClient. Here is an example of how to use a DataClient:

```
import edu.utk.phys.guts.DataPacket;
import edu.utk.phys.guts.DataClient;
import java.util.date;

..
..

//Make a client that connects to ``myserver.domain.org'' on port 4000
DataClient client = new DataClient(``myserver.domain.org'', 4000);

try{
   client.connect(); // Connect the client
} catch(IOException e) {
   e.printStackTrace();
}

DataPacket data = null;
try{
data = client.getDataPacket(); // Get a data packet from the serve
} catch(IOException e) {
// The server isn't responding
} catch(ServerException e) {
//The server wants you know that a server-side exception occurred
} catch(MissedPacketException e) {
//You're going too slow, and you missed one or more data packets
}

Date timestamp = data.getTimestamp(); // The time in which the data was sent
String serverName = data.getServerName(); // The name of the server from which the
     packet was sent
Date dateData = data.getData(); // The data that the server sent us; in this case, a
     date

client.disconnect(); //Disconnects the client
```

A few key notes:

Don't wait for a long time to get a data packet from a connected DataClient. You must ensure that $f_c > f_s$, i.e., you must receive data faster than the server produces data. If you fail to do so, the server will eventually kick you. In particular, you must be able to call DataClient.getDataPacket() with a higher frequency than the server is calling DataEventGenerator.fireEvent().

Don't ignore an IOException in getDataPacket(). If DataClient.getDataPacket() throws an IOException, it should be interpreted as a critical error. You will want to either:

1. Terminate the program
2. Disconnect/Reconnect the client

A ServerException will make your data that you were expecting stay null. Even though it's fine to ignore a ServerException, make sure that you have logic in place to deal with the fact that the data you were acquiring is not there. In our previous DataClient example, if the server had sent us a ServerException as the data packet that we received, we would have gotten a NullPointerException when we called data.* because data would have been null. You could, of course, prevent this by putting all your logic in the try block (that's up to your coding style).

That's it for the DataClient. There are large number of examples available in the examples/ directory of the distribution if any more information is necessary.

3.6 The DataSubscriber

The DataSubscriber is an alternative to using the DataClient. It provides a subscription model for accessing the data from a GUTS server; every time you receive a DataPacket from the server, it provides a callback class with the data. Here is an example of using the DataSubscriber:

```
import java.io.IOException;

import edu.utk.phys.guts.DataSubscriber;
import edu.utk.phys.guts.event.DataEvent;
import edu.utk.phys.guts.event.DataSubscriptionListener;
import edu.utk.phys.guts.event.ErrorEvent;
import edu.utk.phys.guts.exceptions.ServerException;

public class DateSubscriptionClient {

   public static class DateSubscriptionListener implements
         DataSubscriptionListener {

      @Override
      public void dataPacketReceived(DataEvent event) {
         //Receive the data and print it to the console
         System.out.println("Data Packet received--here is the data: ");
         System.out.println(event.getDataPacket().getData());
      }

      @Override
      public void dataVerificationExceptionReceived(DataEvent event) {
         //not thrown, unless you configure the subscriber another way (see docs)

         System.err.println(event.getDataPacket());
      }

      @Override
      public void ioExceptionThrown(DataSubscriber subscriber, IOException e) {
         System.err.println("Subscriber threw an IO Exception. "
               + "Here is the message:");
         System.err.println(e.getMessage());
         System.err.println(e);
```

```
        }

        @Override
        public void missedPacketExceptionReceived(ErrorEvent event) {
            System.err.println("You missed a packet");

        }

        @Override
        public void serverExceptionReceived(ErrorEvent event, ServerException e) {
            System.err.println("The server said: " + e.getMessage());
        }

    }

    /**
     * @param args
     */
    public static void main(String[] args) {

        //create the subscriber
        DataSubscriber subscriber = new DataSubscriber("localhost", 4445);

        // register the listener
        subscriber.registerDataEventListener(new DateSubscriptionListener());

        try {
            //begin receiving data
            subscriber.startSubscription();
        } catch (IOException e) {
            e.printStackTrace();
        }

        System.out.println("Press enter to exit.");
        try {
            //stop receiving data whenever the user presses enter
            System.in.read();
            subscriber.stopSubscription();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

}
```

3.7 Management

There are two basic options for management: the GutsManager and the ConsoleManager. The GutsManager provides an API for managing the servers; it's meant to be called from a higher level (for example, JPype/Python or Jython script). The console manager allows you to manage a GUTS server from the console, using a command prompt that is shown the server starts up. Both of these interfaces perform actions such as starting/stopping the server, kicking clients, etc.

GutsManager The GutsManager makes use of the names of the servers in order to control them. You first register a server with the GutsManager, using the function registerServer(DataServer). The server that you register must have a *unique* name, for the logic in the GutsManager is of a static nature, and the name is mapped to the server instance that you wish to control. All of the management functions accept the server name and whatever other arguments are necessary to carry out the command. Most everything may be dong using strings [3]. For example, to start/stop a server named "My Server", you would do the following:

[3] This is meant to make it easy for web interfaces, consoles, etc. to control the servers

```
GutsManager.registerServer(myServer); //if you haven't already done this
GutsManager.startServer("My Server");
GutsManager.stopServer("My Server");
```

Or, if you wish to kick a client:

```
GutsManager.kickClient("My Server", "127.0.0.1");//kicks all local clients

GutsManager.kickClient("My Server", ".*");//Kicks all clients
```

ConsoleManager

To use the ConsoleManager, you first instantiate it and then connect it to io streams that will provide the command line input and output. Here is an example:

```
edu.utk.phys.guts.management.ConsoleManager manager =
   new edu.utk.phys.guts.management.ConsoleManager(relay,
   new BufferedReader(new InputStreamReader(System.in)),
   new PrintWriter(System.out), new PrintWriter(System.err));

while(true) {
   manager.printPrompt();
   try {
      manager.parseInput();
   } catch (IOException e) {
      e.printStackTrace();
   } catch (CommandException e) {
      System.out.println(e.getMessage());
      if(e.getSource()!=null) {
         e.getSource().printHelp(new PrintWriter(System.out));
      }
   }
}
```

The biggest downside to the ConsoleManager is that you may only connect a console to one server; managing multiple servers isn't possible.

3.8 Exceptions

The exceptions for the GUTS package are defined in edu.utk.phys.guts.exceptions.$*$. These are all the exceptions that are thrown by the classes in GUTS (except for standard ones like java.io.IOException, etc).

GutsException

This exception is thrown whenever a function either does something unexpected, or you pass it an argument that does not conform to the expectations for a function. Also, occasionally, lower-level exceptions are wrapped into a GutsException (for example, an xml parsing exception would be wrapped into a GutsException).

MissedPacketException

The MissedPacketException can be thrown by the various clients for the GUTS protocol. This exception is sent by a GUTS server to the client whenever a client is going too slow, and missed a packet that was sent. Once one of these exceptions is thrown to the client, the client quits receiving packets until the client catches up to the packets that it's waiting on.

ServerException

The ServerException is also thrown by the various clients. It is a wrapper for any exception that happened server-side that the server wants the clients to know about. For example, if you received a TimeoutException from an instrument that the server is reading, you might want the clients to know that the instrument timed out. Thus, you call DataEventGenerator.fireException(e), where e is the exception that was thrown, and the clients receive the exception as a ServerException.

DataVerificationException

This exception can be thrown by the various GUTS clients. If you use the various data verification features available in the guts package, and the verification fails, then a DataVerificationException is thrown. For example, if you use DataPacket.getData(true), you could receive a DataVerificationException if the data is corrupt (or cannot verified).

4 Extra Features

4.1 The guts-extensions Package

There is a package called "guts-extensions" that provides a basic plugin framework for guts. It allows one to package all of their server logic into a jar file and provide clients an easy to create complex servers. A separate paper will be written about this package.

4.2 Python

It is possible to use GUTS from python with the use of an additional package called JPype[7]. By utilizing JPype and python, you may interface GUTS with packages such as ROOT (which has a python interface). Documentation for both the installation procedure and usage for JPype can also be found here [7].

5 Summary

The GUTS package provides a secure and reliable solution for the unilateral transport of data. It's lightweight, flexible nature and minimal external dependencies make it ideal for quick inclusion into a pre-existing software infrastructure. For more information, visit the Wiki page at:
`http://hep.phys.utk.edu/BRM_Interface/index.php/GUTS.`

References

[1] Beam Radiation Monitoring section in 'The CMS experiment at the CERN LHC', submitted to the *Journal of Instrumentation JINST*, January 2008.

[2] Java 6, http://java.sun.com/.

[3] CERN controls middleware, http://proj-cmw.web.cern.ch/proj-cmw/.

[4] Data Interchange Protocol,
https://twiki.cern.ch/twiki/bin/view/Leade/WebHome and
http://itcofe.web.cern.ch/itcofe/Services/DIP/welcome.html.

[5] Apache logger project, http://logging.apache.org/log4j/1.2/index.html.

[6] http://www.eclipse.org/.

[7] Python package JPype: http://jpype.sourceforge.net.