

# **GramGraph Graphical User Interface**

by

**Christopher Scannell**

**March 12, 2002**

The Processing Graph Method Tool (PGMT) product is being released under the GNU General Public License Version 2, June 1991 and related documentation under the GNU Free Documentation License Version 1.1, March 2000.

<http://www.gnu.org/licenses/gpl.html>

This document is intended to provide an overview of the GramGraph Graphical User Interface (GUI) with emphasis on the relationship between the GUI and the GramGraph application Command Program process with which it communicates. It will talk about how the GUI is started, how it communicates bi-directionally with a separate process that is under the control of the Command Program, how it is stopped and the handshaking between processes involved with starting and stopping the graph from the GUI that is used to avoid race conditions.

This documentation will not describe how the GUI is used to change the parameters of the running Graph. It is recommended that the reader examine “Interfacing the Command Program to a PGM Graph” by Christopher Scannell in the PGM Documentation for more information on the relationship between the Command Program processes and the PGM Graph in a PGM application.

## **General**

The source code for the PGM application GramGraph is found in `$(PGM2_HOME)/apps/GramGraph/appsrc`. Portions of the code will be included in this documentation for clarification of the discussion, but it is recommended that the reader have the full source code available when reading this document. The source code discussed here can be found in three directories. The java source code is in the java directory, the socket related C source code is in the socket directory and the C++ source code related to the Command Program is in CP.

GramGraph is intended to be demonstration of a full-fledged functional application. It is a very different application from the PGM application Sort, particularly with regards to the amount and type of interaction the user has with the Graph. Sort is intended to be the simplest possible PGM application involving a running graph. Some of the characteristics of GramGraph can be best understood in contrast with the way Sort works. Sort starts a graph on all Graph Processes immediately by placing the call to `PGMT::startGraph` in the Command Program run method as the sole statement for all Graph processes. While the Graph is running there is a single Command Program process running. This process writes a fixed amount of data to the graph, stops the graph and exits. There is no GUI for Sort. GramGraph has a GUI in which the user presses a button to start and stop the Graph. Until the user presses the Start button for the first time, all processes are Command Program processes and `PGMT::startGraph` is not called anywhere. Once the Start button is pressed, the Graph Processes are under control of the running Graph until the user presses the Stop button. The GramGraph application is configured to use four separate Unix processes for four specialized tasks. All remaining processes (no inherent limit) become Graph processes when the graph is running. Data to feed the Graph is read from a data files, but when this data is exhausted it is reread from the beginning so that Graph can run continuously without stopping. The Graph can be stopped and started repeatedly any number of times by the user from the GUI without risk of conflicts between Graph processes trying to start while others are trying to stop.

The behavior of the Graph can be manipulated from the GUI by means of control tokens being sent to Graph Ports. The GUI has a Quit button to stop the application which can be pressed at any time, regardless of the state of the Graph, that stops the Graph (if it is running) and then sends messages to the Command Program processes to exit their run methods.

This functionality in GramGraph is accomplished by means of a separate java application receiving interactions from the user and presenting information to the user via the GUI that is communicating with one of the Command Program processes, called the FrontEnd, via a TCP socket. Some actions, such as starting and stopping the Graph, are atomic; they are carried out completely before other tasks are started. This is accomplished by means of acknowledgement messages which will be discussed at greater length later in this document.

### **Starting and Stopping the Java GUI**

The Java GUI is started as a separate Unix process by one of the Command Program processes, the FrontEnd process, by means of a system call. This call occurs in the CommandProgram::run method after differentiation of processes (see the “Interfacing the Command Program to a PGM Graph”).

```
void CommandProgram::run( int argc, char* argv[] ){

    /* snip */
    switch( rank() ){

        /* snip */
        case FRONTEND:
            /******
            /* The FrontEnd process is the control process for the Demograph GUI */
            /******
            {
                DBG( "FRONTEND is running on " << Machine::getCurrentProcessID() );
                int socketPort;
                if( argc > 2 ) {
                    socketPort = atoi( argv[2] );
                } else {
                    socketPort = DEFAULT_GUI_PORT;
                }
                char socketPortInt[10];
                sprintf( socketPortInt, "%d", socketPort );
                string str;
                DBG( "socketPort = " << socketPort );
                DBG( "DISPLAY environment variable set to "
                    << Machine::getDisplayProcessorName() );
                GCL::buildmsg( "socketPort = ", socketPort );
                GCL::buildmsg( "DISPLAY environment variable set to ",
                    Machine::getDisplayProcessorName() );
                str = str + "( PATH=$PATH:/usr/java/bin; export PATH; "
                    + "DISPLAY=" + Machine::getDisplayProcessorName()
                    + ":0.0 ; export DISPLAY; cd java; java JFrontEnd "
                    + Machine::getProcessorName() + " " + socketPortInt + " & )";
                system( str.c_str() );
                DBG( "Command = " << str );
                init_server( socketPort );
                connection = accept_client();
            }
        }
    }
}
```

```

        if( connection ){
            while( quitNotPressed() ){
                checkGUI();
            }
        } else {
            sendFrontEndToExecutorMessage( QUIT );
        }
    }
}
break;

/* snip */
} /* end switch( rank() ) */

/* snip */
} /* end CommandProgram::run() */

```

**Figure 1 CommandProgram::run method**

The portion of the run method that launches the java application, JFrontEnd, is shown in figure 1. Two values are passed to the java application as command line arguments passed to the system command, the name of the processor on which both the java application and the Command Program process FrontEnd are running and the TCP Port to use for the socket through which these two processes will communicate. The first of these arguments is provided by the PGM Middleware layer via a call to Machine::getProcessorName(), while the second is an optional command line argument placed in the mpirun script with a default value of DEFAULT\_GUI\_PORT if none is provided. [Note: the value placed in mpirun by the makefile that generates it uses the value of the users environment variable "PGMT\_SOCKET" if it is defined.] Before the Java virtual machine is invoked, the DISPLAY environment variable is set for the Unix process to the value returned by the PGM Middleware method Machine::getDisplayProcessorName. This is passed as a command line argument to the GramGraph application in mpirun, but unlike socketPort it is stripped off before the CommandProgram::run method is invoked. Setting this environment variable is not necessary if the installation of MPICH on the machine on which mpirun is being invoked uses ssh to spawn processes on remote processors (as opposed to rsh).

When the user presses the Quit button, the transmits this message to the FrontEnd process via the socket and exits, as seen in figure 2.

```

class ButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent e){
        if( e.getActionCommand() == "Start" ){
            //System.out.println( "Start pressed" );
            toCmdProg.println( "GUI: start was pressed" );
            toCmdProg.flush();

            /* snip */

        } else if( e.getActionCommand() == "Quit" ){
            System.out.println( "Quit pressed" );
            toCmdProg.println( "GUI: quit was pressed" );
            toCmdProg.flush();
            System.exit(0);
        } else {

```

```

        System.out.println( "Error: Wrong Action" );
    }
}
}

```

**Figure 2 JFrontEnd - exiting when Quit button is pressed**

## **Interprocess Communication**

Bi-directional communication between the Java GUI process and the Command Program FrontEnd process is accomplished by means of a socket. [Note: no data is currently being sent from the Command Program process to the Java GUI.] The Command Program process FrontEnd is the socket server and initializes the socket with a call to the function `init_server` as seen in figure 1, with the TCP port passed as an argument to the function. Then the method `accept_client` is called which blocks waiting for the Java GUI to connect.

On the Java GUI side, a call is first made to sleep for a short period so that the socket is completely initialized before an attempt to connect is made as seen in figure 3. Then an instance of `Socket` is using the host name and TCP port passed as command line parameters. An input stream is attached to the socket to read data from the Command Program process FrontEnd and an output stream is attached to the socket to send data to FrontEnd. A `PrintWriter` is created to simply the process of sending data (`PrintWriter` provides the `println` method). Figure 2 shows the socket being used to send a Start message to FrontEnd when the user presses the Start button and a Quit message when the Quit button is pressed.

```

public class JFrontEnd extends JPanel {

    public JFrontEnd( String host, String guiPort ) {

        boolean failed;
        int i=10;
        do{
            failed = false;
            try{
                Thread.currentThread().sleep( 100 );
                System.out.println( "Opening socket to " + host +
                    " on port " + guiPort );
                socket = new Socket( host, Integer.parseInt( guiPort ) );
                fromCmdProg = socket.getInputStream();
                toCmdProg = new PrintWriter( new OutputStreamWriter(
                    socket.getOutputStream() ) );
            } catch( Exception e ){ System.err.println( e ); failed = true; }
        } while( failed );

        /* snip */
    }

    public static void main(String arg[]) {
        if( arg.length > 1 ){
            frame = new JFrame("GramGraph Control Panel");
            frame.addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {System.exit(0);}
            });
            frame.getContentPane().add(new JFrontEnd( arg[0], arg[1] ),
                BorderLayout.CENTER);
        }
    }
}

```

```

        frame.pack();
        frame.setVisible(true);
    } else {
        System.err.println( "Usage: java JFrontEnd serverName socketID" );
    }
}
}
}

```

**Figure 3 JFrontEnd - creating socket**

### Avoiding Race Conditions

The messages from the Java GUI to the FrontEnd process to start and stop the graph are sent using the socket in a very similar fashion. Both messages are forwarded to the Executor via MPI in a similar fashion. Once they are received by the Executor process, they are treated very differently. Starting the Graph requires the Executor to send MPI messages to all processes on which the Graph is to run so that `PGMT::startGraph()` can be called on those processes, while stopping the Graph results in calling `PGMT::stopGraph()` on the Executor process which in turn spawns MPI messages directed to the processes where the Graph is running. However, the messages sent by the Executor process to the processes where the graph will be started and the messages sent on behalf of the executor as a result of calling `PGMT::stopGraph()` are not sent on the same MPI communicator. Therefore, the ordering of these messages is not guaranteed. In order to make the GramGraph application robust with respect to the messages from the Java GUI, so that no combination of button presses in the GUI could cause the application to be in an unstable state, a system of acknowledgements is used to make the stopping and stopping of a Graph an atomic action.

When starting the Graph, the `sendExecutorToGraphWrapperMessage` method assures that all processes on which the Graph will be running have acknowledged to start message before completing, as seen in figure 4. A similar system of sending is used by the PGM Execution Program (PEP) to make stopping the graph an atomic process.

```

void Executor::sendExecutorToGraphWrapperMessage( int message ){

    controlMessage[0] = message;
    GraphRanks& graphRanks = get_gRanks();
    vector<MW_ProcessID>::iterator iter = graphRanks.begin();
    iter = graphRanks.begin();
    while( iter < graphRanks.end() ){
        Machine::sendCPMessage( controlMessage, 1, PGMT_INT, *iter );
        Machine::bRcvCPMessage( controlMessage, 1, PGMT_INT, *iter, CP_ACK,
                                &returned_tag, &returned_source );

        iter++;
    }
}
}

```

**Figure 4 Receiving acknowledgements from Graph processes**

In the case of GramGraph, the Graph is started (messages are sent and acknowledgements received via a call to `Executor::sendExecutorToGraphMessage()`) and stopped (`PGMT::stopGraph()` is called) from the same Command Program process. This

is not necessary. Any of the four Command Program processes could stop the Graph, for example. Unfortunately this would provide many opportunities for race conditions among the start and stop messages sent to the processes on which the Graph runs. The responsibility for avoiding this would be on the Command Program writer and doing so might be difficult as MPI only guarantees that messages between the same two processes sent on the same MPI communicator arrive in order.