Diverse Adaptable Display System
Developers Guide

by

Andrew A. Ray
Patrick Shinpaugh
Ron Kriz

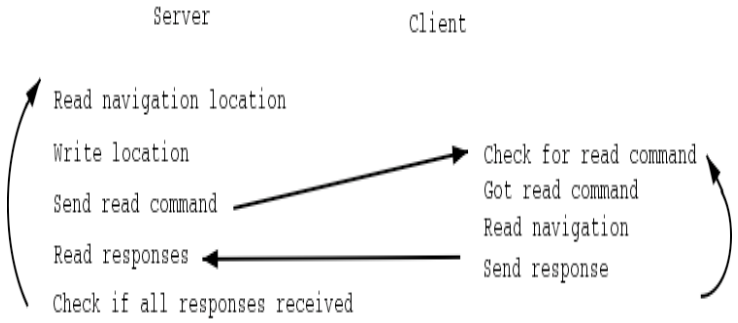
The Diverse Adaptable Display System is a simple system to develop for. We have designed it to require zero modifications from a normal DPF/DTK program except for special cases. Given the fact that you have 4 different computers running a program there can be some communication difficulties if you decide to communicate in-between different walls. This document goes over how the additional tools provided to you for the DADS system.

Outline of document:
I)Workflow
II)Messaging Library
III)Using DSOs on the DADS system
IV)Inter-process / Inter-machine communication
V)MPI and DADS

Workflow

The DADS system here at Virginia Tech is composed of one master machine and four slave display nodes. The DADS system works using the messaging library to send a message from a DSO loaded on the console machine to daemons running on the client machines. The daemons then fork processes which replicates what you ran on the console machine. The only difference is that specific display DSOs are loaded on the client systems. While an application is running the current DTK navigation data is read on the console, and a simple custom muxtex is used to make sure the location of all the display nodes is at the correct spot. This means you have to use DTK navigations otherwise the program won't work correctly. This is a very simple way of distributing the information that DTK/DPF need to function properly.

Loop diagram:



This is the basis for the DADS system. Navigation information is considered to be location, head, wand, joystick, and button information.

Messaging Library - #include <dMessaging.h>

One of DTK's strengths is its shared memory component. Instead of developing yet another socket communication protocol we decided to leverage the power of DTK and create a simple messaging library that functions similar to email.

The idea behind a message is that it is like a postcard with a queue that you can use to hold your data. It is a postcard because anyone can read the message. It is up to you to implement security if you need it in your application. The queue of data allows you to push int/string/float/double/char values into the message. After a message has been constructed you can then send it to the local post office. Post offices may be connected to post offices on different machines so that you can send messages from one computer to another.

API description:

Class dMessage:
Description: The messaging class for D.A.D.S.
Purpose: To provide a simple and complete way to send control messages to programs. This is not designed to send large amounts of data from point A to point B. This is because DTK Shared Memory cannot do this and because messages are based on DTK so they inherit this limitation.

Constructors:
dMessage() - Creates a blank message

dMessage(string) – Creates a messssage from the input string. This expects the message to be in the form from, to, subject, data.

Functions:

void clear() - Erases all data from the message

string getMessage() - Get the entire message as a string. This is mainly used internally when sending messages

The get/(set/add) functions for the message either return the specified information or set it. They message header get/(set/add)functions use strings and the rest of them function based the type specified in the get/(set/add) function call.

Headers (get/set):

From – The message origin

To – The message destination

Subject – The message subject

Data Values (get/add):

int

float

double

char

string

Class dPostOffice:

Description: The message sending class for D.A.D.S.

Purpose: To provide a simple and complete way to send dMessage instances to a everyone connected to the local postoffice

Constructors:

dPostOffice(name) – Create a post office that will use the shared memory segment name to communicate. This segment will also be the default segment.

Functions:

int sendMessage(dMessage*) - Send the specified message on the default segment and return a 0 on success or a 01 on error.

dMessage* getMessage() - returns a message from the post office mailbox if one exists. If it doesn't exist then the Post Office will return NULL.

int connectTo(string) – Connect the default local segment on this machine to the post office on the input string. This is a simple dtk-connectRemoteShared mem command and will return 0 on success and a different value on failure.

int addSegment(string) – add an extra segment in addition to the one specified in the constructor. Returns 0 on success and a different value on failure.

int setDefaultSegment(string) – set the default segment to be the input string. Returns 0 on success and a different value on failure.

int sendMessage(dMessage*, string) - send a message on the specified segment. Returns 0 on success and a different value on failure.

int connectTo(string, string) – connect to the segment on the specified computer. Returns 0 on success and a different value on failure.

dMessage* getMessageFrom(string) – get a message from the specified segment. Returns NULL if no message exists.

Examples of how to use the message library:

Sending a message:

```
dPostOffice MyOffice(“MY_SEGMENT”);
dMessage Amessage;
Amessage.setFrom(“Test_Client”);
Amessage.setTo(“Test_Server”);
Amessage-.setSubject(“Message Example”);
Amessage.addInt(3);
Amessage.addChar('.').
Amessage.addInt(1);
Amessage.addInt(4);
MyOffice.sendMessage(&Amessage);
```

Receiving a message:

```
dPostOffice MyOffice(“MY_SEGMENT”);
dMessage* TestMessage = NULL;
while (TestMessage == NULL)
{
        TestMessage = MyOffice.getMessage();
        usleep(10000);
}
cout << “Test message from “ << TestMessage->getFrom() << endl;
cout << “Test message to “ << TestMessage->getTo() << endl;
cout << “Test message subject is“ << TestMessage->getSubject() << endl;
cout << “Test message number is “ << TestMessage->getInt() << TestMessage-
```

>getChar() << TestMessage->getInt() << TestMessage.getInt() << endl;
cout <<"Done" << endl

Extending the previous example to running on multiple machines:

Simply add the line:
MyOffice.connectTo("machinename");
to your server and client programs that you want to connect to.

Building programs with the messaging code:

Make sure to use #include <dMessaging.h> for the definitions of the two classes above. In the link command in your makefile make sure you have a -ldmessage included. All of these files are located in the standard DTK installation directory so if you use dtk-config in your makefiles everything should work.

Using DSO's with the D.A.D.S system:

There are three different types of DSO's on the D.A.D.S system. Console DSO's, cluster DSO's, and cluster machine DSO's. The console DSO's are simply the DPF_DSO_FILES used in DPF/DTK. They are used to configure the application running on the console. The cluster DSO's are represented by DPF_CLUSTER_DSO_FILES. They allow you to specify what DSO's are going to be loaded on all of the other nodes in the D.A.D.S system. The cluster machine DSO's allow you to specify specific DSO's to load on a specific machine. An example of this is to load the left wall DSO on machine X and the right wall display DSO on machine Y.

What does this mean for you the developer? This means that if you develop a DSO that loads on all the cluster machines then you want to specifiy it in the DPF_CLUSTER_DSO_FILES. If you want to have a DSO that is loaded on the console machine you use DPF_DSO_FILES. If you want to have a specific DSO loaded on machine X then you would use DPF_CLUSTER_DSO_FILES_X.

Inter-process / Inter-machine communication:

If your program needs to communicate data between the different nodes in the DADS system then you need to use the messaging library or something similar. A simple way to make sure that your program will scale to any system is to have all control decisions sent by messages. If this is the case then you can simply connect all of the nodes in the system to the event generator (console to the front wall, etc...) and they should on a big iron computer(SGI).

An example of this is AtomView. AtomView generates events based on a 2D gui displayed on the front wall of the cave or on your desktop. Because the events generated

from this gui are in message form you can make all of the other machines receive these events simply by connecting the shared memory segments. As long as you use this method encode your events porting your program from a desktop to the D.A.D.S system should be painless.

MPI and DADS:

The notion of MPI fits the problem DADS solves very well. It provides a simple and elegant method for replicating process among different machines and provides data passing, and synchronization capabilities while completely hiding low level network details. During the development of the DADS system MPI was investigated as a potential method for implementation. However, it was decided that because of its center of the universe concept at runtime, and the changes that would have to be made in user workflow MPI was not a proper choice for developement. This design choice was validated by performance tests done by Andrew Ray. MPI provides similar performance to the DADS system, but the DADS system is marginally faster. At the current state of VR every ounce of performance counts. If you wish to use MPI with DIVERSE it requires integration of MPI into your DIVERSE application. With the current status of DTK/DPF it is not possible to allow the usage of MPI in DSO's only. Here is a code example of how to use MPI in a DADS style system. It hardcodes a file for model navigation. This program assumes only four walls are present and that the front wall(dads1) is connected to the tracker system In short, this is a mpi program that causes DIVERSE to work on the four different walls (dads1-4).

```
//Author Andrew A. Ray
//Based on diversifly
#include <mpi.h>
using namespace std;
#include <dpf.h>
#include <stdio.h>
#include <unistd.h>

#define DIFFUSE_R (1.0f)
#define DIFFUSE_G (1.0f)
#define DIFFUSE_B (0.8f)

#define AMBIENT_R (0.4f)
#define AMBIENT_G (0.4f)
#define AMBIENT_B (0.3f)

int sceneLightc;
pfVec3 **sceneLightv;

// number of etherlights to load, and their positions
int etherLightc;
```

```
pfVec3 **etherLightv;

// number of worldlights to load, and their positions
int worldLightc;
pfVec3 **worldLightv;
dtkSharedMem* wandSegment;
dtkSharedMem* headSegment;
dtkSharedMem* buttonSegment;
dtkSharedMem* joystickSegment;
dtkNavList*   navList;
dtkNav*       nav;
float  headData[6];
float  wandData[6];
float  joystickData[2];
char   buttonData;
float location[6];
char  data[100];

int main(int argc, char** argv)
{
        MPI::Init(argc, argv);
        int rank = MPI::COMM_WORLD.Get_rank();
        int size = MPI::COMM_WORLD.Get_size();

        //if (rank > 0)
        //{

        setenv("DISPLAY", "localhost:0",1);
        setenv("DPF_CLUSTER_SERVER", "hammer",1);
        setenv("DPF_CLUSTER_CLIENTS", "dads1:dads2:dads3:dads4",1);

        char* hostname= new char[80];
        gethostname(hostname, 80);
        dpf* app =NULL;
        if (strcmp(hostname, "dads1") ==0)
                app  = new dpf("xkeyboardMouseInput", "caveDTKInput",
"setHeadView", "caveSim", "dHideCursor", "toggleObjectsGroup",
"toggleScreenFrame", "vtCaveDisplayFront", "wandJoystickNav", "buttonNavControl",
NULL);
        else if (strcmp(hostname, "dads2") ==0)
                app  = new dpf("xkeyboardMouseInput", "caveDTKInput",
"setHeadView", "caveSim", "dHideCursor", "toggleObjectsGroup",
"toggleScreenFrame", "vtCaveDisplayFloor", "dummyNav", NULL);
```

```
        else if (strcmp(hostname, "dads3") ==0)
                app  = new dpf("xkeyboardMouseInput", "caveDTKInput",
"setHeadView", "caveSim", "dHideCursor", "toggleObjectsGroup",
"toggleScreenFrame", "vtCaveDisplayRight", "dummyNav", NULL);
        else if (strcmp(hostname, "dads4") ==0)
                app  = new dpf("xkeyboardMouseInput", "caveDTKInput",
"setHeadView", "caveSim", "dHideCursor", "toggleObjectsGroup",
"toggleScreenFrame", "vtCaveDisplayLeft", "dummyNav", NULL);
        else
                app = new dpf("desktopCaveEmulateGroup", "dPerformance", NULL);
        app->config();

        wandSegment = new dtkSharedMem(sizeof(float)*6, "wand");
        headSegment = new dtkSharedMem(sizeof(float)*6, "head");
        joystickSegment = new dtkSharedMem(sizeof(float)*2, "joystick");
        buttonSegment = new dtkSharedMem(sizeof(char), "buttons");
        navList = (dtkNavList*)app->check("dtkNavList", DTKNAVLIST_TYPE);
        if (navList == NULL)
                printf("ERROR bad navlist\n");
        //setenv("DPF_FILE",
"/home/diverse/vt/arch/StMaria/Models/Textures/SmmMaster.pfb",1);
        //chdir("/home/diverse/vt/arch/StMaria/Models/Textures");
        //setenv("DPF_FILE", "/home/shpatric/dads-cluster/cluny.pfb",1);
        setenv("DPF_FILE", "sub.pfb",1);
        char* LOADFILE=getenv("DPF_FILE");
        if (LOADFILE == NULL)
        {
                printf("Error no file found in DPF_FILE\n");
                return 0;
        }
//BEGIN LIGHT
  sceneLightv = (pfVec3 **)
    realloc(sceneLightv, sizeof(pfVec3 *)*(sceneLightc+1));
  sceneLightv[sceneLightc] = new pfVec3 ;
  sceneLightv[sceneLightc]->set(0.0f, -sqrt(2.0f), sqrt(2.0f)) ;
  sceneLightc++ ;

    pfGroup* lightgroup = new pfGroup;
  int c ;
  for (c=0; c<sceneLightc; c++) {
    pfLightSource *l = new pfLightSource ;
    l->setColor(PFLT_AMBIENT, AMBIENT_R, AMBIENT_G, AMBIENT_B) ;
    l->setColor(PFLT_DIFFUSE, DIFFUSE_R, DIFFUSE_G, DIFFUSE_B) ;
```

```
    l->setPos((*sceneLightv[c])[0],
              (*sceneLightv[c])[1],
              (*sceneLightv[c])[2], 0.0f ) ;

    if(!lightgroup->addChild(l)) {
      dtkMsg.add(DTKMSG_ERROR,
                 "scene->addChild(sceneLight[%d]) failed.\n",c) ;
      return 1 ; // failure, error
    }
   }
//END LIGHT

//ADD data to world
        app->display()->world()->addChild(lightgroup);
        app->display()->world()->addChild(pfdLoadFile(LOADFILE));
        int totalFrames=0;
        int numFrames=0;
        int master = 0;
        int numIncrements=0;
        double startTime= MPI_Wtime();
        double beginTime= MPI_Wtime();
        double elapsedTime = 0;
//RUN
        while(app->state & DTK_ISRUNNING)
        {
                totalFrames++;
                numFrames++;
                app->frame();
                if (rank == master)
                {
                        elapsedTime = MPI_Wtime() - startTime;
                        if (elapsedTime > 1.0)
                        {
                                printf("Number frames = %f\n", numFrames/elapsedTime);
                                printf("Average frame rate = %f\n",
totalFrames/(MPI_Wtime()-beginTime));
                                numFrames=0;
                                startTime=MPI_Wtime();
                        }
                        nav =  navList->current();
                        if (nav == NULL)
                                printf("ERROR bad navigation\n");
                        headSegment->read(headData);
```

```
                wandSegment->read(wandData);
                joystickSegment->read(joystickData);
                buttonSegment->read(&buttonData);
                memcpy(location, nav->location.d, sizeof(float)*6);
                //printf("%d\n", numFrames);
                /*
                if (numFrames > 1000)
                {
                        dtkAugment* augment = app->check("dPerformance");
                        app->remove("dPerformance");
                        delete augment;
                        //dPerformance* perf = (dPerformance*) augment;
                        //perf->die();
                }
                */
                memcpy(data, headData, sizeof(float)*6);
                memcpy(data+sizeof(float)*6, wandData, sizeof(float)*6);
                memcpy(data+sizeof(float)*12, location, sizeof(float)*6);
                memcpy(data+sizeof(float)*18, joystickData, sizeof(float)*2);
                memcpy(data+sizeof(float)*20, &buttonData, sizeof(char));
        }
        MPI_Bcast(data, sizeof(float)*20+sizeof(char), MPI_CHAR, master,
MPI_COMM_WORLD);


        if (rank != master)
        {
                memcpy(headData, data, sizeof(float)*6);
                memcpy(wandData, data+sizeof(float)*6, sizeof(float)*6);
                memcpy(location, data+sizeof(float)*12, sizeof(float)*6);
                memcpy(joystickData, data+sizeof(float)*18, sizeof(float)*2);
                memcpy(&buttonData, data+sizeof(float)*20, sizeof(char));
                nav =  navList->current();
                if (nav == NULL)
                        printf("ERROR bad navigation\n");
                headSegment->write(headData);
                wandSegment->write(wandData);
                joystickSegment->write(joystickData);
                buttonSegment->write(&buttonData);
                nav->location.set(location);
        }
        MPI_Barrier(MPI_COMM_WORLD);
}
```

```
        delete app;
        pfExit();
        MPI::Finalize();
        return 0;
}
```

Feedback: Questions/Comments/Constructive Criticism:
Feel free to send us your thoughts on the D.A.D.S system. You can contact us via diverse-devel@lists.sourceforge.net