UCRL-MA-147859

Tool Gear Documentation

J. May, J. Gyllenhaal

April 3, 2002

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

DISCLAIMER

This work was performed under the auspices of the U. S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

Tool Gear Documentation

Developing Tools with Tool Gear

Tool Gear is designed to allow tool developers to insert instrumentation code into target programs using the DPCL library. This code can gather data and send it back to the Client for display or analysis. Tools can use the Tool Gear client without using the DPCL Collector. Any collector using the right protocols can send data to the Client for display and analysis. However, this document will focus on how to gather data with the DPCL Collector.

There are three parts to the task of using Tool Gear to gather data through DPCL:

1. Write the instrumentation code that will be loaded and run in the target program. The code should be in the form of one or more functions, which can pass data structures back to the Client by way of DPCL. The collections of functions is compiled into a library, as described below.

2. Write the code that tells the DPCL Collector about the instrumentation and how to forward data back to the Client.

3. Extend the client to accept data from the Collector and display it in a useful way.

The rest of this document describes how to carry out each of these steps.

I. Writing instrumentation code.

DPCL loads instrumentation code into a target program, and this instrumentation gathers data (or controls the program in some other way) and optionally sends data back to the Client by way of DPCL and the Collector. DPCL can cause a program to call any function, so there are numerous possibilities for interacting with the target code. DPCL can execute instrumentation three ways: as a "point probe," meaning that the code is executed whenever the target program reaches a specific location; as a "one-shot probe," which is executed immediately regardless of what the target program is doing at the time; or as a "phase probe," which is executed repeatedly at a specific time interval. At present, DPCL can insert point probes only at certain locations in a target program: just before and after function calls, and at the beginning and end of functions. Regardless of how the instrumentation is activated, the process of writing the probe functions is the same.

Although DPCL supports a variety of features for building "probe expressions," which are segments of code that run in a target program, Tool Gear focuses on supporting function calls that the the tool developer has written to gather data. These functions are called "probe functions," and they are contained in libraries called "probe modules," which DPCL loads into a running target program. Tool developers can extend the DPCL Collector to use the full capabilities of DPCL to create probe expressions, but that task is complicated, and it is not described here.

Probe functions are typically written in C and take one of two forms:

```
void MyProbe( void );
```

-- or --

```
#include <dpclExt.h>
void MyCounterProbe( AisPointer ais_send_handle );
```

The first form is used when the function doesn't need to send any data back. This might be appropriate when the tool is initializing a measurement or shutting down the instrumentation. In the second form, the probe function can send data to the Collector. The ais_send_handle is an opaque object that DPCL passes to the function. The function uses this handle to send data back to the DPCL Collector as follows:

```
#include <dpclExt.h>
void MyCounterProbe( AisPointer ais_send_handle )
{
        static int counter;

        counter++;

        Ais_send( ais_send_handle, &counter, sizeof( counter ) );

}
```

This example implements a simple counter and reports how many times it has been called. Calling Ais_send sends the data to the Collector, which invokes a user-defined callback to handle the data, as described in Part II. Although this probe function is sending back only a single integer, it could also send a larger structure containing more data.

Once the instrumentation has been written, it is compiled and converted to a probe module. This process requires two additional files: an export file and an import file. (This information applies to IBM platforms; other systems may have different requirements.) The export file lists the names of the probe functions. It is a plain-text file, and for the two functions listed above, it would look like this:

```
* Asterisks at the beginning of a line indicate comments.
MyProbe
MyCounterProbe
```

The import file is also plain-text file. It lists functions whose names are resolved at run time. In particular, this includes Ais_send. So a simple import file would consist of these two lines (the #! is required for the linker to generate a file in the format that DPCL needs):

```
#!
Ais_send
```

The import and export files are often named with .imp and .exp suffixes, respectively. Once they have been created, you can compile the probe module as follows (assuming your probe module source code is in mycountermodule.c, and the import and export files are named accordingly):

```
cc -g -o mycountermodule mycountermodule.c -I/usr/lpp/ppe.dpcl/include \
        -bE:mycountermodule.exp -bI:mycountermodule.imp -bnoentry
```

The -g flag is needed (in our experience) for DPCL to pass the ais_send_handle correctly. The -bnoentry flag prevents the linker from looking for a main function and trying to create an executable program. The -I flag points to the location of dpclExt.h. You can also link additional libraries into the probe module using the standard -L and -l flags.

II. Customizing the DPCL Collector

The DPCL Collector runs on the same computer as the target application, and it serves as a bridge between the target and the Client, which stores and displays data, and may run on a different computer. The Collector, at the request of the Client, controls the target application, inserts instrumentation, and receives and forwards data. As provided, the Collector knows how to execute a target application, start, stop, and terminate it, change directories, find the source code, and determine the points where it can be instrumented. The Collector also knows how to communicate with the Client, and it includes a set of functions for creating columns in the Client's source code display and placing data there.

The tool developer is responsible for writing the customization code that tells the Collector where to find the probe module, what functions it contains, and what requests from the Client should trigger various actions. The tool developer is also responsible for the callback functions that receive data from the target application and forward it to the Client. The Collector partly automates many of these steps, so they require a relatively small amount of code. The Collector is written in C++, and it's easiest to write the customization code in C++ as well.

The DPCL Collector offers several classes that the customization code will likely use. Full descriptions of these classes' members appear later in this section.

A DPCLActionType represents a probe function in the Collector. It does not represent a particular instantiation of the function; only the ability to instantiate that function.

A DPCLPointAction is a DPCLActionType that has been instantiated as point probe. In other words, the specified probe function will be called whenever the target program reaches a particular location. Since the tool user specifies the location through the Client's GUI, and this location is transmitted from the Client to the Collector, the developer of customization code normally doesn't have to deal with the objects that represent instrumentation points, which are encapsulated in a class called DPCLActionPoint (not DPCLPointAction).

A DPCLOneShotAction is a DPCLActionType that has been instantiated as a one shot probe. The Collector currently has only limited abilities to instantiate this type of probe automatically.

A DPCLPhaseAction is a DPCLActionType that has been instantiated as a phase probe. THIS CLASS HAS NOT YET BEEN IMPLEMENTED.

The first step for the tool builder writing customization code is to provide the Collector with a list of DPCLActionTypes. This is done through an initialization function. The tool builder writes this function, which typically has a prototype like this:

```
#include "dpcl_action_type.h"
(other include files will also be needed)
int InitializeMyActions( int socket, int& actionCount,
                         DPCLActionType **& actionList );
```

The socket is used for communicating with the Client, and the actionList and actionCount contain the array of (pointers to)

DPCLActionTypes that the Collector knows about already, and the length of that list. This function will extend the array, add pointers to new DPCLActionTypes, and update the actionCount.

The basic steps are: Find the probe module, extend the actionList to hold the new DPCLActionTypes, create the new DPCLActionTypes, and tell the Client about the DPCLActionTypes and the data they will be sending.

Here is a sample initialization function that carries out these steps. It creates just one action, representing the counter probe defined in Section I. See the comments for details.

```
#define COUNTER_ENV_VAR "COUNTER_PROBE_MODULE"
#define COUNTER_MODULE "mycountermodule"
#define COUNTER_ACTIONS 1
#define COUNTER_TAG "startCounters"
#define COUNTER_COLUMN "count"

#include <dpcl.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include "counter_actions.h"
#include "dpcl_action_type.h"
#include "dpcl_action_instance.h"
#include "dpcl_pack.h"
#include "md.h" // defines datatypes for Client display
#include "tg_socket.h"

// Declaration of the callback function
void CounterProbe_cb( GCBSysType sys, GCBTagType tag,
                      GCBObjType obj, GCBMsgType msg );

// Initialize the counter probe and tell the client about it.
// Returns 0 on success, nonzero otherwise.  actionCount
// and actionList are in/out parameters.
int InitializeCounterAction( int sock, int& actionCount,
                      DPCLActionType **& actionList )
{
        // Find the probe module; look first in an environment
        // variable, then try a hard-coded location.
        char * module_name = getenv( COUNTER_ENV_VAR );
        if( module_name == NULL )
                module_name = COUNTER_MODULE;

        // Check that the file exists and is readable.
        if( access( module_name, R_OK | X_OK ) != 0 ) {
                fprintf( stderr,
                        "Error accessing probe module %s: %d\n",
                        module_name, errno );
                return -1;
        }

        // See if another set of actions is already in
        // the list; if not, create the list from scratch;
        // otherwise, add to the end of the list.
        if( actionCount == 0 ) {
                actionList = new
                        DPCLActionType*[COUNTER_ACTIONS];
                if( actionList == NULL ) return -1;
        } else {
                // Make a new, longer list; copy the old list to
                // the beginning, then we'll add the new actions
                // to the end.
                DPCLActionType ** oldList = actionList;
```

```
                DPCLActionType ** newList
                        = new DPCLActionType*[actionCount +
                                                COUNTER_ACTIONS];
                if( newList == NULL ) return -1;

                for( int i = 0; i < actionCount; i++ ) {
                        newList[i] = oldList[i];
                }
                actionList = newList;
                delete [] oldlist;
        }

        // Define the action type and tell the Client about it
        // The parameter are: the text tag that identifies the
        // action to the Client, the name of the probe module,
        // the name of the probe function, the socket identifier,
        // and an optional callback function.
        actionList[actionCount++] = new DPCLActionType(
                                COUNTER_TAG,
                                module_name, "MyCounterProbe",
                                sock, CounterProbe_cb );

        // Tell the client about this action type.  Parameters
        // are: tag, short name, description (used for help
        // text), and socket name.
        pack_and_send_action_attr( COUNTER_TAG, "Get count",
                "Cumulative counter", sock );

        // Declare columns in the client's display where data
        // will be shown.  Parameters are: tag for the column,
        // column label, description of data, data type, and
        // socket.
        pack_and_send_data_attr( COUNTER_COLUMN, "Count",
                "Number of time point has been reached",
                MD_INT, sock );

        // Data isn't sent to the Client until this function
        // is called.
        TG_flush( sock );

        return 0;
}
```

The next step is to define the callback function that will be invoked each time the corresponding probe function executes. As shown in the code above, this function is specified when the DPCLActionType object is created. If a callback is given, the probe function will be called with a Ais_handle parameter. If a null pointer is given for the callback (or the last parameter of the constructor is omitted), then the probe function will be called with no arguments. Failing to specify a callback when a probe function expects one or supplying one when it isn't needed is likely to cause a crash when the probe function is called.

The callback function's prototype is the standard form for DPCL. The four parameters are:

GCBSysType is a DPCL data structure that describes how the message was sent from the target. See the DPCL documentation for details.

GCBTagType is a pointer-size value. In the DPCL Collector, the tag is set to point to the DPCLPointAction that caused the instrumentation to execute.

GCBObjType is a pointer to the DPCL Process object that represents the process in the target program that sent the data. In a parallel program, several different processes could execute the probe function for the same DPCLPointAction.

GCBMsgType is a pointer to the message that the probe function passed to Ais_send. The size of the message is given in the GCBSysType structure as sys.msg_size.

Here is an example of a callback that receives the count data from the probe function and forwards it to the client:

```
void CounterProbe_cb( GCBSysType sys, GCBTagType tag,
                      GCBObjType obj, GCBMsgType msg )
{
        int * count = (int *)msg;        // msg contains the count
        Process * p = (Process *) obj;   // DPCL Process
        int task = p->get_task();        // parallel process id
        DPCLPointAction * action =
                (DPCLPointAction *) tag;

        // Retrieve the location in the program of this
        // point probe
        DPCLActionPoint * location = action->get_owner();

        // Where to send the data
        int sock = action->get_type()->get_socket();

        // Send the count data to the Client.  Parameters are:
        // target program function name, identifier for the
        // instrumentation point (so the Client knows where in
        // the source code to display the data), name of the
        // column where data will appear, task id, thread id
        // (unused here), data value, and socket.
        pack_and_send_int( location->get_function_name(),
                location->get_tag(), COUNTER_COLUMN, task, 0,
                *count, sock );

        TG_flush( sock );

}
```

If your instrumentation needs some type of cleanup before the program exits, you can create a special probe function and identify it with the tag "finalize" (when you call the DPCLActionType constructor). An action type with this tag will be called automatically as a one-shot probe just before the program exits.

Once you have defined the callbacks and the initialization code for the actions, you can compile it in with the rest of the Collector code. (Dynamic loading isn't offered yet.) To do this, first insert a call to your initialization code in the file dpcl_collector.cpp, as shown here:

```
        ...
        // Set up handler to receive messages from GUI
        Ais_add_fd( sock, dpcl_socket_handler );

        // Set up the tool-specific actions that the collector
        // will use
        if( InitializeCounterAction( sock, actionCount,
                        actionList ) != 0 ) {
                // Initialization failed; tell Client to quit
                TG_send( sock, DPCL_SAYS_QUIT, 0, 0, NULL );
                TG_flush( sock );
```

```
        }
        ...
```

You can call several initialization functions in succession, if necessary. You will also need to declare these functions at the beginning of the file. To compile everything, simply add the name of the .o file for your initialization code and callbacks to the TG_DPCL_OBJECTS line in the Makefile. When you run the program, you will need to set an appropriate environment variable to point to the absolute path of the probe module. Remember that the Client will launch a new shell to run the Collector, so the environment variable must be set in a .login file. You can also hard-code the file name into the initialization code, as shown above, but this may not work as well as an environment variable, because the Client may change to any working directory that the user requests before starting the Collector. For that reason, the Collector's executable code must be in the search path for the shell the that Client starts.

The simple example of Collector code shown in this section illustrates the basic steps, but the tool it defines isn't very useful. For a real-world example of a tool built using Tool Gear, see mpx_counters.cpp and the MPX probe module software in cacheperf.c.

III. Extending the Client to display data

Tool Gear currently allows the tool writer to customize the icons, menu text, and tool tips the user encounters when using the generic tool gear GUIs. As the infrastructure evolves, it is envisioned that customization can occur both on client and server side and significant changes are planned to support these and other customizations. Currently, most customizations are placed in the client side in gui_socket_reader.cpp in the void GUISocketReader::unpack_and_declare_action_attr (char *buf) method.

Every action that is declared by the server passes thru this routine. Customizations are added by testing the action_name and adding the desired customizations using UIManager calls.

Pixmaps are expected in the XPM 3 format. This text-based format is the de facto standard and can be easily embedded in C/C++. Please see http://koala.ilog.fr/lehors/xpm.html for the format details, tools for converting from earlier versions, etc.

For example, the startCounters pixmap is declared as follows in gui_socket_reader.cpp:

```
static const char * startCounters_xpm[] = {
"15 15 6 1",
"       c None",
".      c #000000000000",
"G      c #0000FFFF0000",
"R      c #FFFF00000000",
":      c #E500E500E500",
"W      c #FFFFFFFFFFFF",
"WWWWWWWWWWWWWWW",
"W:::GGGGGGG:::.",
"W::GG.G.G.GG::.",
"W:GGG.G.G.GGG:.",
"WGGG.......GGG.",
"WG...........G.",
"WGGG.......GGG.",
"WG...........G.",
"WGGG.......GGG.",
```

```
"WG..........G.",
"WGGG.......GGG.",
"W:GGG.G.G.GGG:.",
"W::GG.G.G.GG::.",
"W:::GGGGGGG:::.",
"W.............."};
```

Using as small set of UIManager calls (uimanager.h), the pixmaps used by an action can be declared as well as the states the action can be in. Currently the DPCL server only expects two states, instrumented and uninstrumented and in the following example we name these states 'In' and 'Out'.

Example for handling 'startCounters' in unpack_and_declare_action_attr():

```
...
// Set up GUI for MPX action 'startCounters'
if (strcmp (actionName, "startCounters") == 0)
{
    // Declare start counter pixmap
    um->declarePixmap ("startCountersPixmap", startCounters_xpm);

    // Declare pixmaps, menu text, and tool tip for the
    // uninstrumented state 'Out'
    um->declareActionState ("startCounters",
            "Out",
            // Pixmap for removing
            "startCountersPixmap",
            "Remove Start Counters",
            "Remove mpx probe to start cache and FLOP measurements from here",
            "", // No pixmap if removed
            "Uninstrumented potential start counters location",
            TRUE);

    // Declare pixmaps, menu text, and tool tip for the
    // instrumented state 'In'
    um->declareActionState ("startCounters",
            "In",
            "startCountersPixmap",
            "Start Counters",
            "Insert mpx probe to start cache and FLOP measurements here",
            "startCountersPixmap",
            "Starts cache and FLOP measurements here (starts new mpx region)",
            TRUE);
}
else if (<test for other actions>)
...
```

As the user changes states from 'Out' to 'In' and back in the GUI, the UIManager state gets changed. This causes UIManager throw a signal that the action state has changed. The GUIActionSender (gui_action_sender.cpp) listens for these state changes and relays each state change to DPCL server.