This document contains information relating to technology that is still under development. IBM may or may not decide to incorporate some or all of the information contained herein into future IBM products.

Dynamic Probe Class Library (DPCL): Tutorial and Reference Guide

Version 0.1

June 30, 1998

Dr. Douglas M. Pase
email:pase@us.ibm.com

IBM Corporation
RS/6000 Development
522 South Road, MS P-963
Poughkeepsie, New York 12601

Draft Document

Table of Contents

1.0 About DPCL

1.1 Why DPCL Is Interesting

DPCL is an application program interface (API) for installing instrumentation into and removing instrumentation from a serial or parallel program as the program is running. DPCL instrumentation may measure execution time for performance tools, pass counts for test coverage tools, report or modify the contents of variables for debuggers, and many other things. Instrumentation is defined by the tool builder and therefore has unlimited flexibility. Instrumentation is placed in the application dynamically, as it is needed, and removed when it is no longer desirable. This keeps the cost of gathering data quite low, even on long running and highly parallel jobs. DPCL is designed to:

- reduce the cost of developing new programming tools
- reduce the intrusion cost of instrumentation
- increase the flexibility and usability of tools
- increase interoperability among tools
- increase innovation in tool development
- increase the number and variety of available programming tools.
- provide a mechanism for creating common tools across the industry

DPCL is designed to take advantage of dynamic instrumentation technology originally developed under Bart Miller at the University of Wisconsin, Madison, by Jeff Hollingsworth, who is currently at the University of Maryland. The API allows a tool built on DPCL to insert data, functions, and code patches into a program while it is running. Code patches, or probes, can collect and report performance information, program state, or modify the program execution.

The original motivation for DPCL came from the observation that customers were often asking for more application performance analysis tools than the various tool suppliers had resources to build. High performance application developers were asking for tools that would provide detailed, accurate information about disk usage, cache and other memory usage, CPU and functional unit usage, message passing and synchronization, and operating system interference with achieving high performance. Furthermore, they were asking for application profiles to determine what problems were occurring (problem determination), and traces to determine what was causing observed problems (root cause analysis).

As DPCL development became better known, other uses for this technology began to surface. In addition to performance analysis tools, other potential tools were identified as candidates for development under DPCL. Among them are correctness debuggers, memory debuggers, relative debuggers, hardware performance monitors, test coverage tools, Reliability/Availability/Serviceability support tools, application steering tools, dynamic load balancing tools, and more.

One of the major costs of developing new tools is in developing its instrumentation. Some examples of the types of instrumentation are: manually inserted and compiler generated instrumentation, instrumentation of object files, instrumentation at load time, instrumented libraries, and dynamic instrumentation. The specific cost of developing the instrumentation depends on the type of instrumentation chosen and has to be played against the user effort required to use the tool. Each different type of instrumentation also carries its own impact on the accuracy and precision of gathered data, as well as the flexibility with which the data can be gathered. Dynamic instrumentation allows the greatest flexibility in gathering data, which can be used to focus attention on specific items of interest, increase accuracy by reducing interference caused by gathering unwanted data, or increase convenience to the user by delaying the decision point for instrumentation until run-time.

With dynamic instrumentation the instrumentation code need only reside in the application as long as it is needed to gather data. When a problem is suspected the instrumentation can be inserted into the application to gather data needed to verify the problem. Once the problem is verified the instrumentation can be replaced with more detailed instrumentation to establish the cause of the problem. If the initial guess turns out to be incorrect, the original instrumentation can be replaced with new instrumentation that examines other possible causes.

Since the goal is to create a variety of tools that are inexpensive as well as flexible, it is important to address the cost of tool creation. The exact fraction of the cost will vary from one tool to the next, but in many cases the direct and indirect costs of writing instrumentation software dominates the cost of the tool. Whether the instrumentation is part of the compiler, the linker, or a library of subroutines, writing the instrumentation package often requires substantial communication, coordination, and cooperation across development groups, which can be expensive. If a way is provided that allows large portions of the instrumentation system to be reused, the cost of writing instrumentation could be reduced.

DPCL provides a general purpose infrastructure that flexibly supports the generation of arbitrary instrumentation. It is capable of instrumenting serial, shared memory, and message passing applications. Based on dynamic instrumentation it requires only the information usually found in an `a.out`. Once the infrastructure is in place, the cost of writing instrumentation for an individual tool is a tiny fraction of the cost of writing tool instrumentation from scratch. Because the infrastructure provides general programming facilities for writing instrumentation the system can be used for a wide variety of tools and uses. DPCL is expected to be used for developing several application performance analysis tools, correctness debuggers, memory debuggers, application steering tools, and many other types of tools.

DPCL is also important in a different sense -- it provides an abstract layer for tool development that is machine independent. This means DPCL can make it possible for tools to be truly available across the industry. Major efforts, such as the PTools Forum and HPDF, are under way to standardize available programming tools across the industry. By defining standard tools and interfaces these organizations hope to make the transition from one vender system to another easier for developers, especially developers that must concurrently develop and maintain software for multiple vender machines. Progress is slow, however, in part because each vender must provide their own implementation of the tool. These efforts show there is a strong desire within the industry, among labs, customers and some hardware venders themselves, to provide a common set of tools across the industry.

Making tools available across multiple platforms is generally a huge undertaking. The use of standard languages and graphic interface libraries reduces the required effort considerably. Building the tools on top of DPCL reduces that effort further. Once DPCL is available on a given platform, completing the port of instrumentation is relatively simple. The cost of porting DPCL to the new machine can also be amortized over many tools. Thus DPCL can provide a low cost means for making a wider variety of tools available across the industry.

Easy porting of tools to multiple platforms provides additional benefits. When the cost is low to create and maintain tools across multiple platforms, ISVs have motivation to support a wider variety of development and support tools across those platforms. This reduces the development costs that must be shouldered by hardware venders while simultaneously making the venders' machines more attractive.

The reduced cost of developing tools also translates into greater innovation. The level of effort and expertise is reduced, which allows universities and labs to experiment with more speculative analysis techniques and tools ideas while maintaining a limited budget. Relatively small investment in a proof-of-concept prototype is required to determine whether the idea has validity before investing significant effort in full scale tool development. When the cost of developing instrumentation is high, the cost of trying out an uncertain idea may be too great. In other words, DPCL can be used as an engine of innovation.

1.2 DPCL Concepts and Components

Described briefly, DPCL is implemented as a distributed, asynchronous system, where the end-user-tool is a client process that requests services, through library calls, from special daemons. Daemon management is performed by the underlying system and not by the end-user-tool. A tool calls member functions in the DPCL class library to request a desired service. The library then forms the request into a message that is sent to a daemon process. The results of a service request can be an acknowledgement or negative acknowledgement that the request was performed. In some cases, such as instrumentation (probe) activation, a service request may also result in a stream of data messages being sent to the client.

In each case a message is received by the DPCL message system and transferred as an argument to a callback function. Messages containing system information are processed by DPCL system callbacks. Messages containing positive or negative acknowledgments are processed by user or system callbacks, depending on how the service was requested. Messages containing data sent by instrumentation probes are processed by user data callbacks.

Systems that rely on processing events with callback functions are called *asynchronous systems.* In DPCL the events to be processed are messages received from a daemon. Asynchronous systems are often used in client/server applications. DPCL clients are the end-user-tools built upon DPCL, while the daemons are the service providers or servers. Daemon services include connecting to an application process, installing and activating instrumentation, deactivating and removing instrumentation, and a host of other services. Service requests may be asynchronous or pseudo-synchronous. Asynchronous service requests explicitly provide user callback functions to process acknowledgments.

The request function returns immediately upon issuing the request and does not wait for the request to succeed or fail. Pseudo-synchronous service requests are called *blocking* requests or functions, and differ from purely synchronous requests in that the blocking requests do not return control to the caller until the request has succeeded or failed, and therefore do not require a callback function to process the acknowledgement, positive or negative.

Instrumentation is defined by the end-user-tool using a combination of *probe expressions* and *probe modules*. Probe expressions are *abstract syntax trees* that represent integer values and variables, connected by the usual arithmetic, logical, and bit-wise operators. A small amount of conditional control flow is also supported, as are function calls and some pointer operations. Probe modules are collections of functions, written in a standard language such as C and compiled into object files, that are loaded into an application and called from within a probe expression.

Source objects, also known as source trees, are data structures that reflect, to some degree of granularity, the structure of the original source code that was used to create the executable program. Information to construct the source objects is gathered from both the executable file and the executing program image in memory. Through the process of compilation, much of the information needed to construct a detailed source representation is lost. This is especially true when higher levels of compiler optimization are used, because they use techniques that move significant amounts of code around. Examples of these techniques are common sub-expression elimination, removal of invariant code from loops, loop splitting, loop fusion and loop fission.

Source objects are organized much as a compiler or linker sees a program. The highest level in the structure is the program itself. Beneath the program is a collection of modules, that represent the source files in the program. Within each module is a collection of functions and data. The functions contain data and various forms of nested blocks and statements. Functions also contain *instrumentation points*. Instrumentation points are locations within executable code, within functions, where instrumentation can be placed. Depending on the type of point, instrumentation may be placed before the point, or after, or both. Probe expressions are the type of instrumentation that may be placed at an instrumentation point.

There are two other types of instrumentation that may be employed. The simplest is *one-shot*, or *inferior remote procedure call* instrumentation, also known as IRPCs. IRPCs are executed immediately upon request, whatever the application happens to be doing. Upon completion of the IRPC the instrumentation is immediately removed. Probe expressions are the type of instrumentation used with IRPCs.

The remaining type of instrumentation is activated periodically, upon expiration of an interval timer. This type of instrumentation is called a *phase*. The period of the interval timer is set by the end-user-tool when the phase is created. It may also be changed at a later time. Phases use functions as the type of instrumentation executed when the phase interval timer expires. The functions to be used are established when the phase is created and cannot be changed later. Phases are also associated with instances of probe data.

When a phase interval timer expires three functions are executed for the phase. The first function is executed once each time the interval timer expires, and can be used to set up any information that is useful to this execution of the phase. This is called the *phase begin function*. When the phase begin function completes, a second function is executed once for each piece of probe data associated with the phase. This function is called the *phase iteration function*. When the phase iteration function completes for the last piece of data associated with the phase, a *phase end function* is executed to clean up any details that might be needed.

Instrumentation may communicate with the client tool by explicitly sending messages. Messages are treated as unstructured byte streams, the format of which is to be interpreted and understood by the end-user-tool. DPCL does not understand the contents of the message. Each piece of instrumentation has a unique callback to receive data. In order to send a message a probe must be able to specify the callback and client to whom the message is intended. This information is contained in the *message handle*. The message handle is an opaque data object, stored with the instrumentation in the application process, that contains the information necessary to direct a message to the correct client and callback function for the message.

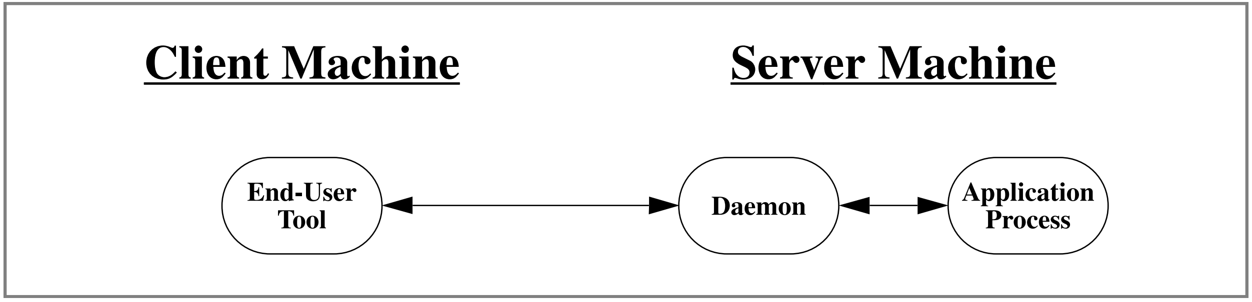
A typical sequence of operations a tool might use is:

1. connect to the process or application,
2. expand the source tree,
3. find the locations where instrumentation is desired,
4. set up any phases if they are to be used,
5. allocate data storage for the instrumentation data,
6. install and activate the instrumentation,
7. gather the data and process it using data callback functions,
8. remove the data and instrumentation,
9. and finally, disconnect from the application.

1.3 DPCL Features

End-user-tools built upon DPCL are able to instrument applications that use a variety of serial and parallel programming models. The simplest connection is to a serial application, and works as shown in Figure 1 on page 6. The end-user-tool establishes a connection to the target application process through a daemon. The daemon resides on the same machine as the target process, and is able to hook up to it like a debugger. Once the connection is established, the tool may send its requests to the daemon, who acts on those requests.

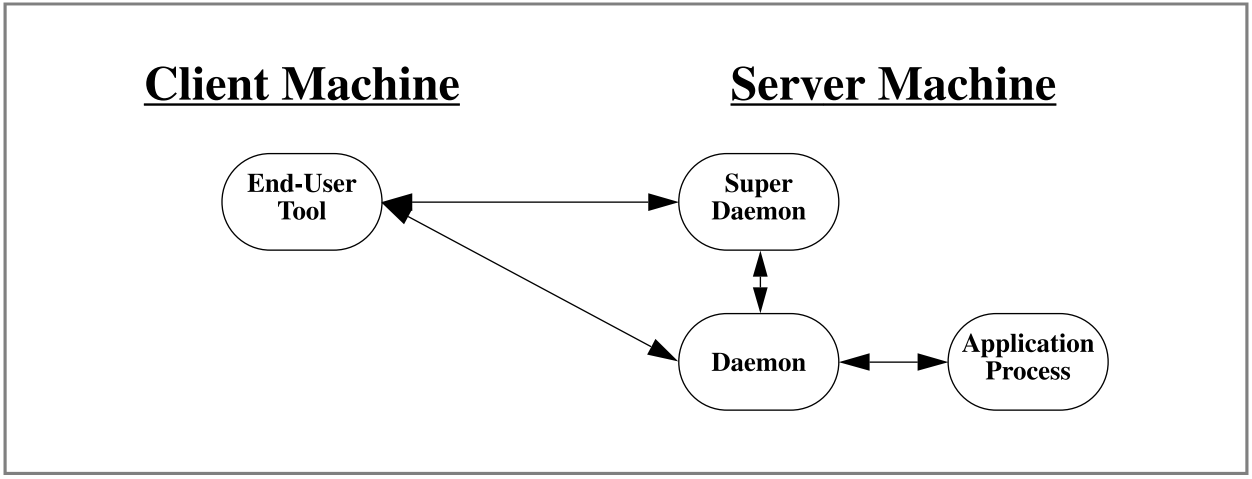
FIGURE 1 A Tool Connects to a Program through a Daemon



Daemons are independent processes that act in behalf of a client process. DPCL daemons are not persistent, in the sense that they do not exist when there is no connection to be maintained. They must be created at times when connections are established. However, daemons are shared in the sense that when multiple clients wish to connect to the same application, they do so by connecting through the same daemon. A single daemon may also serve as an agent in connecting to multiple application processes. This coordination is managed through the use of a *super daemon*, or a daemon that creates other daemons.

When a client wishes to create a connection to a process on a given server, the DPCL library creates a super daemon on the server. This is illustrated in Figure 2 on page 6. The new super daemon checks for the existence of other super daemons on the server. If an older super daemon exists, the new super daemon transfers the connection between the new super daemon and the client over to the older super daemon and exits. The remaining super daemon then decides whether to use a new daemon or an existing one. If a new daemon is to be used, it creates the new daemon and transfers the client connection to the new daemon. Otherwise it transfers the client connection to an existing daemon. The decision to create a new daemon or use an existing one is based on issues of security as well as whether other daemons currently exist.

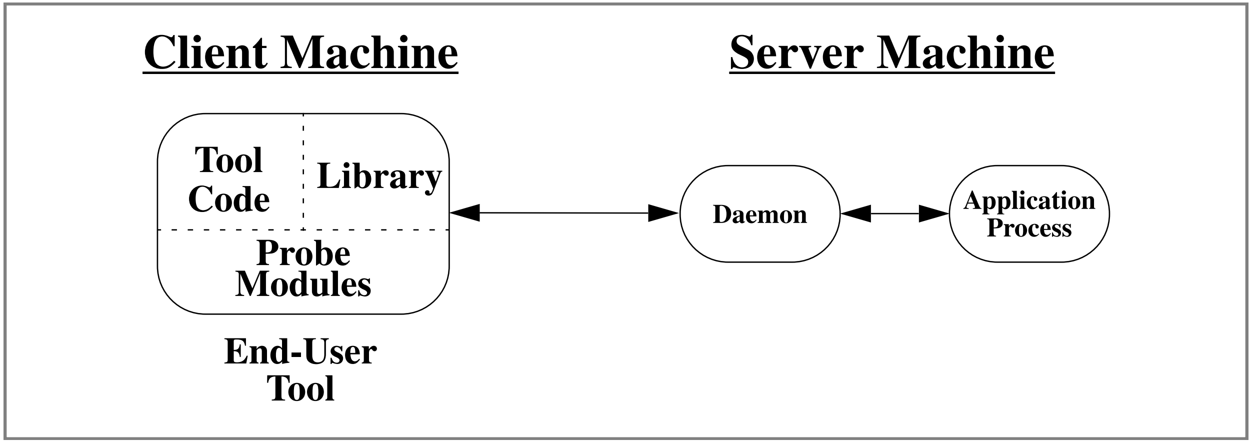
FIGURE 2 Connections to Daemons Are Made through Super Daemons



Security within the system is maintained in several ways. First, when a new connection is established between a client and super daemon, the super daemon requires that the client be authenticated using the most robust means available. When DCE is in use on the server, DCE authentication is required. Otherwise an internal authentication procedure is used. Second, in order for the super daemon to be able to create appropriate daemons it must have super user privileges. For this reason, the super daemon is not allowed to access or modify any user applications. Third, any daemons the super daemon creates must first change their ownership from the super user to the user ID of the client. Thus the daemon has only those privileges that would be granted by the system to the client tool if it were running locally on the daemon.

Client tools are not responsible for connecting directly with a super daemon. Rather, the tool requests services from the DPCL library, as illustrated in Figure 3 on page 7, and the library handles the details of establishing the connection. Furthermore, the library also handles all of the details of communicating between the tool and daemon, including details such as making sure the correct message formats are used, and that the messages are delivered reliably. When it is appropriate that the tool process a message, such as a service request acknowledgment from a daemon or a data message from a probe, the library handles the details of message reception and hands the message over to a callback function supplied by the tool for the purpose of processing that kind of message.

FIGURE 3 A Tool Communicates with a Daemon through the Library



After a connection is established, a tool is free to install instrumentation into the application. A tool does so by allocating data, forming probe e
to be called by probe expressions) into the application, as illustrated in Figure 4 on page 8. As mentioned before, the tool submits its requests via the library, and the library handles all the details of communication. Once this preparatory work is done, the instrumentation probes can be installed within the application and activated. Activated instrumentation is free to send messages containing relevant data back to the client tool, as shown in Figure 5 on page 8.

FIGURE 4 An End-User Tool Installs Probes in an Application

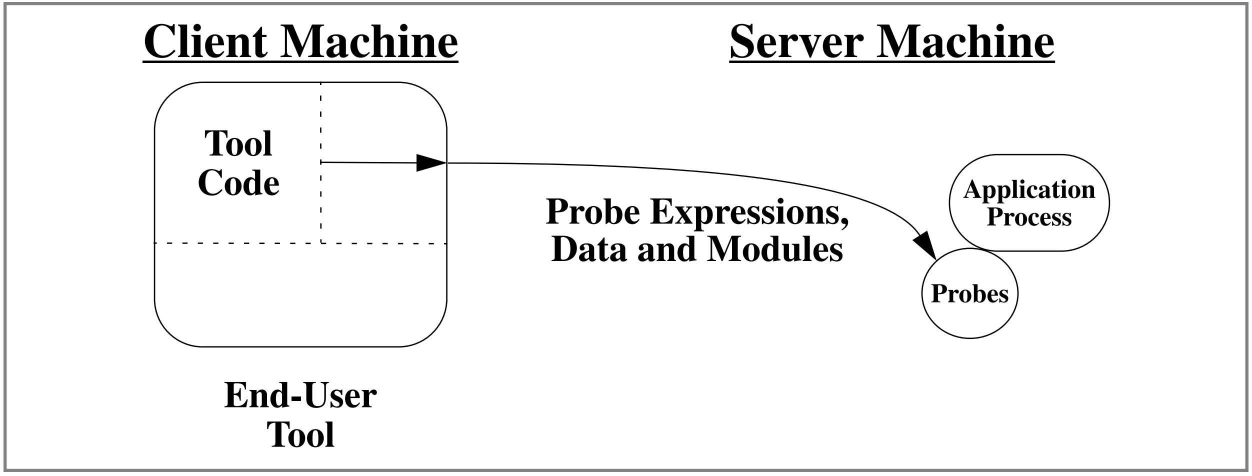
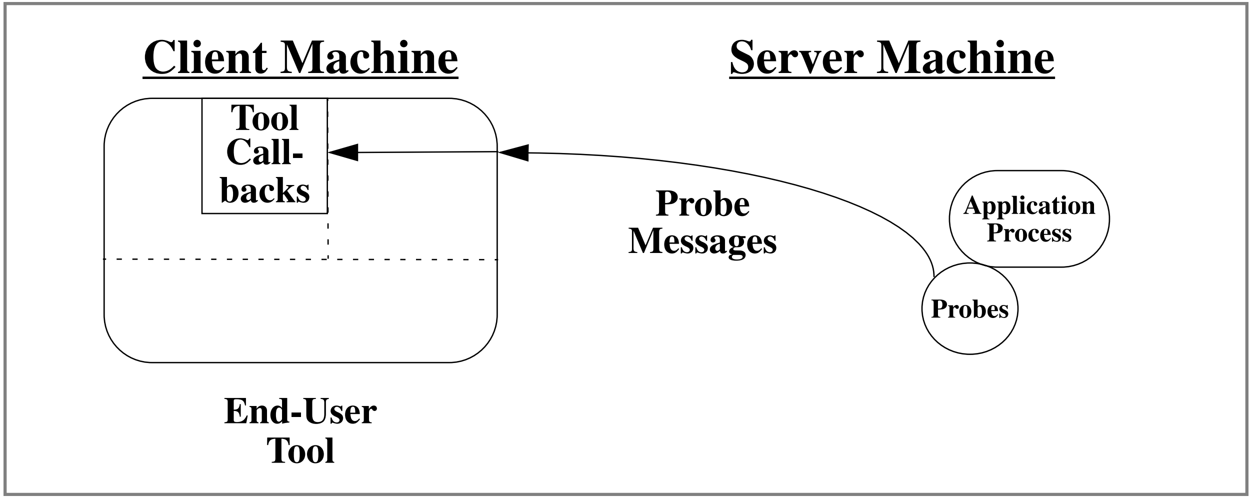
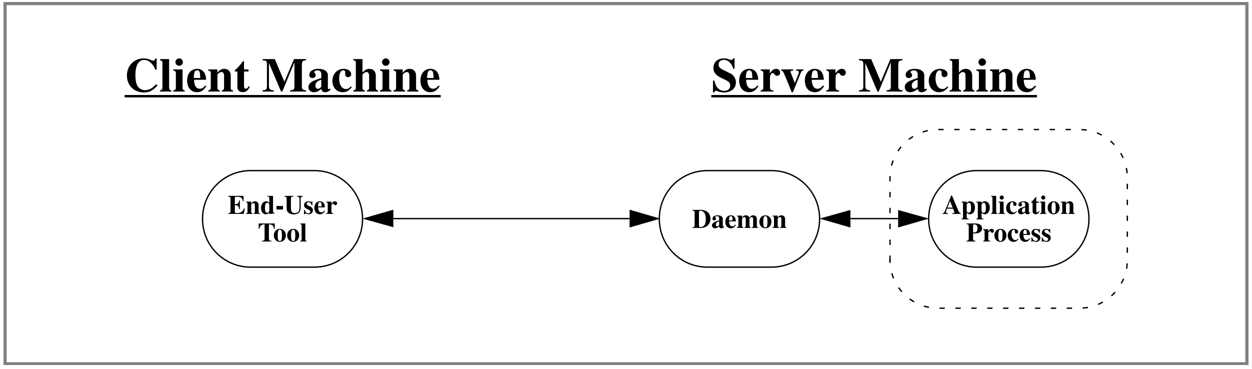


FIGURE 5 Probes Send Messages to the Client



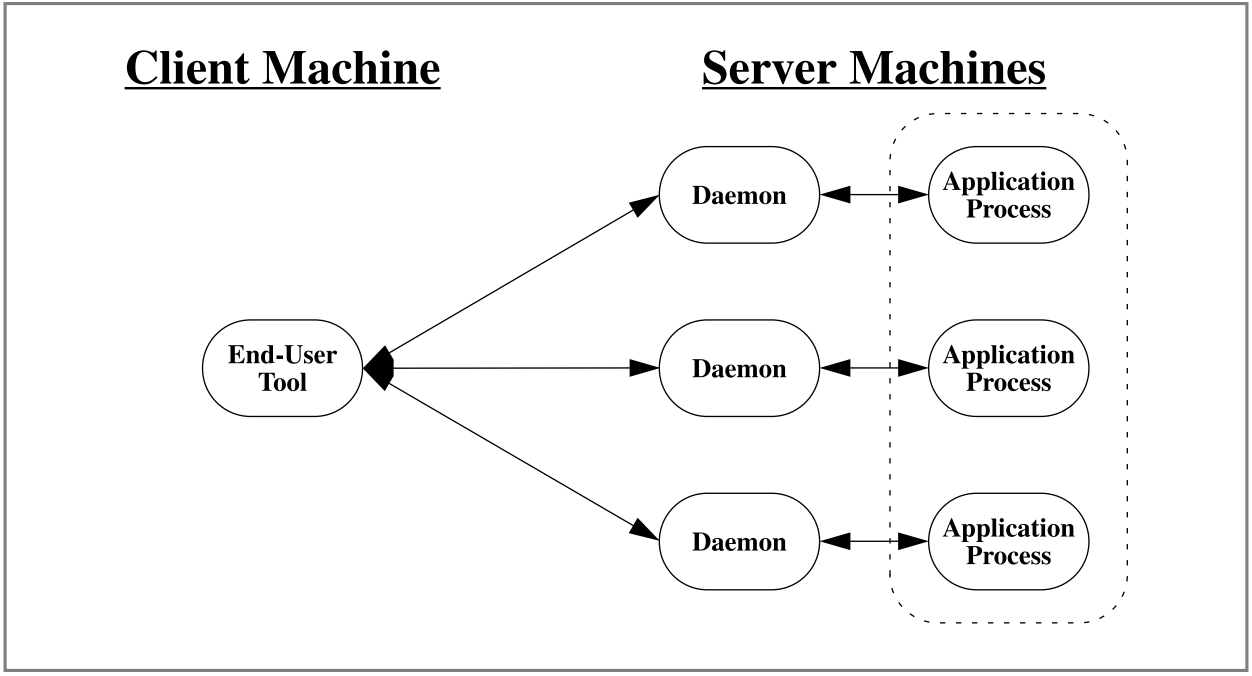
As mentioned before, DPCL is capable of managing the instrumentation of multiple applications and multiple clients. The simplest situation is shown in Figure 6 on page 9, where there is exactly one client and one application process. Serial programs like this require no special considerations from DPCL. The dotted box in the figure represents the application. This type of connection requires only a single daemon, and the daemon must manage communication from a single client.

FIGURE 6 A Tool May Connect to a Serial Program



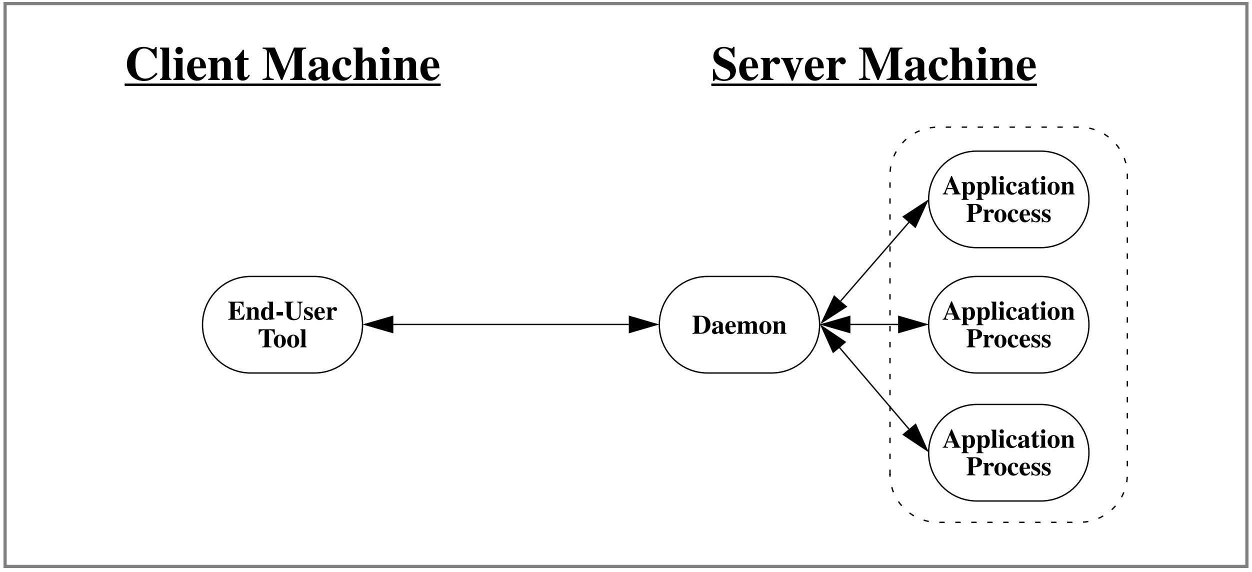
Parallel programs introduce an additional level of complexity, whether they are multiple processes on multiple machines, as illustrated in Figure 7 on page 9 or multiple processes on a single machine, as illustrated in Figure 8 on page 10. In the former case the client must manage the additional daemons, while the daemons have the simpler connections. In the latter case, the client maintains a connection with a single daemon, but both client and daemon must deal with the complexity of multiple application processes.

FIGURE 7 A Tool May Connect to a Parallel Program



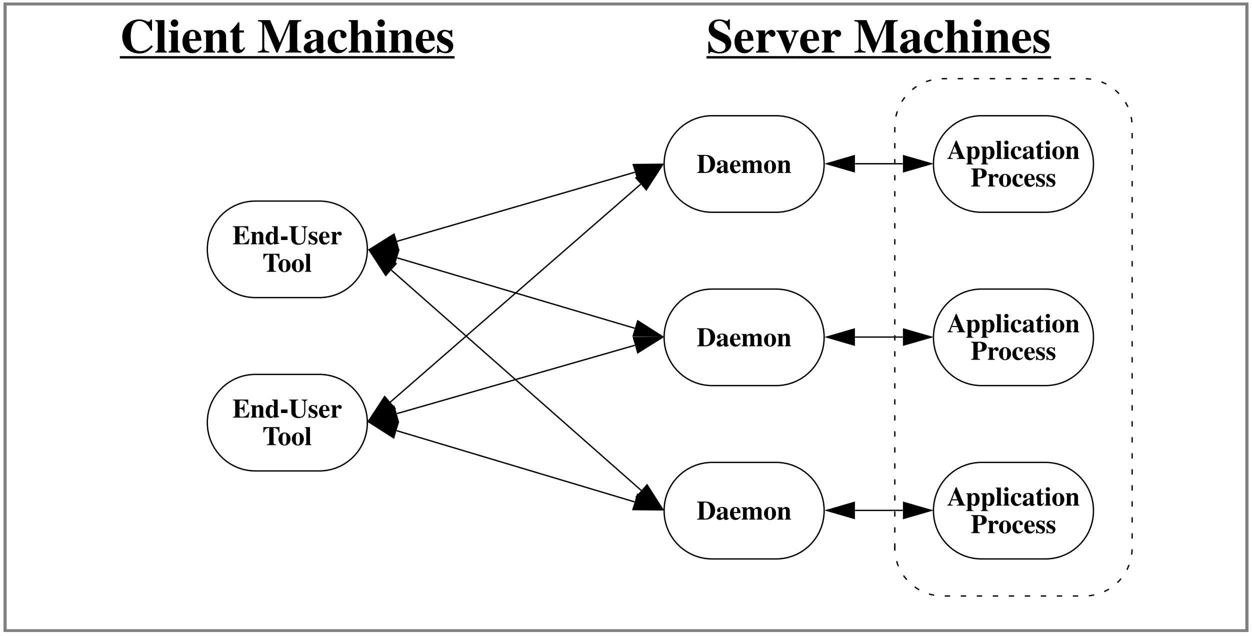
Something worthy of note is that while a daemon may connect to multiple processes within the same application, as described before, there is nothing about the processes that require they all be in the same application. DPCL library code on the client may track processes and group them together to form an application, but this is a convenience for the client and not required for daemon operation.

FIGURE 8 A Daemon May Connect to Multiple Processes



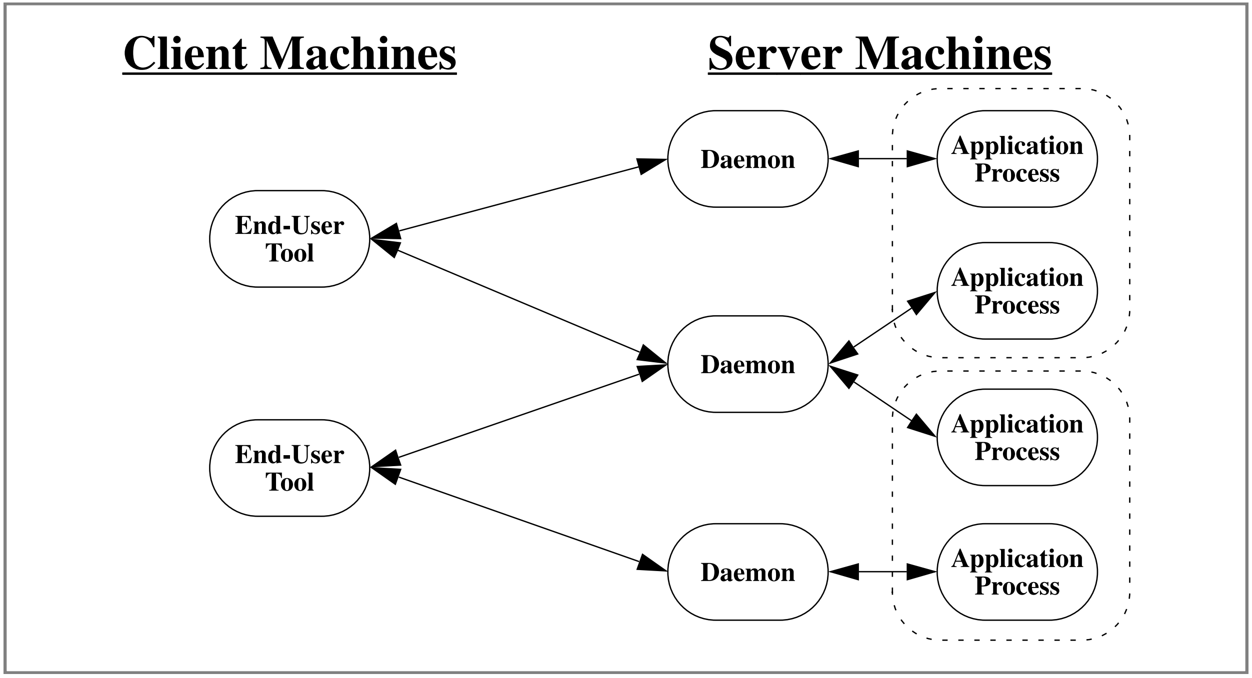
Daemons support not only connections to multiple processes, but also connections to multiple clients. Figure 9 on page 10 illustrates two clients connected to the same parallel application. The two applications need not be aware of each other. DPCL does not provide explicit facilities at this time for two tools to be aware of each other nor to communicate except with daemons. However, this could be a topic for future extensions.

FIGURE 9 Multiple Tools May Connect to a Single Program



Finally, for completeness, DPCL also supports sharing of daemons when multiple clients and multiple applications are involved. This is illustrated in Figure 10 on page 11. In this situation the shared daemon may not respond as quickly to requests or in forwarding data because of the extra demands placed on it by serving two clients. Other factors, such as varying work loads across systems can also affect performance in this manner.

FIGURE 10 Multiple Tools May Share a Daemon



2.0 Introduction to DPCL Concepts

This chapter provides a brief introduction to individual concepts used within DPCL.

2.1 Asynchronous Programming

Asynchronous programming is a style of programming where the major activities in a program involve acting upon events that may arrive at an undetermined time, and in an undetermined order. The system recognizes the occurrence of an event and reports the event to the program. The program has event handlers set up to process those events when they arrive. One example of a commonly used asynchronous system is the Unix signal facility. A Unix process may receive a signal from another process, from itself, or from the operating system at any time. The event to be acted upon, then, is the signal. The operating system is the mechanism that transfers the signal to the program and activates the signal handler, which in this case is the event handler. While this aspect of a program that uses signals matches the asynchronous programming model, such programs also typically use signals as indicators of exceptional circumstances rather than the norm.

A better example might be something like a Motif or other X-Windows application. Mouse movements, object selection, and keyboard strokes are the typical events in these programs. The program first sets up whatever relevant program data structures and event handlers there might be, then enters a main event loop. The main event loop recognizes incoming events and makes a call to the appropriate event handler. Event handlers in this system are called *callback functions*, or just *callbacks*.

DPCL works very similarly to Motif, at least in some ways. End-user-tools built upon DPCL initialize data and enter a main event loop. The main ev
is detected, a dispatch routine for that file descriptor is called. The dispatch routine understands the protocol for that file descriptor
a daemon, unstructured text from a keyboard, screen events from a GUI, or whatever it might be. When the event is a structured message from a daemon, the dispatch routine reads the message from the file descriptor, looks up the callback function associated with the incoming message, and executes the callback function.

The end-user-tool may supply an appropriate dispatch routine when adding a file descriptor to the list of file descriptors DPCL listens to. The end-user-tool does not get directly involved in messages passed between client and daemon, except to provide some of the client callback functions that are activated when certain messages arrive.

2.1.1 Callbacks

Callback functions in DPCL are functions that are called when certain messages arrive from a daemon. A message may represent an acknowledgement of a service request being completed successfully, or a service request failing to complete. Or it may represent data being sent from instrumentation within the application to the client for processing. All callbacks have the same function prototype, or type signature, regardless of the message type that activates it. The function prototype for callbacks is:

```
void callback (
     GCBSysType sys,     // system data structure
     GCBTagType tag,     // user-supplied tag value
     GCBObjType obj,     // object that registers the callback
     GCBMsgType msg)     // message that invokes the callback
```

The callback function has four parameters. The `sys` parameter is a data structure that includes basic system information about the message. Values in the structure are: the socket or file descriptor from which the message was received; a message key or type value that represents the protocol or purpose behind the message; and the size of the message in 8-bit bytes. The structure is defined as

```
struct GCBSysType {
     int msg_socket;    // socket over which msg was received
     int msg_type;      // message type
     int msg_size;      // size of the message sent
}
```

The message type is used as a key value to identify the callback function to be executed. This information is provided by the system.

The `tag` parameter is a value, large enough to contain a pointer, that is supplied by the end-user-tool at the time the service is requested and the callback is identified. This allows callback functions to be used for more than one purpose when desired. For example, the tag may be used as a pointer to relevant data that changes from one usage of the callback to another. It can be used in many different ways, but it is entirely up to the end-user-tool to decide. DPCL merely records its value when the callback is registered, and passes the value to the callback function when the callback is executed.

One piece of data that is often useful is the object being used to request a service at the time the callback is registered. For example, "`Process p; ... p.connect(cb, tag);`" uses the data object `p`, of type `Process`, to request a service. The requested service in this example is to connect the client to a physical process. The callback function might find it useful to know which process data object requested the connection when it decides what to do with the success or failure of the request. As an example it might use this information to display informative error messages when requests fail. The object used in a service request is also known as the *invoking object*. A pointer to the invoking object is stored in the parameter `obj`.

The actual content of the message is presented as a raw byte stream. A pointer to the message content is given in the parameter `msg`. The number of bytes in the message is stored in `sys.msg_size`. When the message is the acknowledgement of a service request, the message format is defined by the system. When the message is a data message it is generated by the end-user-tool instrumentation running within the application. The format is determined by the instrumentation, since the instrumentation determines what to place in the `Ais_send` buffer. `Ais_send` sends the message.

2.1.2 Event Recognition and Dispatch

DPCL recognizes two kinds of events: file descriptor input events and signal events. DPCL programs sit in a main loop that waits for events to occur. Each time an event occurs a dispatch routine, or handler, for that class of event is dispatched to deal appropriately with recognizing the event. In some cases a callback function associated with the event is also called.

Signal events are the simplest to process. A tool may register a signal event dispatch routine to deal with certain Unix™ signals. When a signal is raised, an internal signal handler catches the signal and schedules the signal event dispatch routine to be executed. The internal signal handler releases control to the normal tool environment, where the signal event dispatch routine is then executed. Because the signal event dispatch routine is executed within the normal environment, the application stack is used for function calls rather than the signal stack.

Signal event handling routines are also exempt from the concerns of typical signal handlers, such as executing a function that uses a data structure caught in an inconsistent state. Subroutines such as `malloc` cannot be safely executed within a signal handler because the signal that caused the handler to be executed may have interrupted the application while it was in the process of executing a call to `malloc`. This would mean the second `malloc` invocation might catch its internal data structures in an inconsistent state, which could potentially cause an error to occur and the program to terminate abnormally. Signal event handlers are not invoked within the internal signal handler, so calls to routines like `malloc` have an opportunity to restore consistency to their internal data structures and return normally before the signal event dispatch routine is called.

DPCL only allows one signal event dispatch routine to be registered for each signal. The old signal event dispatch routine must be removed before a new signal event dispatch routine may be installed. Signals are not stacked, either. If two signals of the same type are received in rapid succession before the signal event handler for the first signal is allowed to execute, it is treated as only one signal.

File descriptor input events are the most common events to occur. They can be divided into two major categories, namely input events that represent communication between the client and daemon, and those that represent input from a user-tool-specified input file descriptor. When input arrives on a file descriptor the system recognizes that an event has occurred, but it does not yet know what type of event has occurred. How one interprets the input to determine the type of event depends on the source of the event. If the input is from a daemon, certain information is guaranteed to be part of the input stream -- information that identifies the type of event that occurred and the callback function, if any, associated with it. If the input is from any other source, DPCL does not know how to determine the action to take in order to process the input.

For this reason DPCL has dispatch routines associated with every file descriptor it watches. The dispatch routine is responsible for interpreting the input stream and breaking it into individual events. For file descriptors connected to daemons, a special dispatch routine is used that interprets the input stream as a stream of structured messages, and invokes the appropriate callback function for each message. Each message represents an individual event to the system.

User-tools are also able to register file descriptors to be watched by the system. That allows other types of input to be handled asynchronously as well. For example, a user-tool may wish to incorporate keyboard input into its processing. The tool may register file descriptor 0 and a dispatch routine with the system. File descriptor 0 is the default descriptor for standard input, which is usually keyboard input. A different file descriptor could be substituted for 0 assuming it is opened to a suitable keyboard device. The dispatch routine that is registered with the file descriptor is the routine that will handle all input processing for that file descriptor. Each time the user types input at the keyboard, DPCL will recognize that input has been supplied and will invoke the registered dispatch routine. After the input has been processed the dispatch routine must return to the caller so the system can continue to monitor other input streams and act on them as well.

Dispatch routines accept a single value, an integer, as its input argument. The value represents the file descriptor or signal number. The function prototype is

```
int dispatch (int sig_or_fd)
```

The function return value indicates whether the socket has reached the end-of-file. The dispatch function must return a -1 when an end-of-file is reached. Otherwise it should return 0. Signal event dispatch routines do not reach an end-of-file condition, and the return value is ignored.

2.1.3 Asynchronous and Semi-Asynchronous Programming

Event driven programming may take on either of two forms, or perhaps a combination of both. The first is asynchronous programming. Purely asynchronous programming sets up s pecial functions, known as callbacks, that are activated when certain events occur. In DPCL the events are often acknowledgement responses to service requests sent to daemons on other hosts. When the service request is sent a callback is set up so it can be activated when the acknowledgement arrives. There is only one acknowledgement, so when it arrives the callback can be removed. The key issues are that the callback is registered, the service request is sent, and control returns immediately to the caller before the service is granted or denied. The registered callback function takes action when the acknowledgement arrives.

In contrast, semi-asynchronous programming, or pseudo-asynchronous programming, does not register a callback, and does not return control until the service request has been granted or denied. These functions are also called blocking functions or blocking service requests, because they block the execution of the caller until the service request has been processed. The underlying system must continue to process events and therefore it must continue to invoke callback functions that may make additional service requests even though the execution of the caller is temporarily blocked. The advantage to the user is that series of services can be requested in a style that is quite similar to procedural programming, what most programmers are accustomed to using.

The naming convention for blocking service requests is to prefix the letter “b” to the name of the asynchronous service request routine. For example, `connect` is an asynchronous service request routine that requests a connection be established to a process. Because it is asynchronous it requires that a callback function be provided to take action when the acknowledgement arrives. The corresponding blocking request is `bconnect`. Since it is a blocking service request that will not return control to the caller until the request has succeeded or failed, the action to be taken may follow the blocking request call in the calling code.

2.2 Setting Up a Tool to Use DPCL

As was mentioned in an earlier section, a typical sequence of operations a tool might use is:

1. connect to the process or application,
2. expand the source tree,
3. find the locations where instrumentation is desired,
4. set up any phases if they are to be used,
5. allocate data storage for the instrumentation data,
6. install and activate the instrumentation,
7. gather the data and process it using data callback functions,
8. remove the data and instrumentation,
9. and finally, disconnect from the application.

Each of these steps will be examined in detail in later sections. Before any of that can take place the end-user-tool must be set up to run in an asynchronous environment. The minimum requirements are quite simple. The tool must first register all of the system callback functions with the function `Ais_initialize`, then it must enter the main event loop with the function `Ais_main_loop`. This is done as follows.

```
main(void)
{
     Ais_initialize();
     Ais_main_loop();
}
```

Of course the tool may also do whatever additional initialization is desired for proper tool initialization, but it should generally be done before the call to `Ais_main_loop`. The tool will not exit this function until asynchronous operation of the tool is complete.

2.3 Application and Process Management

2.3.1 Working with Single-Process Programs

Serial applications are programs that use only a single Unix process. DPCL allows two methods of access to serial applications. Since there is only a single process to manage, one may use the `Process` class directly. This technique is described in this section. On the other hand, it may also be thought of as an application that contains a single process, so the `Application` class may also be used. The latter technique is described in Section 2.3.1.3, “Disconnect from a Running Process” on page 19.

All code examples in this section require the following header file.

```
#include <Process.h>
```

2.3.1.1 Create a Process Class Object

The `Process` class is a DPCL data type that describes the attributes needed for tracking and manipulating Unix processes. `Process` data objects may be initialized using one of several functions. One may use a constructor, a copy constructor, or the assignment operator. The default constructor creates the necessary storage but does not initialize the object to a valid application process. The following examples illustrate different ways the data objects may be initialized to valid application processes.

The following example illustrates the simplest method of creating a `Process` object that represents an application process. In this example the process is currently executing on a host machine with the IP host name of “myhost.xyz.edu”. The process ID is 12345. Note that no connection is as yet established with the process. It is purely a local data structure that resides within the end-user-tool. No attempt is made to determine whether there is actually a host with that name, or a process with that ID running on the indicated host.

```
Process p(“myhost.xyz.edu”, 12345);
```

This next example is identical to the previous with the following exception. Each process also has a task identifier that may be used for the purpose of tracking its rank within a parallel application. When the task identifier is not explicitly provided it is set to zero. When provided, the last parameter on the constructor determines the value to which it is set.

```
#include <Process.h>

Process p(“myhost.xyz.edu”, 12345, 3);
```

The above examples use non-default constructors directly to assign values to the object. In the next example a non-default constructor and a copy constructor is used to assign values to the object.

```
Process p = Process(“myhost.xyz.edu”, 12345);
```

The next example illustrates the use of a default constructor and an assignment operator to assign values to an object. Of course the right-hand-side of the assignment operator may be any valid object and need not strictly be a temporary object created directly by a constructor, as is illustrated here.

```
Process p;

p = Process(“myhost.xyz.edu”, 12345);
```

2.3.1.2 Connect to a Running Process

In order to perform any subsequent operation on a process, the end-user-tool must first establish a connection to the process. This may be done with a blocking service request, or with a non-blocking service request. The simplest approach is to use the `Process::bconnect` member function, which is a blocking service request. The service being requested is that DPCL establish a connection to the indicated process. This function does not return control to the caller until the requested service has either succeeded or failed.

```
p.bconnect();
```

A second method of connecting to the process uses a non-blocking service request. In this case the request is sent but control is returned immediately to the caller. The function does not wait until the request succeeds or fails. Instead, a callback function is provided that will process the acknowledgement of success or failure when it arrives.

```
int tag = ...;
void cb(GCBSysType s,GCBTagType t,GCBObjType o,GCBMsgType m);
p.connect(cb, (GCBTagType)tag);
```

Each of these examples requests a connection to the process indicated in the `Process` object `p`. The service request may fail if the host does not exist or is unreachable, if the requested process identifier does not correspond to an existing process on that host, or if the process exists but the user does not have the necessary access privileges to connect to the process.

If the request succeeds the end-user-tool may request additional services that affect the process, such as installing and activating probes.

2.3.1.3 Disconnect from a Running Process

When an end-user-tool has finished with a process it may release the resources associated with the connection by disconnecting from the process. This may be done with a blocking or a non-blocking service request. The simplest approach is to use the `Process::bdisconnect` member function, which is a blocking service request. The service being requested is that DPCL release a connection to the indicated process. This function does not return control to the caller until the requested service has either succeeded or failed.

```
p.bdisconnect();
```

A second method of disconnecting from the process uses a non-blocking service request. In this case the request is sent but control is returned immediately to the caller. The function does not wait until the request succeeds or fails. Instead, a callback function is provided that will process the acknowledgement of success or failure when it arrives.

```
int tag = ...;
void cb(GCBSysType s,GCBTagType t,GCBObjType o,GCBMsgType m);
p.connect(cb, (GCBTagType)tag);
```

Each of these examples requests that the end-user-tool disconnect from the process indicated in the `Process` object `p`. The service request may fail if process is not connected.

2.3.2 Working with Multiple-Process Programs

Multiple process programs may be handled in two ways. They may be treated as a loosely connected group of processes, or the processes may be explicitly grouped together in an `Application` class object. This section addresses the latter approach.

All code examples in this section require the following header file.

```
#include <Application.h>
```

2.3.2.1 Create an Application Class Object

There are two constructors for the `Application` class -- the default constructor and the copy constructor. Initially an `Application` object is created as an empty application. Application processes represented by objects of type `Process` may be added to or removed from the `Application` object as needed. `Application` objects are indexed collections of `Process` objects, so individual processes within an `Application` object may be referenced by knowing its index.

```
Application app1;

Process p(“kim”, 1080);

app1.add_process(p);

Application app2 = app1;
```

In the above example `app1` is created initially as an empty application. A process is then added to the application. Note that so far there has been no attempt to verify that the process actually exists nor to connect to the process. `app2` is created as a copy of `app1` using the copy constructor.

2.3.2.2 Adding and Removing Processes from an Application Class Object

`Process` objects may be added and removed from an `Application` object at any time. For example, we might continue the example from the previous section with

```
Process q(“ted”, 7693);

app1.add_process(q);
```

A word of caution is in order here. As mentioned before, `Application` objects are indexed collections of `Process` objects. The first valid index is always zero and the last valid index is one less than the value returned by the function `get_count`. The index map is guaranteed to be dense, that is, there are no gaps in the index range. Each time that a process is removed from an application the mapping of indexes to processes changes. When a process is added to an application it is placed at the next available index at the end of the index range. When a process is removed all processes with indexes higher than the removed process are shifted downwards to close the gap.

2.3.2.3 Obtain a Parallel Application Process List

When the application of interest is an IBM Parallel Operating Environment (POE) application, a faster method exists for loading the processes into the application. If one knows the parent POE process ID one may create a `PoeApp` and load the processes automatically. The `PoeApp` class is derived from the `Application` class so all functions available in `Application` are also available in `PoeApp`. The process list may be loaded using either a blocking or a non-blocking service request. The blocking service request is shown in the next example.

```
app1.bread_config("john", 7177);
```

This example asserts that the parent POE process of an interesting application may be found on a machine with a host name of "john" and it has a process ID of 7177. This request queries the system for the list of processes associated with that particular POE application. That information is then loaded into the `PoeApp` object `app1`. Since it is a blocking request, control does not return to the caller until the request has either succeeded or failed.

A second method of obtaining the POE application configuration uses a non-blocking service request. In this case the request is sent but control is returned immediately to the caller. The function does not wait until the request succeeds or fails. Instead, a callback function is provided that will process the acknowledgement of success or failure when it arrives.

```
int tag = ...;

void cb(GCBSysType s,GCBTagType t,GCBObjType o,GCBMsgType m);

app1.read_config("john", 7177, cb, (GCBTagType)tag);
```

Each of these examples requests a process list be loaded into the `PoeApp` object. The service request may fail if the host does not exist or is unreachable, if the requested process identifier does not correspond to an existing process on that host, if the process exists but the user does not have the necessary access privileges to access the process information, or if it exists but it is not the root process of a POE application.

2.3.2.4 Connect to a Running Application

In order to perform any subsequent operation on an application, the end-user-tool must first establish a connection to all processes within the application. This may be done with a blocking or a non-blocking service request. The simplest approach is to use the `Application::bconnect` member function, which is a blocking service request. The service being requested is that DPCL establish a connection to all processes within the indicated application. This function does not return control to the caller until the requested service has either succeeded or failed.

```
app1.bconnect();
```

A second method of connecting to the process uses a non-blocking service request. In this case the request is sent but control is returned immediately to the caller. The function does not wait until the request succeeds or fails. Instead, a callback function is provided that will process the acknowledgement of success or failure when it arrives.

```
int tag = ...;
void cb(GCBSysType s,GCBTagType t,GCBObjType o,GCBMsgType m);
app1.connect(cb, (GCBTagType)tag);
```

Each of these examples requests a connection to the processes included in the `Application` object `app1`. The service request may fail if the host does not exist or is unreachable, if the requested process identifier does not correspond to an existing process on that host, or if the process exists but the user does not have the necessary access privileges to connect to the process.

If the request succeeds the end-user-tool may request additional services that affect the processes, such as installing and activating probes.

2.3.2.5 Disconnect from an Application

When an end-user-tool has finished with an application it may release the resources associated with the connection by disconnecting from the application. This may be done with a blocking or a non-blocking service request. The simplest approach is to use the `Application::bdisconnect` member function, which is a blocking service request. The service being requested is that DPCL release a connection to all processes within the indicated application. This function does not return control to the caller until the requested service has either succeeded or failed.

```
app1.bdisconnect();
```

A second method of disconnecting from the application uses a non-blocking service request. In this case the request is sent but control is returned immediately to the caller. The function does not wait until the request succeeds or fails. Instead, a callback function is provided that will process the acknowledgement of success or failure when it arrives.

```
int tag = ...;
void cb(GCBSysType s,GCBTagType t,GCBObjType o,GCBMsgType m);
app1.disconnect(cb, (GCBTagType)tag);
```

Each of these examples requests that the end-user-tool disconnect from the process indicated in the `Process` object `p`. The service request may fail if process is not connected.

2.3.3 Creating New Application Programs

To be provided in a later release.

2.3.3.1 Create an Application

2.3.3.2 Start a New Application

2.3.3.3 Restart an Application

2.3.4 Application Control Functions

To be provided in a later release.

2.3.4.1 Suspend an Application

2.3.4.2 Resume an Application

2.3.4.3 Signal an Application

2.3.4.4 Attach to an Existing Application

2.3.4.5 Detach from an Application

2.3.4.6 Terminate an Application

2.3.5 Application Memory Functions

To be provided in a later release.

2.3.5.1 Read Application Memory

2.3.5.2 Write Application Memory

2.3.6 Application File Operations

To be provided in a later release.

2.3.6.1 Remote File Open

2.3.6.2 Remote File Close

2.3.6.3 Remote File Read

2.3.6.4 Remote File Write

2.3.6.5 Remote File Seek

2.4 Navigating Application Source Structure

Source structure is a concept that applies to individual processes. This means that parallel applications must expand and navigate the source structure on a process by process basis. Single program, multiple data (SPMD) applications have an advantage that the expansion need take place with only one process, but a tool may still wish to treat each process differently and thus needs the ability to operate on a process by process basis.

2.4.1 Obtaining the Program Source Object

When an end-user-tool connects with an application or process it requests the structure of the application as part of the connection process. Since applications may be very large, the initial structure requested is a very coarse view. Essentially it is the list of modules or source files in the application. This is done to avoid excessive delay and memory use when dealing with large applications. The program source object may be obtained through the `SourceObj::get_program_object` member function. This is shown in the next example.

```
Process p;
SourceObj pgm = p.get_program_object();
```

One may examine the program object to determine the number of modules it contains with the `SourceObj::child_count` member function. Source objects representing each of the modules may also be obtained with the `SourceObj::child` member function. The name of the module may be obtained with the `SourceObj::module_name` member function. For example,

```
int count = pgm.child_count();
for (int i=0; i < count; i++) {
     SourceObj mod = pgm.child(i);
     cout << mod.module_name() << endl;
}
```

prints the names of all the source files from the program.

2.4.2 Expanding the Source Structure

In order to view the structure of a given source file one must expand the source file using the `SourceObj::bexpand` or `SourceObj::expand` functions. These functions expand the source file into functions and data objects, and functions within a source file are expanded into its finer control structure. The degree to which functions can be expanded is heavily dependent upon which compiler and compiler options are used to compile the application program. The source structure is gathered from the executable and high degrees of compiler optimization severely reduce the amount of information available within the executable.

An individual process is required to expand the source structure. The process must be connected.

The simplest approach to expand the structure of a source file is to use the `SourceObj::bexpand` member function, which is a blocking service request. The service being requested is that DPCL expand the source structure for a specified file. This function does not return control to the caller until the requested service has either succeeded or failed.

```
mod.bexpand(p);
```

A second method of expanding the source structure uses a non-blocking service request. In this case the request is sent but control is returned immediately to the caller. The function does not wait until the request succeeds or fails. Instead, a callback function is provided that will process the acknowledgement of success or failure when it arrives.

```
int tag = ...;
void cb(GCBSysType s,GCBTagType t,GCBObjType o,GCBMsgType m);
mod.expand(p, cb, (GCBTagType)tag);
```

Each of these examples requests that the end-user-tool expand the source structure using the process indicated in the `Process` object `p`. The service request may fail if the process is not connected.

The following example starts with a program object, gets the first child (which is a module), expands it, then prints the names of the functions contained within the module.

```
Process p;
SourceObj pgm = p.get_program_object();
SourceObj mod = pgm.child( 0 );
assert ( mod.src_type() == SOT_module );
mod.bexpand( p );
int count = mod.child_count();
for (int i=0; i < count; i++) {
     SourceObj func = mod.child(i);
     cout << func.get_demangled_name() << endl;
}
```

2.4.3 Selecting and Identifying Instrumentation Points

Instrumentation points are locations in the program instruction stream where instrumentation code (probes) may be placed. Probes are executed any time the program executes that part of the code, for as long as the probes are in place. Instrumentation points represent locations in the program that, in some sense, are reasonably “safe” to insert new code. Examples of such locations are function entry, function exit, and call sites to other functions.

Instrumentation points are obtained from source objects at the function level or lower using the `point` or `all_point` functions. Both functions accept an integer index value as an input value and return an instrumentation point as the result. The index must be in the range of 0 to `point_count()`-1 or 0 to `all_point_count()`-1, respectively. The difference between the two functions is that `point` gives access to instrumentation points that are only tied to the given source object. In contrast `all_point` gives access to all instrumentation points associated with the given source object and all of its lower levels in the source object hierarchy.

```
SourceObj func = ...;
assert ( 0 <= i && i < point_count() );
InstPoint ipt = func.point(i);
```

One may query instrumentation points for different attributes, such as the instrumentation point type, location type, source object container, and approximate line number in source. The instrumentation point type is a value that reflects whether the instrumentation point represents a function entry point, a function exit point, a function call site (where another function is called), *etc*. The available values are contained within the `InstPtType` enumeration data type. They may be obtained through the `get_type` member function.

The instrumentation point location type reflects whether the instrumentation will come before the point in question, after the point in question, or replace it altogether. An instrumentation point represents a single instruction or a small collection of instructions in the instruction stream. Instrumentation at a point includes a branch instruction to the instrumentation site so the amount of work that can be done in the instrumentation is not limited to the number of instructions that are replaced within the instruction stream. When instrumentation is placed before or after a point, the replaced instructions are executed as part of the instrumentation code. The location type is represented by the `InstPtLocation` enumeration data type. It may be obtained using the function `get_location`.

The source object container is simply the source object that originally generated the instrumentation point. When the instrumentation point is retrieved using the `point` function, the result of the `get_container` function will always match the invoking object. When the instrumentation point is retrieved using the `all_point` function, the result of `get_container` may not. For example, consider the following code fragment.

```
SourceObj func = ...;

InstPoint pt = func.point(i);

SourceObj sop = pt.get_container();

InstPoint apt = func.all_point(j);

SourceObj soap = apt.get_container();
```

In this code fragment `sop`, the container source object of `pt`, the instrumentation point, will always match `func`, the source object originally used to retrieve the instrumentation point. This is so because the function `point` was used, and `point` may only retrieve instrumentation points that are immediately contained within a source object. In contrast, `soap` and `func` may or may not match depending on whether the instrumentation point obtained was contained within the source object `func` or a child object. If the instrumentation point is contained within `func` they would match. If it is contained within a child of `func` they would not match.

The function `get_line` queries the instrumentation point for its approximate location in source code. Code transformations that take place in the optimization phases of compilers rearrange the program in many complicated ways. Loop fusion, fission, and splitting, common sub-expression elimination, invariant code motion, and many other transformations create a complicated relationship between the source code the programmer wrote and the instruction stream that is actually executed by the machine. For this reason it is often difficult to determine precisely what line or lines of source code were responsible for creating a particular instruction in the program. Even so, this is not to say that a good approximation cannot be obtained. `get_line` returns the best approximation available.

2.4.4 Selecting Application Data

To be provided in a later release.

2.5 Instrumentation in an Application's Instruction Stream

Instrumentation of an application comes in three forms. One may create instrumentation that is time activated, that is, it is activated by the expiration of an interval timer. One may execute an Inferior Remote Procedure Call (IRPC). IRPCs are pieces of code that are executed exactly once then removed. The process is stopped wherever it happens to be executing. The IRPC is then installed inside the application, executed, and removed. Afterwards the process resumes execution where it was stopped. Periodically activated instrumentation is described in Section 2.9, "Periodically Activated Instrumentation (Phases)" on page 33. IRPCs are described in Section 2.10, "Single Execution Instrumentation (IRPCs)" on page 33.

The third kind of instrumentation is that which is placed at a particular location within the application process and executed whenever that part of the application is executed. This form of instrumentation is the topic of this section. It requires the creation of simple instruction sequences, called *probe expressions*, that serve as the instrumentation code. Probe expressions may perform conditional control flow, integer and pointer arithmetic, bit-wise operations, and call functions. When complicated instrumentation is needed, such as iteration, recursion, or manipulating complex data structures, one may call a function written in a standard language such as C to perform the complex operations.

2.5.1 Creating a Probe Expression

One of the most important notions to understand is that probe expressions within the end-user-tool are data structures. They are data structures that represent executable segments of code when they are installed and activated within an application. To make it easier to create the desired data structures the common operators have been overloaded in such a way that expressions involving probe expressions and operators almost always create new data structures, rather than executing the expression locally on the client. For example,

```
ProbeExp pe3 = pe1 + pe2;
```

creates a data structure that represents the addition of the subexpressions `pe1` and `pe2`. The addition is *not* executed in the end-user-tool, where the above statement may be found. Instead the above expression is formed into a data structure that may later be installed and activated within a process. Once the probe is activated, if the application executes the instruction at the instrumentation point, the probe will be executed and the addition will be performed at that time.

Probe expressions are like statements and expressions in a procedural language like C. There are no input parameters nor return values, because they are not functions. They are segments of code that operate within their own context so they need not interfere with the hosting application, but they also share some portions of the application's context so they can gather needed information and influence the application's behavior when desired.

The vast majority of the operators in C++ have been defined for probe expressions to create data structures rather than be executed directly within the end-user-tool. This includes arithmetic operators (+, -, *, /, %), bit-wise operators (<<, >>, ~, ^, &, |), relational operators (<, >, ==, !=, <=, >=), logical operators (&&, ||, !), assignment operators (+=, -=, *=, /=, %=, <<=, >>=, ^=, &=, |=), increment and decrement operators (++, --), and dereference operators (*, []). There are two notable exceptions. Whenever any of the listed operators are used with probe expressions, they create a new probe expression data structure that represents the same operation as if it were executed within a C program using integer or pointer operands.

There are two important omissions to the above list of operators. They are simple assignment (=) and the unary address operator (&). The assignment operator cannot be overloaded without causing simple expression manipulation to become unwieldy. Secondly, the unary address operator cannot be overloaded because it is used in passing arguments to functions that use call-by-reference. So these two operators retain their original semantics. “`a = b`” performs an assignment of probe expression `b` into probe expression `a` within the end-user-tool. “`&a`” takes the address of the probe expression object `a` within the end-user-tool. Neither operator creates a new probe expression.

Simple assignment and the unary address operators have other functions that create appropriate probe expressions. For assignment the member function is called `assign`. For the unary address operator the member function is called `address`.

```
ProbeExp pea = pe1.assign(pe2);
```

This example assigns the value computed by the probe expression `pe2` into the storage location indicated by `pe1`. The expression `pea` then represents that assignment. The next example shows how one might build an expression that represents taking the address of an object.

```
ProbeExp peb = pe.address();
```

There are additional functions that create probe expressions although they do not have corresponding operators associated with them. Examples are `call`, `ifelse` and `sequence`. `call` allows one to call a function from within a probe expression. `ifelse` allows one to perform conditional control flow within the probe expression. `sequence` allows one to chain multiple expressions together, much like a semicolon or comma operator does in C.

As mentioned before, the equality operator (==) is defined to create a new probe expression. That means some other function must be used to determine whether two probe expressions are the same. That function is called `is_same_as`.

Probe expression operands may be constants, other probe expressions, or data objects. Data objects may be application data variables, temporary probe stack objects, or persistent probe data objects. Data objects are described in more detail in Section 2.6, “Probe Data” on page 30.

2.5.2 Installing a Probe Expression

Probe expressions when they reside within the end-user-tool are passive data objects. In order to execute them as code they must be first installed within an application process, then the installed probes must be activated. Activated probes have the ability to send data back to the end-user-tool, so the end-user-tool must be prepared for that.

Probe installation requires four pieces of data. It requires the probe expression to be installed, the location where the expression is to be installed, a callback function to handle data sent by the probe expression, and a tag field for the callback function. The installation procedure returns a special probe handle for each probe expression that is installed. Probe handles may be used to identify the probe, which is needed to activate, deactivate, and remove probes from a process.

The installation interface also allows multiple probe expressions to be installed within a process at a time. If one probe expression fails to install, the interface does not install any probes for that call. This allows a tool to create a series of related probes that depend upon each other. For example, a pair of probes might turn a timer on and off. Either probe would be useless without the other, and in fact if for some region of code only one of the probes were installed it might produce inaccurate or inconsistent data. The general form is

```
Process p = ...;
p.binstall_probe(count, pe, ip, cb, tag, ph);
```

where `pe` is an array of probe expressions, `ip` is an array of instrumentation points, `cb` is an array of pointers to callback functions that will process any data sent from probes, `tag` is an array of tags that will be used when the callback is activated, `ph` is an array of probe handles that serve as identifiers for the probes, and `count` is the number of elements in each array.

A second method of installing probes uses a non-blocking service request. In this case the request is sent but control is returned immediately to the caller. The function does not wait until the request succeeds or fails. Instead, a callback function is provided that will process the acknowledgement of success or failure when it arrives.

```
Process p = ...;
p.install_probe(count, pe, ip, cb, tag, ackcb, acktag, ph);
```

`ackcb` and `acktag` are the acknowledgement callback function pointer and tag, respectively. While most of the other fields are arrays, `ackcb` and `acktag` are single values.

2.5.3 Activating a Probe Expression

After a set of probes have been installed in an application they remain passive objects until they are activated. Activation links the probes into the application code so that when the application executes the instrumentation point, the instrumentation is automatically executed as well. The probe expressions must have been previously installed, and therefore may be identified by their probe handles.

Probe activation has both blocking and non-blocking interfaces. The blocking interface is the simpler of the two.

```
p.bactivate_probe(count, ph);
```

The non-blocking interface is very similar.

```
p.activate_probe(count, ph, ackcb, acktag);
```

`ackcb` and `acktag` are the acknowledgement callback function pointer and tag, respectively.

2.5.4 Deactivating a Probe Expression

Once a probe has been installed and activated in an application process it is possible to suspend the probe without removing it completely. This saves time and effort when the probe is to be temporarily suspended and later restarted. Probe deactivation has both blocking and non-blocking interfaces. The blocking interface is shown first.

```
p.bdeactivate_probe(count, ph);
```

The non-blocking interface is very similar.

```
p.deactivate_probe(count, ph, ackcb, acktag);
```

`ackcb` and `acktag` are the acknowledgement callback function pointer and tag, respectively.

Both functions accept an array of probe handles as an input argument.

2.5.5 Removing a Probe Expression

Two major benefits of dynamic instrumentation are that instrumentation may be added when it is needed, and that it may be removed when it is no longer needed. Instrumentation that has been installed may be removed whether it is active or has been deactivated. Once it is removed it must be re-installed in order to use it again. Probe removal has both blocking and non-blocking interfaces. The blocking interface is shown first.

```
p.bremove_probe(count, ph);
```

The non-blocking interface is very similar.

```
p.remove_probe(count, ph, ackcb, acktag);
```

`ackcb` and `acktag` are the acknowledgement callback function pointer and tag, respectively.

2.6 Probe Data

Probes, like most programming v
persists from one invocation to the next. In high level languages they may be described as automatic or stack space, and static or global space. DPCL offers similar concepts. Specifically, DPCL allows one to dynamically allocate persistent data on an application or on a process-by-process basis. Allocated data may have initial values specified at the time of allocation. Once data is allocated it may be used within probe expressions, passed to functions, and used in various ways within the instrumentation. This type of allocation is dynamically managed by the end-user-tool, but it behaves like global data to the probe expression that uses it. Temporary scratch data is also available to probe expressions. DPCL maintains its own stack space where it stores temporary data and the actual parameters in function calls.

2.6.1 Persistent Data

Probe persistent data is allocated and deallocated explicitly from the end-user-tool. One may use either a `Process` or an `Application` object as the invoking object for the operation. Persistent data allocation has both blocking and non-blocking interfaces. The blocking interface is shown first.

```
ProbeExp pd1 = p.bmalloc(Ais_int32, (void*)&it, st);

ProbeExp pd2 = p.bmalloc(Ais_int32, (void*)&it, ps, st);
```

The non-blocking interface is very similar.

```
ProbeExp pd3 = p.malloc(Ais_int32, (void*)&it, cb, tg, st);

ProbeExp pd4 = p.malloc(Ais_int32, (void*)&it, ps, cb, tg, st);
```

`cb` and `tg` are the acknowledgement callback function pointer and tag, respectively.

In these examples `Ais_int32` represents the data type of the object to be allocated. In this case it is the 32-bit integer concrete data type. Other types and sizes may be allocated as well. `it` is the initial value of the object. Immediately after allocating the object the object will be set to the value indicated. If a null pointer is passed in the initial value of the object is zero. `st` is a special status value that indicates whether the request was successful (in the blocking case), or at least successfully requested (in the non-blocking case).

Data deallocation has similar blocking and non-blocking interfaces.

```
p.bfree(pe);

p.free(pe, callback, tag);
```

Daemon processes track the persistent data used by every probe expression. When data is deallocated, all probes that depend on the existence of that data are immediately removed.

2.6.2 Temporary Data

Probe temporary data is allocated automatically each time the probe expression is executed, and deallocated when the probe expression completes. Temporary data does not require an explicit service request through a `Process` or `Application` object, as is required for persistent data. Instead, new stack objects are created as probe expressions within the end-user-tool, and may be included within other probe expressions as part of the local data structure. When the probe expression is installed within the application process, the various references to stack objects are resolved to determine how much stack space is needed and where the stack data is to be allocated. The interface is as follows:

```
ProbeExp pe = Ais_int32.stack((void*)&init_val);
```

This example generates a reference to a new stack variable with a data type of 32-bit signed integer. Its initial value is stored in `init_val`. If a null pointer is passed as an argument, an initial value of zero is used. When any reference to this stack variable is included within a probe expression, the daemon will assign a location on the stack at the time the expression is installed, and space will be allocated at the time the expression is executed.

2.6.3 Probe Data Types

To be provided in a later release.

2.6.3.1 Creating Data from Built-in Probe Data Types

2.7 Passing Messages from Probes to Tools

Once the probe expression is installed and executed, to be useful it needs to be able to communicate data back to the client. DPCL provides a “send” function for this purpose. The send function needs to know what data to send, how large the data is, and to whom it is to be sent. The data to be sent takes the form of a character pointer, the size is an integer value that reflects the number of bytes to be sent, and the addressee is a special probe message handle that identifies the client and callback that are to receive the message.

2.7.1 Probe Message Send Function

In order to send a message from a probe expression to a client callback one must include a call to the function `Ais_send` as part of the expression. This may be done as follows.

```
ProbeExp args[3];
args[0] = Ais_msg_handle;
args[1] = msg;
args[2] = int32(msg_size);
ProbeExp pe = Ais_send.call(3, args);
```

This example sends a message whose reference is stored in the probe expression `msg`, and whose size is stored in the end-user-tool variable `msg_size`. The expression `pe` must be included as part of the probe expression when it is installed into the application or process.

2.7.2 Probe Message Handles

Probe message handles are data structures that contain the “address” of the message recipient. In DPCL it is possible to have many end-user-tools connected to many daemons. The significant aspect of this is that one daemon manages connections to potentially many clients, and each client may have multiple callbacks that receive data. Each time a probe expression is installed within a process a new probe handle is created with all the information necessary to route messages from the probe to the correct client and callback function.

In order to include references to message handles in probe expressions there is a symbol of type `ProbeExp`, `Ais_msg_handle`, that represents a reference to the message handle for that particular probe expression. Although the reference is the same for all probe expressions within the end-user-tool, when the daemon installs the probe expression it recognizes and changes the reference to the correct value for the new probe expression.

2.8 Probe Modules

To be provided in a later release.

2.8.1 Loading and Removing Probe Modules

2.8.2 Probe Module Functions

2.8.3 Probe Module Data

2.8.4 Probe Module Data Types

2.8.4.1 Creating Data from User-Defined Probe Data Types

2.8.5 Selecting Functions, Data and Data Types in a Module

2.9 Periodically Activated Instrumentation (Phases)

To be provided in a later release.

2.9.1 Phase Functions

2.9.2 Phase Data

2.9.3 Application Signal Handlers

2.10 Single Execution Instrumentation (IRPCs)

2.11 Security

2.11.1 Unsecure Authentication

2.11.2 DCE Authentication

To be provided in a later release.

Glossary

a.out.

Abstract syntax tree.

Acknowledgement.

Activating a probe.

Application.

Application process.

Argument. See *function argument.*

Asynchronous.

AST. See *abstract syntax tree*.

Authentication.

Begin-phase function. See *phase begin function*.

Blocking functions. See *blocking service requests.*

Blocking service requests.

Callback.

Callback function. See *callback*.

Class member function. See *member function.*

Class method. See *member function.*

Client. The AIX process which executes the end-user-tool built upon the API.

Client machine. The machine that executes the client.

Client process. Same as *client*.

Conditional control flow.

Control flow.

Daemon.

Data polymorphism.

Deacitivating a probe.

Distributed name server.

DNS. See distributed name server.

Encryption.

End user tool.

Executable image.

Function argument.

Function parameter.

Inferior remote procedure call (IRPC).

Instruction stream.

Instrumentation.

Instrumentation point.

Intrusiveness.

IRPC. See *inferior remote procedure call.*

Light-weight inferior remote procedure call.

Member function.

Message handle.

Negative acknowledgement.

Non-blocking functions.

Non-blocking service requests.

One-shot instrumentation. See *inferior remote procedure call.*

Parallel application.

Persistent data.

Phase.

Phase begin function.

Phase end function.

Phase iteration function.

Polymorphism.

Positive acknowledgement.

Probe.

Probe activation.

Probe data.

Probe data type.

Probe deactivation.

Probe expression.

Probe function.

Probe module.

Probe type.

Process.

Process ID.

Process identifier.

Program source object.

Pseudo-synchronous.

Root source object. See *program source object*.

Serial application.

Server. One of the daemon processes.

Server machine. Any machine that executes daemon processes.

Server process. Same as *server.*

Shared memory.

Signal.

Signal handler.

Single program, multiple data.

Snippet.

Source object.

SPMD. See *single program, multiple data.*

Stack data.

Super daemon.

Task ID.

Task identifier.

Trampoline.