DISTRIBUTED UPCALLS: A MECHANISM FOR LAYERING ASYNCHRONOUS ABSTRACTIONS

by

David L. Cohrs
Barton P. Miller
Lisa A. Call

Computer Sciences Technical Report #729

November 1987

Distributed Upcalls: A Mechanism for Layering Asynchronous Abstractions

David L. Cohrs
Barton P. Miller
Lisa A. Call

Computer Sciences Department
University of Wisconsin – Madison
1210 W. Dayton Street
Madison, Wisconsin 53706

ABSTRACT

It is common to use servers to provide access to facilities in a distributed system and to use remote procedure call semantics to access these servers. Procedure calls provide a synchronous interface to call downward through successive layers of abstraction, and remote procedure calls allow the layers to reside in different address spaces. But servers often need the ability to initiate asynchronous and independent actions. Examples of this asynchrony are when a network server needs to signal to an upper layer in a protocol, or when a window manager server needs to respond to user input.

Upcalls are a facility that allow a lower level of abstraction to pass information to a higher level of abstraction in a clean way. We describe a facility for distributed upcalls, which allows upcalls to cross address space boundaries. Remote procedure calls (for handling synchronous server requests) and distributed upcalls (for handling asynchronous server activities), complement each other to form a powerful structuring tool. These facilities, together with the ability to dynamically load modules into a server, allow the user to arbitrarily place abstractions in the server or in the client.

Distributed Upcalls have been built into a server structuring system called CLAM, which is currently being used to support an extensible window manager system. The CLAM system, including distributed upcalls, remote procedure call extensions to C++, dynamic loading, and basic window classes, is currently running under 4.3BSD UNIX on Microvax workstations.

1. Introduction

The server model is a common structure for providing access to facilities in a distributed system. A server provides some abstraction to its clients, and this abstraction is often implemented in several layers. The clients (application processes) will layer their own abstractions on top of the base abstractions provided by a server. Servers are typically accessed using procedure call semantics. Procedure calls provide a synchronous interface to call downward through successive layers of abstraction, and remote procedure calls[1] allow the layers to reside in different address spaces. A problem with layering using procedure calls is that it does not allow for asynchronous and independent action by the server. Actions generated at the lowest level of abstraction should be able to, in effect, call upwards through the layers of abstraction. There are natural applications for this upwards calling structure in servers supporting layered network protocols and user interface managers.

A design for structuring asynchronous upward calls, called *upcalls*, was described by Clark [2]. Upcalls allow a programmer to specify, for each layer in a system, a procedure that will be called by a lower layer in response to asynchronous events. Upcalls are implemented between layers that reside in the same address space. This paper describes a design for *distributed upcalls*, a mechanism for propagating upcalls across address space boundaries. Distributed upcalls extend the programmer's flexibility in using layers, and allow asynchronous actions to propagate upwards through any of these layers – in the server's address space and then in the client's. Distributed upcalls provide a natural complement to remote procedure calls.

We have implemented a server structuring system called CLAM [3]. CLAM allows clients to dynamically load new layers (object modules) in the server and then access these modules using remote procedure calls. Users can layer abstractions in client processes (staticly bound) or dynamically load the layer into the server. CLAM allows upcalls to cross between layers in different address spaces. The user decides where to place a particular layer based on frequency of access, speed on communication channels, speed of client and server CPUs, and requirements for sharing, debugging, and protection.

The next section describes the CLAM server and provides an example of the use of distributed upcalls. Section 3 discusses accessing the CLAM server, including the use of our RPC facility, parameter handling, and use of the C++ [4] programming language. Section 4 describes how CLAM supports

distributed upcalls. This includes a discussion of the upcall mechanism and the use of asynchronous tasks. Section 5 describes the current status of CLAM, presents basic performance data, and provides some general conclusions.

2. CLAM and an Example

The CLAM server is currently being used for development of an extensible user interface (window) manager [3]. The server itself consists of approximately 30K bytes of (VAX) code and contains no code specific to window management. CLAM allows client processes to request that new object modules be dynamically loaded into the server. These modules are then accessed by clients using remote procedure calls. Dynamically loaded procedures access other dynamically loaded procedures using normal procedure calls. The server is written in C++ and the dynamically loaded modules are C++ classes. The server contains classes to support dynamic loading, version control, task scheduling and synchronization, and distributed upcalls. All application specific code is dynamically loaded.

Following is an example of the use of distributed upcalls, based on the CLAM user interface manager. Input in user interfaces has typically been handled in one of several ways. One way to handle input is to make it completely synchronous. An input request occurs at the highest level of abstraction. This request is propagated down through the layers until it blocks at the lowest level. When the low level input event occurs, the return values from the procedures form an upward mapping of the input abstraction. This scheme make asynchronous input difficult. A second way to handle input is to have the low level input event asynchronously interrupt the user task. The client will receive a low level input event containing information such as X-Y window coordinates. This information will then have to be passed down through the layers until it reaches a level that can interpret the input, and then passed back up (by returns from procedures). This method is awkward because it forces higher levels to deal with the details of abstraction representations that they should not see.

Input is inherently asynchronous at some level. Asynchronous input events should be able to propagate up through the layers in a system, with each layer given the opportunity to map the event, queue it, discard it, or pass it up to the next layer. Each successive layer can decide whether to propagate the asynchrony (passing the event upwards) or limit the asynchrony (queuing the event). The following example

demonstrates the use of distributed upcalls in processing input.

2.1. Upcall Example

A common operation supported by window managers is to allow the user to be able to "sweep" out a new window. The user invokes this function and then uses the mouse to drag one corner of the window outline until it has the desired size and shape. Sweeping can be implemented in several places in a window system. One place for the sweeping code is directly in the window server. The server can respond quickly to input events and the dragging produces a smooth visual effect. A disadvantage of building sweeping into the server is that options such as window alignment and transparency of the sweep window are decided in the server design; little flexibility is provided to the client. A second place to put the sweeping function is in client code, as is done in the X [5] window system. This allows flexibility in choosing implementation variations, but passing every input event between the server process and a client process may be slow and can produce unpleasing visual effects.

Upcalls provide a simple solution. The code to sweep out a window is dynamically loaded into the CLAM server. Clients can decide the details of window creation and load an appropriate version of the sweeping code. Different clients could have different versions, depending on their application. Low level input routines would perform an upcall to the sweeping layer (module). This layer would process the event, redrawing the window border on each new event. Events would be processed quickly, since upcalls are basically procedure calls. When the user finishes sweeping (indicated by pressing a mouse button), the sweeping layer makes an upcall to the next layer, passing the single "window created" event. This last upcall could pass to an application layer loaded into the server or be a distributed upcall to a layer residing in a client.

3. Remote Procedure Calls

Our goal for the CLAM RPC mechanism is to minimize the distinction between local and remote procedure calls. As we minimize this distinction, we provide the programmer with more flexibility in placing abstractions in a distributed system. Furthermore, CLAM does not require the use of an external specification language for describing bindings on remote calls. We integrated the RPC stub generator with the normal compiler, freeing the programmer from writing stub specifications in addition to the procedures

themselves.

Stubs are procedures added to the client and server to *bundle* and *unbundle* parameters being passed to the remote procedure. Bundling is the task of converting a data object from its internal representation to a machine independent representation. Unbundling converts the data back into its internal representation. The compiler uses the available syntactic and typing information to automatically generate bundlers for most remote parameters. We added an extension to the C++ grammar for specifying parameter bundlers in the cases that cannot be handled automatically by the compiler.

This section first discusses the differences between the automatic and user-specified bundling of parameters to remote procedures. Next, we present the C++ modifications used by CLAM to allow user-specified parameter bundling, and describe the implementation of remote parameter bundling. Last, we describe how CLAM handles pointers that cross address space boundaries.

3.1. Automatic vs. User-defined Bundling

Two ways of generating bundlers are to make the compiler automatically generate them or to have the programmer write them. Among those systems that have compiler generated bundlers are the Lupine compiler in Grapevine[1] and SUN's rpcgen[6]. Many, but not all, data types can be automatically bundled. Primitive data types, like integers and characters that are passed by value and data structures containing only primitive types, are easy to bundle. In these cases, the bundler just passes the parameter to its counterpart in the server. Both Lupine and rpcgen allow these types of pass-by-value parameters.

Reference and pointer data types are more difficult to bundle automatically, because processes typically do not share address spaces in an RPC system. Full reference parameter semantics are difficult to support when there is no shared memory. Lupine does not allow reference parameters to be passed to remote procedures. Pointers can be supported automatically, but require complex bundling algorithms when they are part of a data structure. Consider, for example, the ways in which a node of a threaded, binary tree can be passed to a remote procedure. One way to pass the node would be to just pass the node itself, and nothing else. This bundling method will fail if the remote procedure attempts to examine the node's children as well. The other extreme is to take the transitive closure of all pointers starting at the node by recursively following its pointers. Rpcgen is an example of a system that uses this method. This

method produces correct results but can have a significant performance penalty. Taking the transitive closure can cause the whole tree to be passed remotely. When only the node itself is desired, the work to bundle the other nodes is wasted.

The alternative to automatic stub generation is to have the programmer write bundlers. This method solves the problems the automatic generators had with bundling pointers. Since the programmer may know how the data is to be used in the remote procedure, they can write the stubs to pass only as much data as necessary. While reference parameter semantics are still difficult to support, the programmer can modify the program and the stubs so that reference parameters are not required. This method has its drawbacks. It is tedious, requiring the programmer to write additional code and to deal with the underlying IPC support. Simple data types can easily be bundled automatically, so requiring the programmer to write the bundlers is unnecessary. Also, this method introduces the possibility of additional programmer error while writing the bundlers.

In the CLAM RPC facility, we chose the middle ground. Usually, the compiler can generate appropriate stubs automatically. It can handle the primitive data types and data structures without pointers. When bundling pointers, the CLAM facility allows the programmer to specify their own bundlers. Because the C++ type system is rich, the compiler has enough information to generate the stubs directly (similar to Lupine).

3.2. Grammar Modifications

A stub generator can generate procedure stubs from the source code directly, or it can use a special stub specification language. We integrated stub and bundler generation with the base compiler, like Lupine. Lupine takes a Mesa interface module, a standard part of the Mesa language, and generates the client and server bundlers directly from this specification. No modifications were made to Mesa to support RPC. Rpcgen, which generates stubs for C procedures, uses a separate stub language, RPCL, because C's typing is inadequate. RPCL includes special types to describe fixed and variable length arrays and C character strings. Lupine, because it uses the Mesa interface module, cannot allow all data types to be passed remotely. Rpcgen, by using a special language, allows all types. Because rpcgen uses a separate language, the programmer must write both the program itself and the stub specification.

We extended the C++ grammar to integrate programmer-specified bundlers in CLAM. The modified grammar allows a bundler specification to be made for each parameter and return value. With this extension, almost all C++ data types may be passed to remote procedures.

The extension takes two forms: an in place specification, used when declaring formal parameters and return values, and a type definition specification, used when declaring a new data type. The first method gives the programmer the freedom to specify a different bundler each time a data type is used. The second method, which is a modified version of the `typedef` statement, associates the bundler with the new type. Every time the new type is used as a parameter or a return value, the specified bundler will automatically be used. This is useful when a certain type is to be bundled in the same way every time it is used. The `typedef` specification has the additional benefit of making the body of a program look cleaner. If the type of a parameter has a bundler associated with it and a bundler is also specified in place, the in place bundler will be used.

Figure 3.1 shows examples of how bundlers are specified. Only a portion of the class definition is shown. Bundlers are specified following an at-sign (''`@`''). Two bundlers are specified in this example: `point_bundler`, to bundle a single point, and `point_array_bundler`, to bundle an array of points. The bundler, `point_bundler`, is associated with the type `PointPtr`, and is implied whenever this type is used in the code. The procedures `Drawpoint` and `Drawpoints` specify their bundlers in place. These procedures also take advantage of the type specifier, `const`, to denote that the parameter is read-only. The compiler uses this information to only generate a bundler to pass the parameter from the client to the server, because the parameter cannot change during the call. Two additional specifiers, `out` and `inout`, were added to the C++ grammar to allow the compiler to optimize the use of bundlers. `Out` tells the compiler to only generate a bundler to pass that parameter from the server to the client (a result parameter); `inout` specifies that the associated parameter must be passed in both directions. The `Drawline` and `get_cursor_pos` declarations make use of the `PointPtr` type and its associated bundler.

In most cases, we expect that bundlers will only take one parameter, the object to be bundled. The first parameter to the bundler is always implied; the programmer does not specify it. This also simplifies specifying a bundler with a `typedef` declaration, because the programmer may not know the name of

```
struct Point {
        short x, y, z;
};

extern Point* point_bundler(Point*);
extern Point* point_array_bundler(Point*, int);

typedef Point* PointPtr @ point_bundler();

class 3Dgraphics {
public:
                .
                .
        void Drawpoint(Point* thepoint);
        void Drawpoints(int number, Point* points);
        void Drawline(PointPtr startpoint, PointPtr endpoint);
        PointPtr get_cursor_pos();
                .
                .
};

void 3Dgraphics::Drawline(PointPtr startpoint, PointPtr endpoint)
{  /*  code to draw a line from startpoint to endpoint */  }

void 3Dgraphics::Drawpoint(const Point* thepoint @ point_bundler())
{  /*  code to draw a single point */  }

void 3Dgraphics::Drawpoints(int number,
        const Point* points @ point_array_bundler(number))
{  /*  code to draw number points */  }

PointPtr
3Dgraphics::get_cursor_pos()
{  /*  code to return the location of a 3D cursor */  }
```

Figure 3.1: C++ Procedure Declarations with Bundlers

the parameter to bundle, only its type. There are occasions when additional parameters are needed to bundle the data correctly. For example, when bundling an arbitrary length array, as in the `Drawpoints` procedure, the bundler needs to be passed the array length in addition to the data to be bundled. For generality, we do not limit the number of parameters to bundlers.

Additional parameters are more difficult when a bundler is specified in a `typedef` declaration. The main problem is to determine in which context to check the types of the additional parameters. Our system checks the parameters in the context of the procedure in which the bundler is being used, just as if the bundler were declared in place. This provides consistent results, but has the side effect that the programmer must declare the parameters to procedures using such bundlers with exactly the same names each time. to procedures with exactly the same names each time. While this is somewhat restrictive, we feel that providing the same bundler specification in both in place and type definition declarations is important

enough to allow additional parameters in the `typedef` specification.

3.3. Programmer-defined Parameter bundlers

When the programmer writes a parameter bundler, certain rules must be followed. These rules are necessary because the compiler expects all bundlers to behave the same way. The rules cover parameter specification, the communications protocol, and the use of global variables. First, for parameter specification, the initial parameter to the bundler and the bundler's return value must have the same type as the parameter to be bundled. Second, to satisfy the communications protocol, the bundler must be bidirectional; that is, it must be able to both bundle its first parameter or unbundle data from its machine independent form and return the unbundled data as the return value. This is patterned after the SUN XDR[7] philosophy for data bundling. In XDR, a filter (their term for a bundler) must be able to read data from the IPC connection and write into its data parameter, or read from the parameter and write to the IPC connection. Third, the bundler must stand alone and must not access any global variables. The bundler is dynamically loaded into the CLAM server with the class that uses it, so external references will not be satisfied. Furthermore, since the server may have multiple threads of execution, global state might change unpredictably.

As an example of a bundler definition, Figure 3.2 shows the definition of the `point_bundler` used in Figure 3.1. This bundler bundles `Point*` data types, so the first parameter and the return value are both of this type. The lowest level data bundling is performed by the bidirectional SUN XDR filters, which have been embedded in a C++ class. The variable, `RPC_XDR_stream`, denotes the IPC connection on which the bundler will send the `Point` when it is bundling, and from which it will receive a bundled `Point` when it is unbundling. Except for the special case of allocating space when unbundling data, the bundler is symmetric. The same code is used for both bundling and unbundling, and a `Point*` is returned, making the `pointerbundler` bidirectional. Notice also that `pointerbundler` uses no global variables to store the data when it unbundles a `Point`. When the bundler has no place to store the return value (when it is passed a NIL pointer), it allocates additional storage.

```
struct Point {
        short x, y, z;
};

Point* point_bundler(Point* p)
{
        // allocate some space if unbundling and the passed a NIL pointer
        if(p == 0 && RPC_XDR_stream->xget_op() == XDR_DECODE)
                p = new Point;

        // (un)bundle each member of the Point structure
        RPC_XDR_stream->xint(&p->x);
        RPC_XDR_stream->xint(&p->y);
        RPC_XDR_stream->xint(&p->z);

        return p;
}
```

Figure 3.2: A Bundler Definition

3.4. Compiler and Runtime Operation

The CLAM RPC runtime system depends on the compiler to provide it with the appropriate stubs and bundlers to make remote calls work. The compiler, given a procedure declaration, will generate a pair of stubs, one for clients and one for the server, and the code for the procedure itself. The stubs are used whenever a process makes a remote procedure call. Bundlers and stubs have no effect on local procedure calls. The client stub contains code to bundle each parameter to the procedure and code to unbundle any return value or result parameter. The server stub is complementary. The stubs contain additional code to synchronize the IPC channel and to interact with the RPC runtime code.

The CLAM RPC protocol departs slightly from the traditional RPC semantics by allowing remote calls to proceed asynchronously. This asynchrony can provide for additional parallelism. To improve performance, the CLAM RPC facility batches several asynchronous calls together into a single message, reducing the amount of interprocess communication. Our underlying communication medium guarantees reliable, in-order delivery of messages, so batched calls will arrive in the correct order. To force synchronization, the client program can either call a procedure that returns a value, or call a special synchronization procedure, which flushes the current batch to the server.

3.5. Pointers and Addresses – Crossing Address Spaces

An example of the way bundlers are used in out RPC system is in the bundling of pointers and addresses. If the programmer does not specify a bundler for a pointer data type, the compiler automatically provides a default bundler. The compiler provides special bundlers for two types of pointers, pointers to objects (i.e. class instances) and pointers to procedures. Object pointers are common because of our object-oriented design, and procedure pointers are common because of our emphasis on distributed upcalls. These bundlers are used automatically by the compiler, so the programmer can use object and procedure pointers without specifying bundlers. Like all other bundlers, these bundlers follow the three rules laid out above and provide the semantics the programmer expects from object and procedure pointers. The way in which these semantics are preserved is described below.

3.5.1. Pointers to Objects

Our system operates under three basic assumptions that affect object pointers. First, each process has its own address space, implying that an address is local to only one address space. Second, we assume that all objects are created dynamically, during program execution. Third, an object pointer is passed out of the server (when it is created) before a client passes it in (when the object is referenced). Because pointers are not valid across address spaces, we introduced a new abstraction, called a *handle*. A handle is a capability that can cross address space boundaries. It is created when an object is created in the server. A handle is a uniform means of referencing objects in the CLAM system.

At the abstract level, a handle is a capability. At the implementation level, it contains an object identifier and an arbitrary bit pattern for checking the validity of the handle, called a *tag*. The object identifier refers to the object in the server.

Since handles, not pointers, cross address space boundaries, a bundler is employed to map a pointer into a handle and pass the handle from the server to a client. The compiler generates code to automatically bundle object pointers that are passed out of the server to a client. The client bundler assumes that an incoming object pointer is a handle, stores the handle, and returns a pointer to the stored handle. Object pointers are passed from the server to the client as return values or as `out` parameters. For every such parameter, the compiler generates a call to an object pointer bundler.

The compiler also detects when an object pointer is being passed from the client to the server and generates the appropriate bundler calls. The client bundler assumes that the pointer it is bundling points to a handle and passes the handle to the server. The server then unbundles the handle and uses it to find its local pointer to the object. Figure 3.3 shows this operation. The object identifier in the handle is a pointer to a data structure in the server that contains a class identifier, version number, tag, and pointer to the object itself. The version number allows the server to have more than one version of a class present at a time. When a client wishes to perform a class operation on an object, it is necessary to know both the class and version. The tag in the object identifier is compared with the tag in the handle. If the two tags match, the real object's address can be returned by the bundler inside the server.

Since object pointers must be passed out of the server before they can be passed back in, it is not possible for the client to pass a pointer to an object of a class that is not loaded into the server. Remote NIL pointers are a special case, but can be treated like NIL pointers in a single address space. Bundlers recognize NIL pointers, pass them over the connection in a special way, and unbundle them so that they remain NIL pointers on the other side.

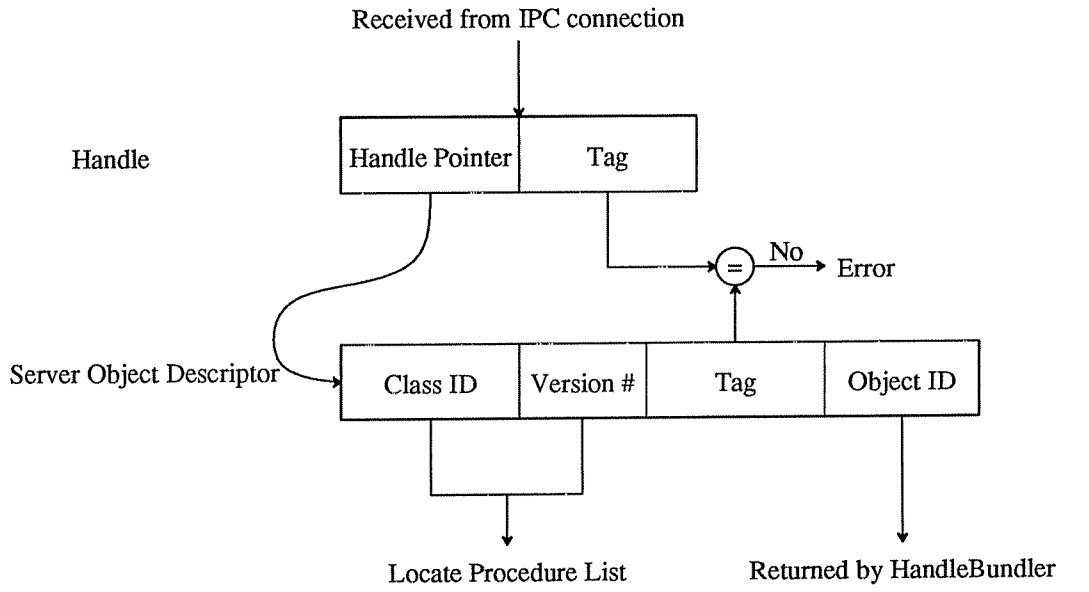

Figure 3.3: Handle Operation

3.5.2. Pointers to Procedures

The compiler automatically bundles procedure pointers. We are interested in procedure pointers that a client passes into the server. It is assumed that the procedure pointer will be used inside the server to perform a distributed upcall. It is possible to pass a procedure pointer to the client, but we have not implemented automatic means of handling these pointers.

A procedure pointer requires the compiler to generate code in addition to that required for pointers to other data types. Code to bundle and unbundle the pointer itself must be generated, as with other pointers. When the pointer is used in a distributed upcall, a pair of stubs is also needed to bundle and unbundle the parameters used during the upcall. Here, the server stub bundles parameters and unbundles return values, like the client stub in a normal procedure call. The standard C++ grammar requires that the declaration of a procedure pointer include a specification of the type of each parameter the procedure expects to be passed. The compiler uses this specification to generate the upcall stubs. The parameter specification also allows the programmer to specify bundlers for the parameters of an upcalled procedure.

The procedure pointer bundler in the server does most of the work. It stores the client's procedure pointer, the client's IPC connection identifier, and a pointer to the upcall bundler in a object in the *Remote Upcall (RUC)* class. The purpose of the RUC class is to control distributed upcalls. A procedure in the RUC class is called whenever a procedure pointer is used. This procedure, called *upcallhdlr*, is passed the object in the RUC class. Upcallhdlr bundles the pointer to the client's upcall stub and the client's procedure pointer, passing them over the IPC connection saved in the RUC object. It then calls the server upcall stub to bundle the parameters themselves and unbundle any return values.

3.5.3. Other pointers

The compiler has no special bundling rules for other types of pointers, such as pointers to integers or data structures that are not classes. In these other cases, the compiler generates code to bundle the data to which the pointer refers. This bundler does not make a transitive closure of pointers; it bundles only the object referred to by the pointer. Because this bundling method is not always appropriate, user-specified bundlers should be used for complex types.

4. Distributed Upcalls

Remote procedure calls provide the downward flow of information through the layers of abstraction. Distributed upcalls provide the flow of information upwards through these layers. We divide the description of upcalls into three parts. First, an upper layer must inform a lower layer of its intent to receive upcalls. This part consists of a registration mechanism. Second, there are the actual upcalls that pass information up to the upper layers. This part supports calls that flow upwards through the layers. Third, is a mechanism to support asynchronous activities within an address space. In CLAM, these activities are called *tasks*.

Since CLAM allows layers of abstraction to be linked either in the client (statically) or in the server (dynamically), both registration and upcalls must be able to travel between the client and server address spaces. The flow of information associated with a task must also be able to span address spaces. Distributed upcalls are conceptually the same as basic upcalls, and the goal is to make the difference between local and distributed upcalls transparent to the user. The RPC mechanisms described in the previous section are used to achieve this goal.

4.1. Upcall Mechanism

This section describes the upcall mechanism for both basic and distributed upcalls. The registration process and support for upward calls is described.

Registration involves informing a lower level object how to call a higher level object when an event occurs. The lower level object provides the upper level object with a registration procedure to call. When its registration procedure is called, a lower level object stores the information it receives in its own state. When an event occurs that requires an upcall to be made, the lower level object uses this stored information to determine which higher level object should receive the call. It is possible that zero or more higher layers maybe registered to receive the upcall. If no higher layers are interested in the event, then the lower level object decides what to do with the event. For example, it may queue the event for later use, or it may throw it away.

When both the upper and lower level objects are in the same address space, registration is a matter of passing a procedure pointer to the registration procedure in the lower level object. Registration is a simple

procedure call. When the appropriate event occurs, the lower level object will use a simple procedure call to call the registered procedure.

The mechanisms that support distributed upcalls are more complex than for local calls, since information must be passed between address spaces. The goal is to make distributed and local upcalls look the same to the applications. During registration, the upper level makes a remote procedure call to the lower level's registration procedure. The higher level object passes the address of a procedure that the lower level object will use to make the upcall. The lower level object cannot simply store the procedure address it received, as this address is only valid in the higher level object's address space. The RUC class, described in Section 3, provides the necessary address translation for the procedure addresses. The lower level object actually stores the address for a procedure in the RUC class. Through the intervention of the RUC class, the lower level object cannot distinguish between registration requests from local objects and those from remote objects. When the appropriate event occurs, the lower level object will call the RUC procedure to pass on the information to the higher level object. The RUC procedure will make the necessary remote call back to the higher level object. The lower level object views the upcall as a simple procedure call. The higher level object behaves the same in a distributed upcall as it would for a local upcall. Distributed upcalls, in most cases, are indistinguishable from the local upcalls.

4.2. An Example

This section presents an example of the use of upcalls and illustrates the behavior of the distributed upcall mechanism. This includes a description of the registration process and the flow of information during an upward call. The example is taken from the CLAM window manager.

In this example (see Figure 4.1) there are two system classes, *window* and *screen* and two additional application defined classes, *user1* and *user2*. Screen is a low level class that handles updates to the display screen. The window class provides a window abstraction layered over the screen abstraction. User1 is a class linked into a client process and accesses the window class using a remote upcall. User2 has been dynamically loaded into the server.

When the server begins execution, it creates an instance, S, of the screen class and an instance, BaseW, of the window class. While creating BaseW, the window class registers the *window::mouse* pro-

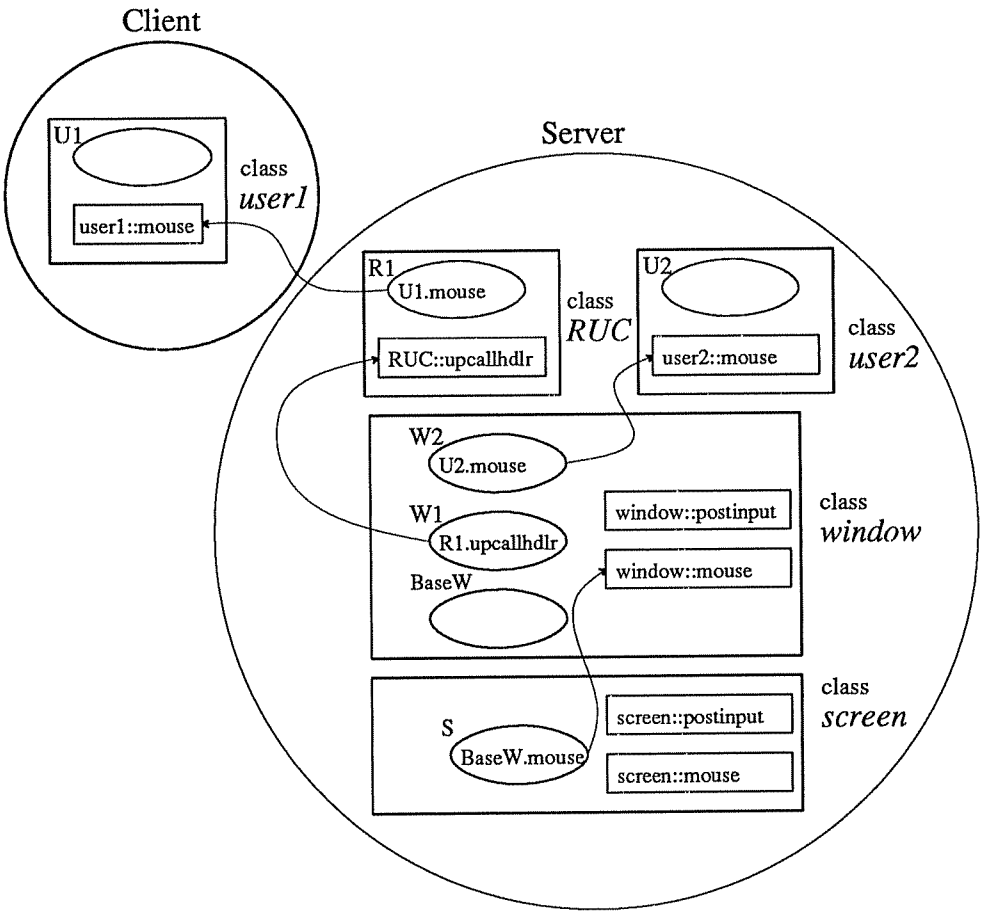

Figure 4.1: Registering Distributed Upcalls

cedure with S (by calling S.postinput) to handle all mouse button events. S.postinput saves the pointer to BaseW and window::mouse in S's state. Later, an instance, U2, of the user2 class is created. It creates an instance, W2, of the window class and registers its *user2::mouse* procedure to receive mouse events by calling *W2.postinput*. Let us assume that creating W2 notifies BaseW of the new window, so it can pass events to objects that have registered themselves with W2. An instance, U1, of the client class user1 is also created. U1 creates a window, W1, and registers its *user1::mouse* procedure to receive mouse events. Notice that the parameter bundler will automatically translate the procedure pointer into a pointer to the RUC class. For each translation, an object instance is created in the RUC class.

At this point, the state of the system is ready to handle mouse events. If a mouse button is pressed, the *screen::mouse* procedure sees the event and, using the previous registration, makes an upcall to the BaseW.mouse procedure. This procedure determines if the mouse was inside any other windows and, if so, makes upcalls to them as well. If the mouse was in the region covered by W1, BaseW then attempts to make an upcall to U1.mouse. This actually involves the upcallhndlr procedure to make a remote procedure call to the client process containing U1.

4.3. Tasks

CLAM uses lightweight processes, called *tasks,* to create asynchronous activities in the client and server processes. Tasks are provided by a thread class, which supports tasks at the user level, (as opposed to implementing them at the kernel level). The thread class includes functions for the creating, terminating, deleting, blocking and unblocking of tasks. Tasks are non-preemptive, but a task can voluntarily block itself by waiting on a specific *event.* The task is unblocked when that event occurs.

Both the client and the server processes are multithreaded. Like distributed upcalls, the flow of information associated with a task must span address spaces. When a task in the server (a server task) makes a distributed upcall, the flow of information crosses address space boundaries. While the server task cannot span this boundary, the flow of information must continue in the client. A new task is started in the client (a client task) to carry out the work in the client. The server task blocks while waiting for its corresponding client task to finish. When the client task completes, it informs the server by returning from the upcall and then terminates. The server task then becomes active, and the flow of control returns to the server.

CLAM uses tasks to create a new thread for objects that handle input events. A new task is started in the server in response to input from the external devices, such as the keyboard and mouse. This task propagates the information from the input event upward through layers of abstraction by using upcalls. If the higher layers of the abstraction are in a client process, then a distributed upcall is made and a task is started in the client to continue handling the input event.

Another application of upcalls and tasks is for error reporting. The CLAM server can protect itself from user bugs by catching error signals (such as memory faults or divide by zero.) Once the server has

determined that an error exists in a dynamically loaded class, it must decide what to do with the class. If the client has registered an error handling procedure, the server can notify the client that it tried to use a faulty class. A new task, created in the server to handle the error report, will make an upcall to notify the client.

4.4. Client/Server Channels

Conceptually, there are many channels of communication between the server and clients. There could be one channel for each client's RPC requests and one channel for each upcall between a client and the server. In CLAM, we allow only one active upcall per client process, so there are at most two communication channels (UNIX connections) between each client and the server. One channel is used for RPC requests from the client and the other is used for upcalls from the server. This limitation simplifies our first implementation and may be relaxed in future designs.

Each client requires at least two tasks, which are created when the client initially connects with the server. The first task executes the code of the application. This task blocks during RPC requests, while waiting for the return value. The second task handles all upcalls. The second task is initially blocked, and is unblocked on receipt of an upcall. After handling the event, any return value is sent back to the server, and the task is blocked again.

The server can have multiple tasks active at any given time. The main task handles RPC requests from clients. A new task is started in response to input events and performs upcalls to handle the input. If the upcall is distributed, the task is blocked while the client task is active. The task is terminated (but not deleted) after the final remote procedure call related to the input event has completed. Tasks are reused, instead of being newly created on each input event, to reduce overhead.

5. Status and Performance

CLAM is a running system. The C++ remote procedure call facility, the dynamic loading facility in the server, and the distributed upcalls facility are all working. The initial use of CLAM was to build an extensible user interface manager, and the basic classes for screen and window management are running. Current work is to experiment with CLAM in building interactive user interfaces [8].

An important motivation for providing flexibility in placing layers is the cost of interactions between layers. We have taken measurements of the CLAM system to compare the costs of remote calls (calls between address spaces) to that of local calls. These results are summarized in Figure 5.1.

	Time per call (μsecs)
Staticly linked procedures	23
Dynamically loaded procedure calling another dynamically loaded procedure	24
Remote call – both processes on same machine (UNIX domain connection)	6800
Remote call – both processes on same machine (TCP/IP connection)	11900
Remote call – processes on different machines (TCP/IP connection)	12800

Figure 5.1: Procedure Call Costs

The results in Figure 5.1 show that local calls within the CLAM server are cheap. Dynamically loaded procedures can call built-in procedures or other dynamically loaded procedures at a cost similar to that of static procedure calls. Calls that cross address spaces, even on the same machine, are significantly more expensive. Dynamically loading classes into the server can have a significant performance benefit. The performance numbers in Figure 5.1 are similar to those found in other systems. For example, the Argus[9] and Mach[10] systems show local and remote calls costs of similar magnitude.

6. Conclusions

CLAM provides flexibility by allowing the programmer to specify the placement of layers between the clients and the server. The remote procedure call facility hides most of the details of crossing address spaces, and distributed upcalls provide a clean mechanism for layering input abstractions and hiding the details of upward address space crossings. RPC and distributed upcalls together form a powerful tool for structuring servers. Remote procedure calls provide the synchronous access associated with requests to a server, and distributed upcalls allow the server to initiate asynchronous operations. Both of these mechanisms allow the programmer to work within a clean, layered structure.

REFERENCES

[1] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems* 2(1) pp. 39-59 (February 1984).

[2] D. Clark, "The Structuring of Systems Using Upcalls," *Proceedings of the 10th Symposium on Operating Systems Principles*, pp. 171-180 Orcas Island, WA, (October 1985).

[3] L. A. Call, D. L. Cohrs, and B. P. Miller, "CLAM – an Open System for Graphical User Interfaces," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 277-286 Orlando, FL, (October 1987).

[4] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass. (1986).

[5] J. Gettys, R. Newman, and T. Della Fera, *Xlib – C Language X Interface*, MIT Project Athena (November 1985).

[6] Sun Microsystems, Inc., "Rpcgen - an RPC Protocol Compiler," in *Networking on the Sun Workstations*, ().

[7] Sun Microsystems, Inc., "External Data Representation Protocol Specification," in *Networking on the Sun Workstations*, ().

[8] B. P. Miller and C.-Q. Yang, "IPS: An Interactive and Automatic Performative Measurement Tool for Parallel and Distributed Programs," *Proceedings of the 7th International Conference on Distributed Computing Systems*, pp. 482-490 Berlin, (September 1987).

[9] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler, "Implementation of Argus," *Proceedings of the 11th Symposium on Operating Systems Principles*, pp. 111-122 Austin, Texas, (November 1987).

[10] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," *Proceedings of the 11th Symposium on Operating Systems Principles*, pp. 63-76 Austin, Texas, (November 1987).