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. Servers, however, 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 allows a lower level of abstraction to pass information to a higher level of abstract in a clean way. We describe a facility for distributed upcalls that allows upcalls to cross address space boundaries. The complement of remote procedure calls for handling synchronous server requests and distributed upcalls for handling asynchronous server activities provide a powerful tool for structuring servers. 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 management 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. 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 on the part of 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 routine 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 provide a natural complement to remote procedure calls.

Distributed upcalls extend the programmer's flexibility in using layers. 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. Distributed upcalls allow asynchronous actions to propagate upwards through any of these layers – in the server's address space and then in the client's.

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 the 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. CLAM's RPC facility directly affects the implementation of distributed upcalls. Section 3 discusses the interaction between a client and 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 threads. 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 new object modules to 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

Research supported in part by the National Science Foundation grant CCR-8703373, an Office of Naval Research Contract, and an AT&T Graduate Fellowship.

are C++ classes. The server contains classes to support the dynamic loading, version control, thread 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 been done typically in one of several ways. One way to do 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 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 do 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 we find 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 manager. This allows flexibility in choosing implementation variations, but passing every input event across 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 with new event. Events would be processed quickly, since upcalls are basicly 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 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++ syntax to specify 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. If you are familiar with the issues involved in bundling parameters and the basic operation of a remote procedure call mechanism, you may skip to section 3.5, which discusses CLAM's support for passing pointers and addresses, which is central to the support of remote upcalls.

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 wants to examine the node's children as well. The other extreme is to take the transitive closure starting at the node by following its pointers recursively. Rpcgen is an example of a system which chooses 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 still cannot be supported, 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. Also, simple data types can easily be bundled automatically, so requiring the programmer to do so is unnecessary. It introduces the possibility of additional programmer error while writing the bundlers.

In the CLAM RPC facility, we chose the middle ground. Usually, he 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 sufficient information to generate the stubs directly (similar to Lupine).

We wanted to have the compiler generate bundlers for all parameters, but pointer data types in C++ have several meanings and cannot be bundled correctly and efficiently in all cases. For example, in the declaration

```
char* CharPointer;
```

`CharPointer` could denote a pointer to a single character, a C style string terminated by a NULL character, or a pointer to the first character of an array of characters of some arbitrary length. Also, if the stub generator is presented with a recursive data structure (a data structure containing pointers), it has no idea how much data to pass remotely. In both cases, passing too little data will produce incorrect results, passing too much will degrade performance. To alleviate this problem, we added a facility to allow the programmer to specify user-defined bundlers for such parameters.

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 chose to integrate stub and bundler generation with the base compiler. Both alternatives are used in other stub generators. 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 is meant to work with the C language. Because C's typing is inadequate, rpcgen requires the programmer to specify data types in a special language called RPCL. It 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.

As was described above, C++ pointers can take on a several meanings, necessitating programmer-specified bundlers. To integrate programmer-specified bundlers, we extended the C++ grammar. The modified syntax 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 be used. This is useful when a certain type of parameter 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 ("@"). `Pt_bundler` bundles a single point, and `pt_array_bundler`, bundles an array of points. The bundler, `pt_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

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

extern Point* pt_bundler(Point*);
extern Point* pt_array_bundler(Point*, int);

typedef Point* PointPtr @ pt_bundler();

class 3Dgraphics {
public:
                .
                .
        void drawpoint(Point* thept);
        void drawpoints(int number, Point* pts);
        void drawline(PointPtr startpt, PointPtr endpt);
        PointPtr get_cursor_pos();
                .
                .
};

void 3Dgraphics::drawline(PointPtr startpt,
        PointPtr endpt)
{  /*  draw a line from startpt to endpt */  }

void 3Dgraphics::drawpoint(
        const Point* thept @ pt_bundler())
{  /*  draw a single point */  }

void 3Dgraphics::drawpoints(int number,
        const Point* points @ pt_array_bundler(number))
{  /*  draw number points */  }

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

Figure 3.1: C++ Procedure Declarations with Bundlers

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 down to the server, because the parameter cannot change during the call. Two additional specifiers, `out` and `inout`, were added to the C++ syntax 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 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 array of an arbitrary length with no well-known terminal value, as in the `drawpoints` procedure, the bundler needs to be passed the array length in addition to the data to be bundled. We do not limit the number of parameters to bundlers.

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, allowing the compiler to use the bundlers at any time based on this behavior. The rules cover parameter specification, the communications protocol, and the use of global variables. First, for parameter specification, the first 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. 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. The programmer must follow these three rules, if their bundlers are to function properly.

As an example of a bundler definition, Figure 3.2 shows the definition of the `pt_bundler` used in Figure 3.1. This bundler is used to bundle `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 read 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. `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. More complex bundlers follow the same rules and structure as our example in Figure 3.2.

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 RPC protocol departs slightly from the traditional RPC semantics by allowing remote calls to proceed asynchronously. This departure allows the CLAM RPC facility to achieve greater performance than a traditional system. In other RPC systems, such as Grapevine[1], remote calls are fully synchronous; the client makes a remote call and waits until that call finishes before continuing. This is necessary whenever there are return values, but, when no return values are needed, the remote call can be delayed, and put in a batch with other calls. To further improve performance, the CLAM RPC facility batches several asynchronous calls together into a single message. Batching reduces the amount of interprocess communication, and introduces asynchrony into the RPC model. 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.

```
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

3.5. Pointers and Addresses – Crossing Address Spaces

An example of the way bundlers are used in our RPC system is in the way pointers and addresses are bundled. If the programmer does not specify a bundler for a pointer data type, the compiler provides a default bundler. 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, as was described in section 3.1, user-specified bundlers are useful here to achieve the correct semantics.

The compiler automatically 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 still 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 the bundling of 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 must be passed out of the server before a client attempts to pass it in, except for nil pointers, which are handled specially. When a pointer to an object is returned to the client, it must be returned in such a way that when the client performs a class member operation on this object, the operation becomes an RPC back into the server.

Remote operations on objects are achieved by converting a pointer to an object into a *handle* when passing it to a client. A handle is a capability for an object. The handle contains an object identifier and a tag, an arbitrary bit pattern for checking the validity of the handle. The object identifier refers to the object itself in the server, and is the only information needed to make remote object references work.

Since handles, not object pointers, cross address space boundaries, the compiler generates code to automatically bundle object pointers that are passed out of the server to a client. The use of such pointers is easy to detect. They include the object pointers that are return values of procedures, and those that are `out` parameters. For every such parameter, the compiler generates a call to an object pointer bundler. The server version of this bundler will pass a handle for the object back to the client. The client bundler assumes that an incoming object pointer is a handle, stores the handle, and returns a pointer to the stored handle.

The compiler must also detect when an object pointer is being passed to the server from the client and generate 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 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 containing a class identifier, a version number and the tag, and a pointer to the object itself. The class identifier and version number are used to locate the correct version of the correct class of the object. The tag in the object identifier is compared with the tag in the handle and, if they match, the real object's address can be returned by the bundler inside the server. Because we assume 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.

3.5.2. Pointers to Procedures

The other common type of pointer that the compiler automatically bundles is a pointer to a procedure. 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. While the server might pass a procedure pointer to the client, we have not implemented any automatic means of handling these pointers.

A procedure pointer requires the compiler to generate more code than pointers to other data types. Code to bundle and unbundle the pointer itself must be generated, just like other pointers. In addition, because we expect the pointer to be used in a distributed upcall, a pair of stubs must be generated to bundle and unbundle the parameters when the upcall is made. Here, the server stub bundles parameters and unbundles return values, like the client stub in a normal procedure call. The standard C++ syntax 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 compiler detects when a procedure pointer is an incoming parameter to a procedure in the same way as it detects incoming object pointers. The compiler first generates stubs for the client and server to bundle the parameters when the upcall occurs. It then generates calls to bundle the procedure pointer. The client bundler bundles the procedure pointer and a pointer to a stub that unbundles

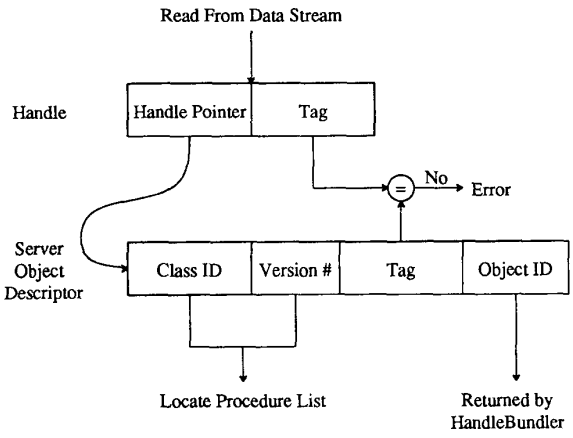

Figure 3.3: Handle Operation

upcalls of this type. The server bundler does most of the work, because the procedure pointer appears to be an arbitrary bit pattern in its address space. It stores the client's procedure pointer, a pointer to the server's upcall bundler, and the client's IPC connection identifier in a object of a *Remote Upcall (RUC)* class. The purpose of the RUC class is to control distributed upcalls. Finally, the compiler generates code to call a procedure in the RUC class whenever this procedure pointer is used, and returns the pointer to the start of this code, which looks like a normal procedure pointer. When the procedure pointer is used, this RUC procedure is executed.

This procedure, called the *upcallhandler* is passed the object that the procedure pointer bundler created when the pointer was sent down to the server. It 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. The compiler-provided bundlers and the RUC class are the basis upon which distributed upcalls are implemented.

4. Distributed Upcalls

Remote procedure calls provide for the downward flow 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 (statically) in the client or (dynamically) in the server, 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 there are no higher layers interested in the event, then the lower level object decides what to do with the event. For example, it may queue up the event for later use or 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, 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 for the lower level object to call. Using the RPC described in Section 3, this remote procedure call looks like a local procedure call to the user. The lower level object can not simply store the procedure address it received from the higher level object. 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 to applications.

A final issue is how procedures are typechecked when they are registered with a lower level abstraction. When a pointer to a procedure is declared as parameter to another procedure in C++ (as is the case in the registration procedure), the types of the parameters must be specified when declaring the pointer. The lower level procedure specifies exactly what kind of procedure is allowed to register itself to receive upcalls. Therefore, typing issues are resolved at compile time.

4.2. An Example

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

In this example there are two system classes, *window* and *screen*, shown in Figure 4.1, 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* procedure 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

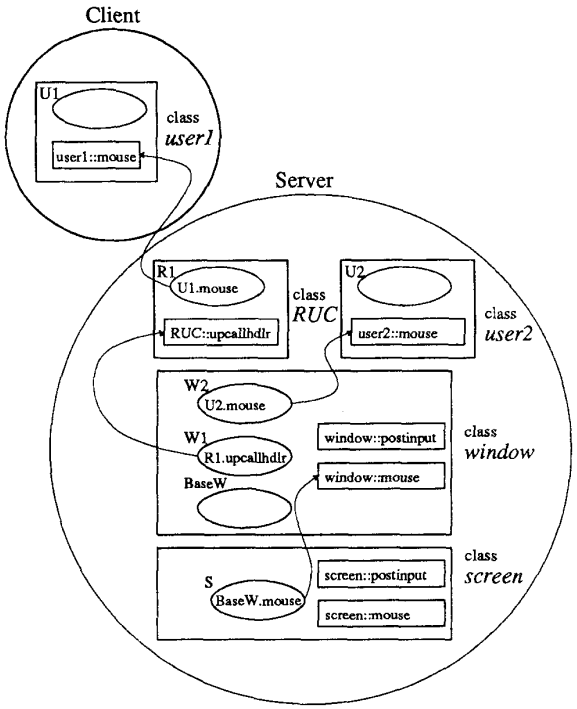

Figure 4.1: Registering Distributed Upcalls

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 RemCall procedure to make a remote procedure call to the client process containing U1.

4.3. Tasks

CLAM uses lightweight processes, called *tasks*, to create asynchrony in the server and clients. 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 creation, deletion, blocking and resumption of tasks. Tasks are created by an asynchronous call to a procedure in the thread class. Tasks are non-preemptive, but a task can voluntarily block itself by waiting on a specific *event*. The task is reactivated 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 start in the client (a client task) to carry out the work on the client. The flow of control has crossed the address space boundary into the client. While the client task is active the server task is blocked, waiting for the client task to finish. When the client task completes, it informs the server (usually by making a RPC) and then terminates. The server task 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, a task is started in the client to continue handling of the input event. The task on the server waits for the client task to complete.

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. The server can choose to notify a client that it tried to use a faulty class. A new task is created in the server that handles the error reporting. This task will make an upcall and then wait for any response the client may have.

4.4. Client/Server Channels

Conceptually there are many channels of communication between the server and clients. There would 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 upcall to be active per client process. This limitation simplifies our first implementation and may be relaxed in future designs. So there are actually at most two channels of communication 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. Without typed messages, multiplexing multiple channels of communication onto one unix stream is difficult, and requires extra information to be passed to specify which conversation is currently active. Therefore, CLAM provides separate unix streams for each communication channel.

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 then 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 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. This includes 10 main classes, representing about 10,000 lines of code. This system makes use of all of the features we described in this paper, making extensive use of remote upcalls for propagating user input and other window management events to client programs. 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.

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.

	Time per call (μsecs)
Staticly linked procedure calls	19
Dynamically loaded procedure calling another dynamically loaded procedure	21
Upcall – both procedures dynamically loaded in the server	19
Remote call – both process on same machine (UNIX domain connection)	7200
Remote *upcall* – both process on same machine (UNIX domain connection)	7200
Remote call – both process on same machine (TCP/IP connection)	11500
Remote *upcall* – both process on same machine (TCP/IP connection)	11500
Remote call – process on different machines (TCP/IP connection)	12400
Remote *upcall* – process on different machines (TCP/IP connection)	12800

Figure 5.1: Procedure Call Costs

6. Conclusions

CLAM provides flexibility by allowing the programmer to specify the placements 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 hide the details of upward address space crossings. The RPC and distributed upcalls together form a powerful for structuring servers. Remote procedure calls provide the synchronous access associated with requests to a server, and the distributed upcalls allow the server to initiate asynchronous operations. Both of these mechanism 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*, (1986).

[7] Sun Microsystems, Inc., "External Data Representation Protocol Specification," in *Networking on the Sun Workstations*, (1986).

[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] B. Liskov, D. Curtis, P. Johnson, and R. 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).