CLAM – an Open System for Graphical User Interfaces

Lisa A. Call
lisa@cs.wisc.edu

David L. Cohrs
dave@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

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

ABSTRACT

CLAM is an object-oriented system designed to support the building of extensible graphical user interfaces. CLAM provides a basic windowing environment with the ability to extend its functions using dynamically loaded C++ classes. The dynamically loaded classes allow for performance tuning (by transparently loading the class in either the client or the CLAM server) and for sharing of new functions.

In addition to the traditionally layering of output abstractions, CLAM allows the programmer to easily layer input abstractions. The input functions include providing distributed upward calls through the layers, and light weight processes to support asynchronous input.

CLAM is currently running under 4.3BSD UNIX on a MicroVax-II workstation.

1. Introduction

CLAM is an open system for graphical user interfaces. The overall goals for CLAM are to supply a flexible and extensible environment for the development and support of parallel and distributed programming applications, and to provide tools for debugging, performance monitoring and specifying such applications. CLAM provides a framework on which a consistent user interface can be made available to the various applications. This interface is based on providing windows on a bitmapped display and various devices for user input, such as a pointing device and a keyboard. CLAM's extensibility gives us the ability to provide an efficient common user interface. The current implementation is on 4.3BSD UNIX [1].

CLAM uses the framework supported by object-oriented data abstractions, such as those in Smalltalk [2] and C++ [3]. Through the use of inheritance and layered classes we have the ability to build flexible abstractions. The layered approach provides us with a user interface that can be easily extended by adding or replacing specific layers provided by the environment. An object-oriented system which provides sharing in a distributed environment, while keeping the concept of layered classes, is especially useful in designing a graphical user interface. The object-oriented environment of Emerald [4] has a different philosophy of sharing, not based on inheritance, but does provide sharing and abstraction in a distributed environment.

Flamingo[5] is another object-oriented system designed for user interface management. It also employs remote procedure calls (called remote method invocation in Flamingo). CLAM uses a much different approach for handling input events, supporting extensions to the basic system, and providing protection. The emphasis in Flamingo is towards supporting multiple display managers.

The overall structure of CLAM is based on the client/server model. Other interface managers, such as NeWS [6] and the X Window System [7] use this model. This model allows us to place low level, commonly shared routines into one package, providing a consistent interface to all clients. Flexibility in CLAM is achieved by allowing each client to specify extensions of these low level functions tailored to their specific applications. This model allows the applications to share all of the abstractions presented by the server.

We chose a less restrictive model than employed by some existing server models. X, for example, provides a fixed set of functions in the server. This can lead to a server whose design must be continually expanded to match changes in its application community. If an application requires some extension to these functions, for example, dragging a user-defined object around inside of a window with the mouse, input events must be sent from the server to the client, which must update the position of the object and send commands back to the server to update the display. This communication can result in lower performance. We would like an extensible environment that would allow clients to dynamically create objects, layers, and extensions as needed, and, for improved performance, to load these classes into the server itself. This would lead to an environment specifically tailored to the needs of each client but with the advantages of having the objects and operations existing in the server. Using this feature we can tune the performance of the system by careful separation of the application into the client and server parts.

1.1. Design Goals

We want the extensibility that is provided by a dynamic environment and to be able to specify these extensions in a natural way. NeWS provides extensibility by communicating with its clients using an extended version of Postscript[8], which allows the definition of new functions at execution time. Unfortunately, NeWS requires the programmer to manipulate the display using a language different from the one in which they write their applications. In addition, the interpreted approach used in NeWS may have lower performance than precompiled functions. To address these issues, we base our environment on C++, an object-oriented high level programming language, and provide dynamical loading of precompiled classes. The programmer has the choice of either statically linking the classes to the application program (as is usually done) or dynamically loading the class into the server.

The interface manager should provide support for both input and output. Output is a well understood problem, and all existing interface managers handle it well. Input, especially in an layered, object-oriented environment, is harder to support in a consistent manner. The approach taken in Swift [9], called *upcalls*, allows low level abstractions to call higher levels in an environment contained within a single address space. We extend the upcall method to *distributed upcalls*, which allows the low level objects to make upcalls that can cross the process boundary back to a client. We would like to provide for the maximum amount of concurrency between applications for both input and display operations. To achieve this goal, the CLAM server provides light-weight processes to the applications. NeWS also provides light-weight processes to its clients. Light-weight processes can increase the concurrency possible in updating the display. They also simplify passing asynchronous input events upwards through the levels of abstraction.

There are several other goals for the CLAM design. First, we want to provide a higher level of sharing of abstractions than is currently available. Second, we want to provide protection in the interface manager, both between clients and from intruders. Last, we want to be able to tune performance by distributing an application between the clients and the server.

The CLAM design addresses these problems. It provides a dynamic loading facility based on the high-level, object-oriented language, C++. The client/server model is employed for the interface manager, but the operations provided by the server are not fixed at compile time; rather, the server defines a set of primitive operations and is extensible both in the number and types of objects it can support. This model, in an object-oriented environment, allows us to provide sharing and protection, and flexibility in performance tuning. Through the use of distributed upcalls, CLAM provides the same level of support for input management as is commonly available for output.

1.2. Structural Overview

Figure 1 shows an overview of the CLAM system. Clients access the basic functions of CLAM by using remote procedure calls. The user writes these remote procedures (indicated by the dashed lines) using the class structure of C++, and can load them into the CLAM server. Our C++ compiler automatically generates the RPC stub routines (shown in the dotted lines).

Our discussion of the design and implementation of CLAM is as follows. First, we address the language and remote access issues. Next, we discuss CLAM's dynamic loading and binding mechanism. This includes a discussion of techniques used for version control and protection. Third, we discuss our approach to the asynchronous input problem, including support for threads of control that cross address space boundaries. Finally, we give our conclusions and the current status of the project.

2. Language and RPC Issues

The first step in designing CLAM was to pick an appropriate object-oriented language. We wanted a language that provided for layering, abstraction, inheritance and distributed programming support. Also, we wanted the programmer to be able to access all objects in a uniform way. Emerald is close to the language we wanted, but its philosophy does not include the idea of inheritance. Rather, Emerald includes *type conformity*, which allows objects to be shared only if they share the same interface; that is, the parameters to operations must be the same, but the operations may be implemented in different ways. We wanted inheritance based on implementation, where classes can inherit the implementation of an operation from other classes. No such language was available to us for a distributed environment, so we chose to add a remote procedure call (RPC) mechanism to C++. We use the RPC mechanism to access classes that have been dynamically loaded into the interface server. The RPC mechanism is restricted to work only with classes and class operations to enforce an object-oriented programming style. The choice of C++ over similar languages was based on practical issues: the C++ compiler was available to us and is a good systems programming language.

An important goal in our design was to integrate, as much as possible, the RPC mechanism into the existing programming language. In some existing languages, for example, in Cedar Mesa[10], a separate language is used by a *stub generator* to describe the parameters to remote procedures and the manner in which they should be passed to the remote procedure. We felt that this duplication of effort was unnecessary. To add RPC to C++, we extended the specification of formal parameters and return values in

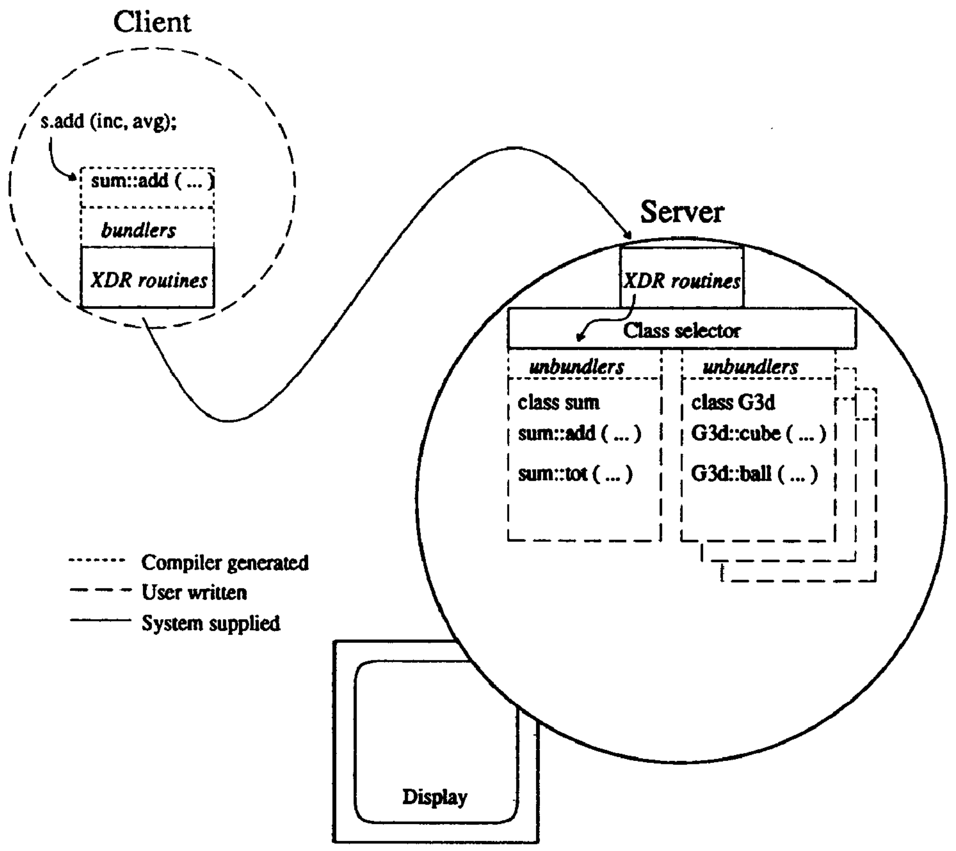

Figure 1: Overview of CLAM System

the language and require a few programming style conventions not normally required in C++. A stub language is not needed and, in most cases, modifications do not need to be made to existing class definitions to use the RPC mechanism. The modified C++ compiler will generate, given only the class specification and the class functions, object code to operate on objects on this class as well as the client and server stubs for parameter passing.

2.1. Parameter Bundlers

Each procedure that is called remotely (from another process) requires additional code to package parameters, ship them to the remote site, unpackage the parameters, and make a local call to the procedure. Output parameters are handled in a similar manner. Procedure return values are handled the same as are output parameters.

Given a formal procedure declaration, our modified C++ compiler automatically generates a *bundler* for each of the parameters. The bundler packages the parameters and uses SUN's External Data Representation[11] to pass them from the client to the server or vice versa, in a machine independent manner. In many cases, the compiler can decide on how to package a parameter. Parameters that contain no pointer references are packaged as simple data items or groups of simple items. Parameters that contain pointers are more complex. One alternative is to have the compiler (or RPC stub generator) form the transitive closure by following all contained pointers, and packaging all data in the closure. While this is conceptually the cleanest approach, this approach has several problems relating to system performance.

First, the mechanism to generate the closure and to follow the pointers to package them is complex, and this complexity can add cost to all bundling activities. Second, there are cases where generating the closure may not be the desired action. For example, to pass a single element from a linked list might cause a large part (or even all) of the list to be packaged. While this is logically correct, the performance cost could be large.

By default, our compiler does not follow pointers in data structures but passes them as raw data. These pointers are unusable to the server but usable if passed back to the client. If the programmer wishes to pass parameters in some way other than the default, they may supply a bundler for this special purpose. This escape mechanism allows the compiler to automatically bundle parameters in the simple cases, but allows C++ parameter semantics to be preserved through manual intervention.

```
class sum {
      int total;
      int num_entrys;
      char* label;
public:
      void makelabel(const char*);
      char* getlabel();
      int add(int, int*);
};

// Newlabel is a const parameter, so it is only transferred in.
// The stringbundler is used to bundle the string newlabel points
// at.  There is no return value.
void
sum::makelabel(const char* newlabel @ stringbundler())
{
      label = new char[strlen(newlabel)+1];
      (void) strcpy(label, newlabel);
}

// Getlabel returns the label that was stored by newlabel.
// Stringbundler is used once again to transfer the string.
// The procedure is written as if the caller and the sum
// class were in the same process.
char*
sum::getlabel() @ stringbundler()
{
      return label;
}

// Avg is an "out" parameter.  The bundler provided by the compiler
// will be used.
// The return value is assumed to be an out parameter.
int
sum::add(int new_value, out int* avg)
{
      total += new_value;
      *avg = total / ++num_entrys;

      return total;
}
```

Figure 2: C++ Procedure Declarations with Bundlers

We provide a library of common user specified bundlers. The most common example is for character strings. In UNIX, character strings are most often represented as a sequence of characters terminated by a null (zero) value character. The string is accessed by using a character pointer. There is a user bundler that understands this format and correctly passes strings to remote procedures. Figure 2 shows an example of a user bundler declaration. The class *sum* is used to keep a running total of values. The sum::makelabel procedure is used to give an object of class sum a name; sum::getlabel returns this label (for printing). Sum::add updates the total. Notice that the procedures themselves assume they can access the strings directly. Sum::makelabel relies on the string bundler to pass it a pointer to the string, and sum::getlabel assumes that the string bundler will bundle its return value on the server side and unbundle it on the client. In this way, C++ semantics are preserved.

We also added two new type specifiers, *out* and *inout*, to the C++ formal parameter syntax; an *in* specifier is unnecessary because C++ already has a *const* specifier with the needed semantics. These specifiers are necessary for efficiently passing C++ pointers. If we wanted to pass only a pointer to a remote procedure there would be no problem; however, sometimes we want to pass the data to which the pointer refers. The specifiers allow the parameter bundler to pass parameters in one direction rather than both, which is what C++ does normally (actually, C++ semantics pass pointers by reference, while with RPC, we can only pass parameters by value-result). Figure 2 shows an example of the use of these new specifiers. The sum::add procedure makes use of the out specifier. The avg parameter will only be passed from the server back to the client.

We consider the user bundler mechanism to be a limitation of the current implementation. One interesting alternative is to provide a *pointer fault* mechanism, whereby a reference to a pointer which is remote would cause the remote object or data structure to be copied only at that time. This is similar to the ideas being researched in distributed virtual memory [12, 13]. We plan to investigate this alternative in the future.

2.2. Programming Conventions for using RPC

We require certain programming conventions to make this work. First, all of the operations for a given object must be compiled together. This requirement simplifies the generation of the stub code that runs in the CLAM server. As a matter of style, we believe this requirement also to be a good idea. Second, only value parameters or explicit pointers are allowed to be passed to remote procedures; C++ reference parameters are not allowed. It is possible to allow remote reference parameters, but the programmer would have to supply bundlers for these parameters (as is done for explicit pointer variables).

The most important change is that all objects that belong to a dynamically loaded class must be declared dynamically. A new instance of an object is created by using the constructor for that class, so all dynamically loaded classes must contain constructor procedures. Staticly declared instances of objects for dynamically loaded class types are not permitted. This is because all object instances for a dynamically loaded class are stored in the server with the class. When a new instance of an object is created, a *handle* for that object instance is created. This handle is essentially a capability. The handle contains 4 parts: class identifier, protection key, version number, and pointer to the address (within the server) of the instance data structure. The class identifier specifies the class to which the object belongs. The protection key is a random number used to protect against simple access errors. The key can be considered part of the class identifier. The version number is described in Section 3.3.

The remote procedure call mechanism for objects is well suited to the C++ language. The additions to the syntax are straight-forward, and the new types are useful in C++ in general, not just for the RPC mechanism. The programmer can access remote classes without adopting a totally different style of programming or using a different programming language.

3. Dynamic Loading and Binding

CLAM provides dynamic loading and binding of C++ classes. The dynamic loading and binding facility supports the goals of extensibility, sharing, and performance tuning. In addition to the basic facilities, the CLAM design includes features for protection and revisions and versions.

3.1. System Structure

The CLAM server uses dynamically loaded classes to define its basic client interface. There are a few *built-in* classes, which are a static part of the server and are linked at server compile time, but all other functions are in dynamically loaded classes. A dynamically loaded class has access to all previously loaded classes and to the built-in library routines contained in the server. The *ControlClass*, which manages dynamic loading of new classes, is an example of a built-in class. The ControlClass also provides other client functions, such as querying the contents of the server. Other built-in classes define the lowest level display and input manipulation routines for the CLAM interface. All clients share these built-in classes, but each client is free to customize its specific environment by dynamically loading higher level classes. The actual loading and binding mechanism of the ControlClass is described in Section 3.3.

The built-in classes do not have to be identical between CLAM implementations. Except for the ControlClass, the client will rarely access the built-in classes directly. Most clients will build their applications on the basic CLAM library that is dynamically loaded when the server is started. This provides a uniform interface for clients while allowing for differences in the server implementation. For example, a workstation may have sophisticated display hardware, so it may be able to directly perform functions that another workstation may have to do in software. The goal is to keep actual built-in server code as small as possible, to reduce the likelihood of bugs and future changes. This is similar to the philosophy behind building a small operating system kernel.

3.2. Versions and Revisions

The user of a class may find that the class needs to be to be modified sometime after it has been loaded. The modification may be compatible with the current object data structure implementation or may be an incompatible change. A compatible change to a class is one that will still work correctly on previously created objects of that class. This type of change is called a *revision*. An incompatible change is called a *version*. These types of changes are similar to the facilities found in some database systems [14, 15]. The goal in CLAM is to make both revisions and versions transparent to the users of classes.

The version feature allows incompatible changes to be made in a class without affecting currently running clients. For example, consider the implementation of a queue or bounded-buffer. We may initially implement the queue as a linked list, but later decide to use an array implementation. Both class implementations have the same name and set of functions and can exist together. To keep the server and its internal data structures from growing too large, old versions are discarded when there are no remaining objects of that class and version.

Revisions are a simple change to the implementation of a class. All subsequent requests to that class will access the revision of the code. The old code is discarded after all requests that are currently in progress complete. To be compatible with the previous implementation of the class, revisions must be both data structure compatible and must have the same set of public functions.

3.3. Loading and Binding Mechanism

The mechanism that performs the dynamic loading is built into the CLAM server and is part of a special *ControlClass*. Through the use of this special class, a client can request that the server load a class. When the server receives a load class request, it locates the precompiled C++ class and then links and loads the code into the server, and returns to the client a handle for the newly loaded class. A client can only access classes in the server for which it has a handle. The dynamic loading of a class is a relatively

```
// Load class "sum" from a standard place if the class is not
// already loaded, otherwise use the latest version.
void* sumhandle = Control->LoadClass("sum");

// Load class "sum" from the specified file if the class is not
// already loaded, otherwise use the latest version.
void* sumhandle = Control->LoadClass("sum", "file");

// Load a new revision of the class "sum" from the specified file,
// replacing the latest version of "sum".
void* sumhandle = Control->LoadClass("sum", "file", REVISION);

// Load a new version of the class "sum" from the specified file.
// This version becomes the latest version.
void* sumhandle = Control->LoadClass("sum", "file", VERSION);
```

Figure 3: Methods for loading a new class

expensive operation compared to calling a remote procedure, but it is done infrequently relative to subsequent accesses to the class.

If clients want to share dynamically loaded classes, then the server needs only to load the code for the class once. The code for a class is loaded the first time any client requests the class. Any subsequent request from that client, or any other client, for loading that same class will be recognized by the server as unnecessary work since the class is already present. With this mechanism, sharing between clients is handled completely in the server. No extra communication between clients is necessary for them to share common classes.

An exception to the "no handle – no access" rule is needed for initially connecting a client to the server. To establish connections, the client calls the *OpenClam()* function, which is a library function linked in with each client. This function makes the initial connection to the CLAM server and requests a handle for an object of the ControlClass. The OpenClam function stores this handle in the private data area of the ControlClass stub. At this point the client is able to call functions in the ControlClass using this handle without any other special treatment.

All dynamic loading requests are made with the ControlClass handle using the *LoadClass()* function. Figure 3 shows the four forms of the *LoadClass()* function. The *LoadClass()* function allows a client to specify whether the class is a regular load request, a revision or a version. It also allows a client to specify the file from which to load the class. The server keeps track of the most recent version loaded. When a request for loading a new version is made (the fourth form in Figure 3), that version becomes the latest version. When a revision is loaded, by using the third form of *LoadClass()* from Figure 3, it replaces only the latest version of the specified class. Other versions remain unchanged. If the client specifies neither revision nor version (first and second forms in Figure 3), the server will either load the new class if it is not yet loaded, or use the latest version of the class. The server maintains the different versions through the use of version numbers in the object handles. Clients that are using older versions continue to access their version, unaffected by new versions. Revisions and versions are transparent to clients.

There are two ways a client can request to load a class. The first is an implicit load request. This is automatically built into the client code for the constructors. When the constructor of a class is called, a check is made to see if the client already has a handle for that class. If the client does not have a handle for the class, a call is made to the *LoadClass()* function requesting the latest version. After the handle is obtained for the class, then the constructor function is actually called. The second way to load a class is through an explicit call to the *LoadClass()* function. The explicit call is useful for loading revisions or versions.

Handles are used for identifying both classes and individual objects. Class handles are stored in a private data area associated with each class stub in the client. The stub only requests a new class handle if it does not already have one. This has the side effect that a client accesses only one version of a class. Even if the client performs an explicit LoadClass a second time, the stub will not obtain the new handle and will continue to use the old version of the class. Object handles are stored on the heap in the client. They appear to the client code as opaque, structureless, data objects. Object handles are automatically created by object constructors and deallocated by destructors.

3.4. Protection

Three layers of protection are needed in a server based, user interface manager. These areas are authorization to access the system, restriction of privilege access once authorization is permitted, and error detection and robustness in the server. These three layers correspond to respectively lower levels of structure in the CLAM server.

The server must be protected from unauthorized access. Our current approach to this problem is to use the host based authorization provided by TCP/IP network connections, similar to the approach used in X [7]. There are two problems with this approach. First, the authorization is done on a per host basis and does not provide a fine enough level of granularity of protection if multiuser systems are used. Any user on the authorized host can gain access to the server, where typically we wish to limit access to a select set of users. The second problem is that we rely on the network to provide host authentication. There is no guarantee that our communication media is secure and the current network protocols do not provide reliable authentication. Host authentication is a temporary mechanism that we are using in our prototype server. We are planning to base connection and authentication on a secure connection server, similar to [16].

After a process has permission to access the server, we may want to limit the classes, functions, and objects that it may access. For example, we may want to prevent the user from loading new classes or modify existing ones. We will also provide a *private* attribute for a class. A private class is one that can be accesses only by the creator of the class or by a user who receives a copy of a handle for (an object of) the class from the creator (much like capability). More work will be done in this area once CLAM starts using user authentication with the connection server.

A third area of protection is coping with execution errors. Once a processes has the capability to modify or add classes to the server, we want to protect against that code violating the integrity of the server. While we cannot (currently) protect the internals of the server from damage by bad memory references, we can prevent some logic errors from crashing the server and disrupting service to other clients. Since we can not determine, a priori, that a given set of functions are error free, we must detect errors at run time. We are able to handle such errors as arithmetic errors (e.g., overflow and divide by zero), nondestructive memory faults, and system protection violations. Whenever such an error is detected, the server regains control and unloads the faulty class.

3.5. Debugging and Performance Tuning

A class can be statically linked with the client or dynamically loaded in the server. Code in the server is much more difficult to debug than code in the client. Testing and debugging of new classes can be done by first linking the class with the client and using the debugging tools provided by the operating system for regular C++ code. Once the routines are debugged they can be removed from the client and dynamically loaded into the server. Using the protection mechanism which unloads faulty classes, we can achieve a primitive debugging facility by informing the appropriate client when the class they were using failed. We are currently investigating other types of debugging facilities that can be provided by the server.

The combination of a server model and dynamic loading permits us to tune system performance by carefully dividing the application into client and server parts so as to optimize communication and computation loads. Using the mechanisms for dynamic loading and knowledge about CLAM's environment, users can dynamically tune the performance of the system. There are two interesting cases to consider when tuning the performance. The first case deals with communications bandwidth. It is often desirable to decrease the amount of communication between the client and the server. Communication costs between the client and the server require interprocess communication overhead, while calls within the server are close to the cost of a procedure call. When the client and server are on different machines, this cost different is increased. The second case is to consider the use of processing power. It is desirable to put the heavy computation where it can best be handled. For example, if the client was on a Cray-2 computer and the server was on a small workstation, then we would put as much computation as possible on the client. If the client was on an overloaded timesharing system and the server was on a powerful workstation, we might do the opposite.

CLAM allows the user to experiment with these options without changing programming semantics. The programmer has already divided the computation into well defined classes (using the object oriented structure of C++), so performance tuning is only a matter of where to place the classes.

4. Handling Asynchronous Input

The layering of output abstractions is well understood. For example, we might display three dimensional (3-D) objects by constructing them from 2-D objects, and constructing these objects from raster operations. Each request to display a high level object propagates down through the layers of abstraction.

Layering of input abstractions is more difficult. A common method for input in a layered system is to have an input request generated at the top level and propagating it down through the lower layers. This is essentially polling and requires that the input requests match the frequency of the actual input events, so as to provide interactive response to the input events. Input events include mouse and keyboard operations.

We would like input events to propagate upwards through the layers of abstraction, as the events occur. Each event causes changes in state to occur in layers up to a specified level. These changes are asynchronous, their timing depending only on the timing of the input events. Above the specified level, input requests are made in the same way as are output requests – synchronously. Each application or input abstraction determines the boundary defined by the level up to which asynchronous input propagates.

Two parts are needed to build the input system described above. The first part is the mechanism to call upwards through the layers of abstraction. CLAM uses a distributed upcall mechanism for propagating asynchronous input to higher level abstractions. Upcalls [9] allow procedure calls to be used when a lower level class wishes to call a higher level class. The distributed upcall mechanism can propagate these requests from the server back to the client.

The second part is a facility that provides multiple threads of execution in the server. This facility is used to support asynchronous input events. Threads of execution must also be able to span process boundaries.

4.1. *Distributed Upcalls*

The concept of distributed upcalls is simple to understand. If an object of a higher level class wishes to be called asynchronously when some lower level event occurs, it *registers* its intent with an object of the lower level class. Registration is a matter of passing the address of a procedure in the higher level class to the registration procedure of an object for the lower level class. The low level object stores this address and a pointer to the higher level object in its own state. At a later time, when the appropriate event occurs, the lower level object will call the registered procedure to pass on this information to the higher level object. The information is passed upwards by a simple procedure call. The goal is to make upcalls work, without special effort by the programmer, even when the upcall requires a remote procedure call back to the client process.

Light-weight processes, called *tasks*[9], are used to provide the flow of control between objects. A new task is started whenever an asynchronous input event occurs and completes when the procedure handling the input terminates. Scheduling is non-preemptive for both simplicity and to assure that events are handled in the order in which they occur. Tasks may also block themselves. Upcalls and downcalls may cross process boundaries, so the flow of control must also be able to cross the boundary between client and server. When an upcall crosses a process boundary, the task in the server is blocked and a corresponding task is created in the client. Control is given to the client task, which eventually returns a result to the object in the server. At this point, the task in the server is unblocked and continues. The use of tasks allows asynchronous events to propagate through as many layers as have registered procedures. CLAM places no limits on the level.

There are several problems in using distributed upcalls. The first problem is what an object should do if no higher level object has registered to receive the event. The answer depends on the semantics of the object. If no receivers are registered, the event could be queued and then sent on later when an object registered to receive it, the event could be thrown away, or the object could perform some arbitrary action depending on its state.

The second problem with implementing upcalls in an object-oriented environment is type-checking the parameters to a registered procedure. The object itself may not be able to have its type checked because its class might not have existed when the lower level class was being compiled. We cannot rely on the compiler to guarantee that the types in an upcall are correct, so we need some kind of runtime support to provide this type checking. We do not have this support at this time, so we depend on the programmer to specify correct types.

The third problem with upcalls in a distributed environment is that pointers to procedures cannot be passed between processes. Our modified C++ compiler provides a special bundler for bundling procedure pointers. When a pointer to a procedure is unbundled in the server, this pointer is saved away in the state of a special distributed (remote) upcall (RUC) class object, and a pointer to a procedure within the RUC class is actually registered. When the procedure pointer is used, a procedure in the RUC class is called and it, in turn, makes a remote procedure call back to the client, receives return values, and returns them to the caller. While this is complex at the implementation level, the programmer is not required to use any special syntax to make the distributed upcall work. The compiler and runtime support automatically make distributed upcalls as easy to use as upcalls in a single process environment.

4.2. An Example

This section presents an example of the use of upcalls. Assume that there are two classes, *window* and *screen*, shown in Figure 4, 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 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 initial 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 makes an upcall to U1.mouse. This causes the RemCall procedure to make a remote procedure call to the client process containing U1.

As this example shows, the combination of distributed upcalls and normal downward procedure calls combine to provide a straight-forward flow of control between objects in an object-orient system. Down calls are not needed to get information from lower level abstractions.

5. Conclusion and Status

CLAM is based on the idea that complex systems are more easily built by providing a small collection of basic functions and a powerful means of composing these functions. The facilities in such a system are easily extended. Programmers write their extensions in the same language as they use for their applications. Programmers also have the flexibility to move these extensions between the client and the server.

A major part of the CLAM design is to allow users to abstract input as easily as output. This facility seems to provide a useful and powerful organization, but more experience is needed to test this design.

Protection in a single address space is a difficult problem. Some relief is offered by using compiler generated checks, but this is not bullet proof. This is an area for future study.

The current CLAM implementation is on a MicroVax-II workstation running 4.3BSD/NFS UNIX [1, 17]. The initial version of CLAM is now being tested. The C++ compiler has been modified to produce the necessary code for remote procedure calls and the language has been extended to allow the user to specify bundler routines. The basic display routines and dynamic loading facility are running. One of the first large applications will be an X window library interface. This will allow us to use the collection of existing

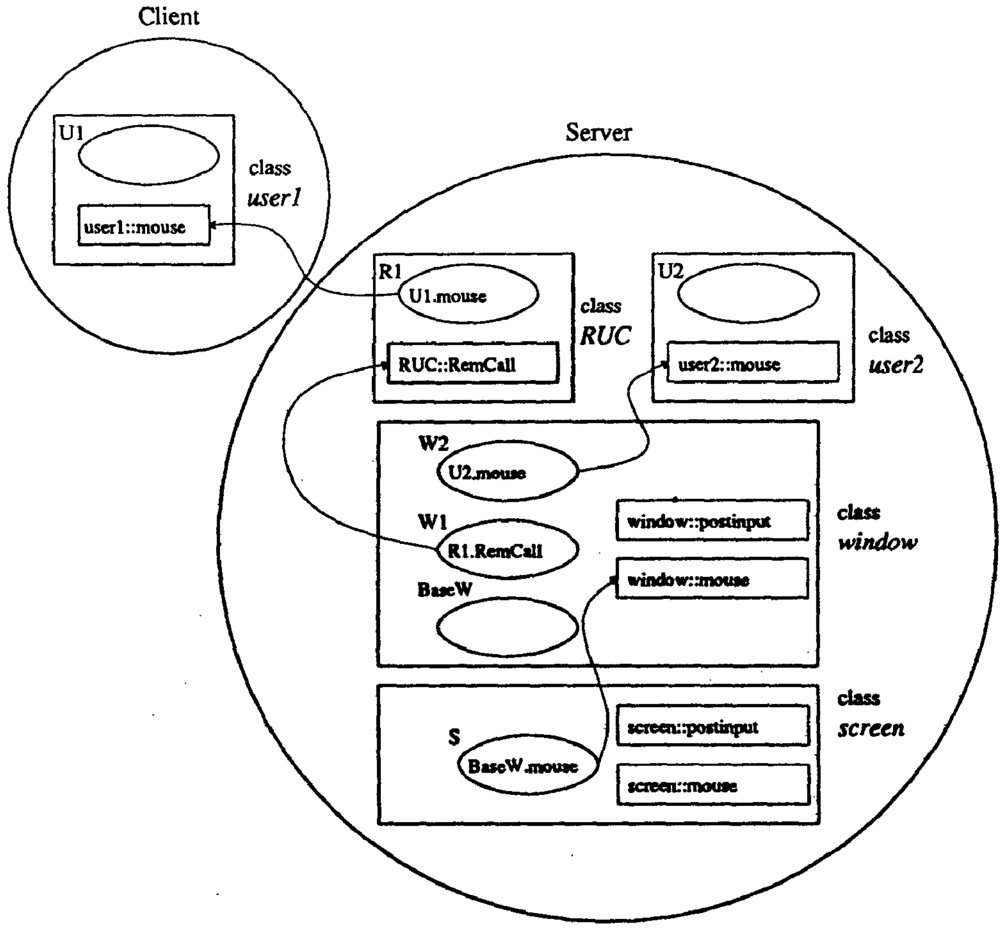

Figure 4: Registering Distributed Upcalls

X-based applications until we can build more of these facilities directly into CLAM.

REFERENCES

[1] W. N. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick, and D. Mosher, *4.2BSD System Manual*, University of California, Berkeley (July 1983).

[2] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass. (1983).

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

[4] A. Black, N. Hutchinson, E. Jul, and H. Levy, "Object Structure in the Emerald System," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, Portland, OR, (October 1986).

[5] D. B. Anderson, "Experience with Flamingo: A Distributed, Object-Oriented User Interface System," *Proc. of the Object-Oriented Programming Systems, Languages and Applications Conf.*, pp. 177-185 Portland, OR, (September 1986).

[6] Sun Microsystems, Inc., *NeWS Preliminary Technical Overview*. October 1986.

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

[8] Adobe Systems, Inc., *PostScript® Language Reference Manual*, Addison-Wesley, Reading, Mass. (1985).

[9] 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).

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

[11] Sun Microsystems, Inc., *External Data Representation Protocol Specification.*

[12] N. Carriero and D. Gelernter, "The S/Net's Linda Kernel," *ACM Trans. on Computer Systems* 4(2) pp. 110-129 (May 1986).

[13] Kai Li, "Shared Virtual Memory on Loosely Coupled Multiprocessors," Technical Report YALEU/DCS/RR-492, Ph.D. Dissertation, Yale University (September 1986).

[14] R. Katz and T. Lehman, "Storage Structures for Versions and Alternative," *IEEE Trans. on Software Engineering* **10**(2)(March 1984).

[15] A. H. Skarra and S. B. Zdonik, "The Management of Changing Types in an Object-Oriented Database," *Proc. of the Object-Oriented Programming Systems, Languages and Applications Conf.*, pp. 483-495 Portland, OR, (September 1986).

[16] D. Draheim, B. P. Miller, and S. Snyder, "A Reliable and Secure UNIX Connection Service," *Proceedings of the Sixth Symposium on Reliability in Distributed Software and Database Systems*, Williamsburg, VA, (March 1987).

[17] R. Sandberg, "The Design and Implementation of the Sun Network File System," *Usenix Association Summer Conference Proceedings*, pp. 119-130 Portland, OR, (June 1985).