For: rees.staff.none.24542
Printed on: Wed, Nov 21, 1990 23:48:27

Document: ost
Last saved on: Mon, Apr 28, 1986 14:07:34

pl2ps 4.0.5 Copyright 1988 Interleaf, Inc.

To be presented at Summer 1986 Usenix Conference, Atlanta, Georgia.

An Extensible I/O System

Jim Rees, Paul H. Levine, Nathaniel Mishkin, Paul J. Leach

apollo computer inc
330 Billerica Road
Chelmsford, MA 01824

Introduction

For years, programming environments have provided **device independent** program I/O. The programmer normally codes file I/O requests using a standard set of procedure calls, such as the Unix *open, close, read,* and *write* system calls, or language specific I/O calls. This model enables a program written primarily to perform I/O to simple files to also read from keyboards or IPC channels, and to write to display windows or IPC channels without any modification. The intent is to unburden the programmer from the necessity of either binding the program to a specific target for its I/O or enabling the program to adjust to the vagaries of different I/O targets at program run–time. That is, to make the applications program I/O independent of target type.

While this concept has been around for a long time, the systems that implemented the concept have generally had one major shortcoming. The only way to add a new type of I/O target to the system was to modify the system source. In the case of Unix operating systems, for example, it is necessary to modify and rebuild the operating system kernel and to have all of the software that implements the management of the new I/O target permanently wired into physical memory. Most schemes for adding new file types to the Unix kernel operate at the file system level, so that within a given file system, all files have the same type. Further, whenever a new type is added, various pieces of the system have to be modified to behave correctly with respect to the new type. Because of this sizable burden, programmers are discouraged from defining numerous I/O target types.

Our goal was to create a framework in which file I/O could be truly extensible — to allow users to define new types without modification to the basic system. Our work consisted of building a general framework for extensibility and then applying those techniques to stream I/O. We call the framework a **typed object management system**; and the associated file I/O facility **Extensible Streams** (ES). The combination of these two is called the DOMAIN® Open Systems Toolkit.

The system resulting from our work is novel because it:

- Supports (relatively large) typed, permanent, sharable objects in a distributed file system.
- Allows users to define new types of objects.
- Allows users to associate generic procedures (operations) with types; the procedures are dynamically loaded into the address space of processes when the procedure is invoked.

The Open Systems Toolkit allows users to extend the DOMAIN file system by inventing new file types and writing managers for these types. The current implementation allows dynamic creation of

Unix is a trademark of AT&T.
DOMAIN is a registered trademark of Apollo Computer Inc.

Copyright 1986 Apollo Computer Inc.

new types, and dynamic binding of typed objects to the **managers** which implement their behavior. Type managers are written and debugged as user programs and require no kernel modifications for installation. This system has been used successfully to write and debug new device drivers, to add new types of files, and to provide remote file system interconnects to foreign file systems.

DOMAIN Architecture

The DOMAIN system [1] is an architecture for networks of personal workstations and server computers that creates an integrated distributed computing environment. A major component of this distributed system is a distributed file system [2] which consists of four major components: the object storage system, mapped file management, concurrency control and naming service.

The DOMAIN distributed object storage system (OSS) provides location transparent typed object management across a network of loosely coupled machines. We say "object" rather than file to specifically include all of the named non-disk objects in a computing environment, such as devices (serial I/O lines, magtapes, null, etc.), IPC facilites (sockets, etc.) and processes. While a naming service manages a network-wide heirarchical name space, at the OSS level objects are named by a 64-bit **unique identifier** (UID). The UID consists of a time stamp and a unique node ID. This guarantees that the UID is unique across all DOMAIN nodes for all time.

A 64-bit **object type UID** is associated with every object. This type is used to divide the set of all objects into classes of like objects; all of the objects in a class have common properties and must be operated upon by a single set of procedures. We use a UID (rather than any other kind of type identifier) because a system facility supports the unique creation of these 64-bit numbers across all Apollo products. In the basic DOMAIN system there are several types, including ASCII text, binary, directory, and record. This strong typing allows the creator of an object to explicitly specify its intended use and interpretation, rather than depending on the conventions and cooperation of other users and programs.

The DOMAIN OSS supports a consistent set of facilites for naming, locating, creating, deleting, and providing access control and administration over all objects. Each object has an inode, which we have extended to contain (among other things) the type UID of the described object.

For disk-based objects OSS also provides storage containers (arrays of pages) for uninterpreted data. A process accesses this data by handing the kernel the object's UID and asking for it to be **mapped** into its address space. The process then uses ordinary machine instructions to directly manipulate the contents of the object — the single level store (SLS) concept of Multics [3], Pilot [4], and System/38 [5].

Layered on top of the file system is the **Streams** library, a user state library mapped into every process's address space, which provides a traditional I/O environment for programs. The Streams library implements the standard I/O interfaces and so provides equal access to both disk and non-disk resident objects. The DOMAIN Stream operations form a superset of the Unix file I/O operations, as they include record-oriented operations and more inquiry operations but are all based on a file descriptor returned to callers of *open*. Streams is an object-oriented facility in that its behavior is determined by the type of object to which its operations are applied. When a stream operation is invoked, Streams calls the manager that handles operations for the type of the object being operated on. Figure 1 diagrams the relationships among the various pieces of the DOMAIN object management system and Streams.

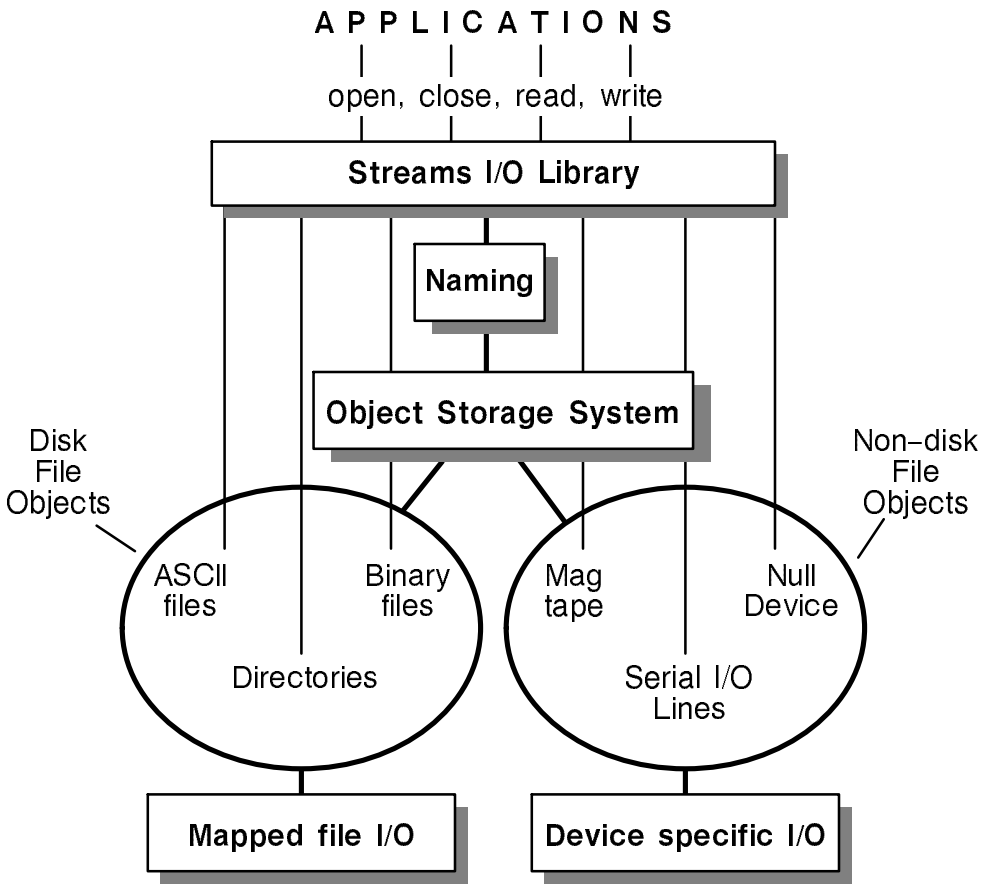

Figure 1. The relationships among the various pieces of the DOMAIN object management system and Streams.

Typed Object Management

The fundamental concept underlying the object–oriented part of our system is the notion that every object is strongly typed and that for each object type there is a set of executable routines that implement a well–specified group of **operations** on that type. This section describes the object typing strategy, defines operations and describes their partitioning into traits. It also explains the management facilites necessary to associate typed objects with the code that implements the operations defined for them.

Unix file system objects do not have an explicit type tag, but do keep a form of type information in several different places. The *mode* field in a file's inode contains some bits that distinguish among ordinary files, directories, character and block special files (devices), and depending on the version of Unix system, FIFOs, sockets, textual links, and other types of file system objects. There may also be type information coded into the major and minor device numbers to, for example, distinguish between tape drives and disk drives. In some cases, type information is encoded in the first few bits of the file data itself. For instance, there may be "magic numbers" for tagging various flavors of executable (a.out) files.

In the DOMAIN system, the type tag is a UID which is explicitly attached to the object at the time that the object is created. This provides the advantage of a single, common mechanism to

distinguish among all types. The use of a UID (rather than a small integer) allows the arbitrary creation of new types without appealing to a central authority.

The fundamental concept underlying the object–oriented part of our system is the notion of an object type as a set of legal states together with a collection of **operations** that implement the state transitions. Operations can be viewed in two ways: as a specification of how to invoke a transformation on the state of an object, or as the executable code that performs the transformation. The collection of code that implements the set of operations for an object type is known as that object type's **type manager**.

A **trait** is an ordered set of operations. It represents a kind of behavior that a client desires from an object. For example, the operations *open, close, read* and *write* could be a "stream-like" trait, and the operations *set speed* and *echo input* could be part of a "tty–like" trait. An object supports a trait if its type manager implements the operations of the trait. For every trait that a type manager supports, the manager provides an **entry point vector (EPV)**, that is an ordered list of pointers to the procedures that implement the operations in the trait.

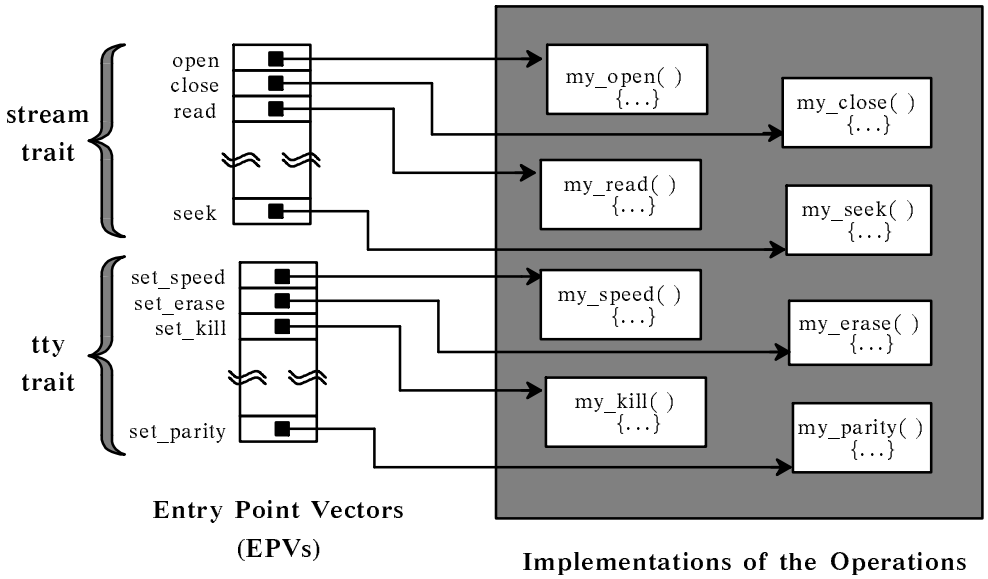

Figure 2. The type manager consists of the routines that implement one or more sets of operations (traits) and the entry point vectors (EPVs) that map the supported operations to the routines that implement them.

The implementation of the typed object management system has two main components: the **type system** and the **trait system**. We use the name **Trait/Types/Managers (TTM)** to refer to these two components plus the set of all type managers.

The type system is responsible for maintaining a data base containing mappings between type UID, type manager, and type name. New types can be created at will. For convenience, there is a name for every type, but a type UID rather than a type name is actually attached to the file system objects. This guarantees that all types are unique, even if two different implementors independently choose the same name. The type system provides procedures that can be used to create new types, associate a name with a type, and look up type UID of a given type name. It can also find the manager for a given type.

The role of the trait system is to bind <object, trait> pairs to type manager EPVs. It provides the *trait_$bind* call for this purpose. This call looks up the object's type UID and then asks the type system for the corresponding type manager. Object code libraries containing managers are not pre-linked with client object code. Rather, the trait system is responsible for dynamically loading them into the address space of clients as necessary. To perform this task, the trait manager uses the type system to locate the manager object code file. It then loads the manager into the address space of the client. The type manager is linked as an autonomous program whose main entry point is called when the manager is loaded. The code at this entry point registers all supported <trait, EPV> pairs with the trait system. Once the manager is loaded, the trait system returns the requested EPV to its client.

The type definition for an EPV corresponding to a trait that describes operations on stacks might look like:

```
typedef struct {
            void                    (*push)(uid_$t, stack_$elem_t);
            stack_$elem_t           (*pop) (uid_$t);
} stack_$epv;
```

The actual EPV for a type manager that supported the stack trait would be declared as:

```
stack_$epv my_stack_epv = {
            my_push,
            my_pop,
};
```

where my_push and my_pop are the names of real procedures that implement the *push* and *pop* operations:

```
void my_push(obj, elem)
uid_$t      obj;
stack_$elem_t elem;
{
...
}

stack_$elem_t my_pop(obj)
uid_$t      obj;
{
...
}
```

The client uses *trait_$bind* to get a pointer to an EPV from the trait system:

```
trait_$epv *trait_$bind(obj, trait, typuidp, statusp)

uid_$t      obj;                    /* IN:  object we want to operate on */
trait_$t    trait;                  /* IN:  trait we want to use */
uid_$t      *typuidp;               /* OUT: type of object */
status_$t   *statusp;               /* OUT: status */
```

Once a client has called *trait_$bind* and received an EPV, it can invoke operations on the object. For example, to call the *push* and *pop* operations in the sample trait above:

```
epv = (stack_$epv *) trait_$bind(my_obj, stack_$trait, &type_uid, &status);
(*(epv->push))(my_obj, an_elem);
an_elem = (*(epv->pop))(my_obj);
```

The DOMAIN system provides a set of programs for creating and installing new types and their managers. A user who creates a new type will also typically write a type manager for that type. The manager is written as a set of subroutines, each implementing an operation for the traits that the manager supports. The programmer can use the standard debugging tools on the type manager. The manager is installed by running a program that puts the executable code in a well-known place and registers the new manager with the type system data base. No kernel modifications are required, and

the machine does not have to be rebooted. There is no limit on the number of object types a single system may support since their managers are only loaded when needed.

Extensible Streams

Extensible Streams is a client of TTM. ES defines three basic traits: IO, IO_OC, and IO_XOC. The IO trait contains the traditional I/O operations — *get (read), put (write), seek,* etc. The IO_OC trait contains the operations *open* and *initialize*. (The IO_XOC trait is similar to IO_OC except that it supports **extended naming**, a facility that allows non-standard pathnames, described below.) ES also defines a set of **auxiliary traits** containing operations that only some type managers will choose to implement. The current set of auxiliary traits include: SIO (operations for manipulating serial I/O lines), SOCKET (operations corresponding to the 4.2bsd Unix "socket" system calls), PAD (operations for manipulating windows), and DIRECTORY (operations for reading and manipulating directories).

ES introduces a layer of abstraction on top of the basic operations. This layer — called the **I/O Switch** — supports the notion of an **open stream** and isolates the user of file system I/O from the TTM. An open stream is created by calling the I/O Switch procedure *ios_$open* which:

- Calls *trait_$bind* to get the IO and IO_OC EPVs for the object being opened.
- Calls the manager's *open* operation. This operation returns a **handle** — a virtual address of a descriptor that is meaningful only to the manager. The manager stores in the handle whatever information it needs in order to maintain the semantics of an open stream (e.g., position in stream, buffers).
- Allocates an entry in the **stream table** — a table of open streams. Each entry in this table contains the EPVs for the IO and IO_OC traits, and the handle returned by the *open* operation.
- Returns the small integer — the **file descriptor** — that identifies the table entry allocated in the previous step. This file descriptor is used by the application program on subsequent calls.

Another I/O Switch procedure, *ios_$create*, is similar to *ios_$open* except that it creates a new object and calls the manager's *initialize* operation. In addition to returning a handle, the *initialize* operation stores any information it needs to in the newly-created object.

For each operation in the IO trait, a trivial I/O Switch procedure takes a file descriptor as its first argument, converts the descriptor to a handle (by consulting the stream table), and calls the appropriate procedure from the EPV (also obtained from the stream table). The various forms of I/O (e.g., Unix I/O system calls, FORTRAN and Pascal language I/O primitives) are implemented in terms of these I/O Switch procedures.

Extended Naming

Extended naming is a facility that allows the pathname of an object being opened to be augmented with additional text to be interpreted by the Streams manager of the object to which the pathname refers. This additional text is called the **residual pathname**.

If an application calls the I/O Switch's open procedure with a pathname containing a residual, and the non-residual part of the pathname names an object whose type manager implements the IO_XOC trait (as opposed to the IO_OC trait), then the I/O Switch passes the residual to the manager as one of the arguments in the IO_XOC *open* operation. The manager is free to interpret the residual in any way it chooses.

Program–level I/O based on a simple system naming facility allows an application program to pass the name of a file system object into the *open* call, for the I/O Switch to locate the specified object, and for the manager of that type of object to then do its job. For example, the pathname */usr/fonts/classic* refers to the object whose name is *classic,* which is catalogued in the directory whose name is *fonts,* which in turn is catalogued in a directory object whose name is *usr*. The I/O Switch *resolves* the entire pathname down into the single target object, and passes a shorthand identifier for that object to the manager.

The intent of extended naming is to allow the object managers themselves to take over part of the pathname–walking responsibility so that they can manage a collection of objects that can be distinguished by the remainder of the pathname. To clarify this notion, consider the following.

The pathname */jim/test.c* would normally be interpreted as a file named *test.c* catalogued in the directory named *jim*. The name also suggests that the file is a C language source file and that all operations that would need to work on such a file (e.g., compiling, printing, editing) could be requested by specifying this name.

Now let's suppose that file *test.c* is of type *history*. The actual file system object contains the entire change history of the file, much the same way that a SCCS [7] file does. Programs that do not care about the change history can open this file and read from it. The *open* and *read* requests are passed on by the I/O Switch to the *history* type manager, and the manager can be written so that the program always reads the latest version of the file.

Extended naming takes the concept one step further by allowing the manger writer for the *history* object type to allow the specification of additional pathname text. Where the simply specified pathname results in the reading of data from the *latest* version of the file *test.c,* the manager writer might wish to allow a naming syntax of the form */jim/test.c/–1* to indicate that the application wishes to use the penultimate version of the file instead of the newest. The I/O Switch allows this additional specification to be issued at the application program layer and passed through to the manager for the target object.

The application passes the pathname (with the extended name) to the I/O Switch *open* routine. The *open* routine evaluates the pathname one pathname component at a time walking from left to right. In the current example, *jim* is a directory where the name *test.c* is located. *test.c* is discovered to be a *history* file (not a directory), and because the original pathname still has remaining text ('`–1`') that the I/O Switch cannot resolve, it passes that remainder to the *history* object manager's IO_XOC *open* routine. The *history* manager is then able to decide what text to provide to subsequent *read* requests and the intended result occurs. In this case, the application program is not affected by the apparent peculiarity of the original pathname. The I/O Switch avoids confusion by only walking the pathname through objects that support the *directory* trait and the manager is able to get whatever information it needs to do the job it was written to do.

Other examples of extended names a *history* manager might be willing to accept are:

```
/jim/test.c/03.02.85
/jim/test.c/original
/jim/test.c/yesterday
```

Another example of the application of extended naming is a gateway to a non–DOMAIN file system. For example, imagine an object whose name is `THEM` and whose type is *UNIX_gate*. A pathname of the form */gateways/THEM/usr/jan/test.c* could be passed by an application program to the I/O Switch. The Switch would see that the object named *gateways* was a directory and would look the name *THEM* up in that directory. *THEM* would be found to be a *UNIX_gate* object, and

since the Switch cannot walk the pathname through objects that are not directories, it would call the *UNIX_gate* object manager's *open* routine. That routine is passed the UID for the object whose name is *THEM* and the remaining pathname (*/usr/jan/test.c*). The *UNIX_gate* manager then has the information it needs to contact a remote file service for the data it needs to meet the demands of the requesting application program. The protocol that the manager uses to access the remote files is entirely up to the manager writer, and because the manager runs in user space, it is not restricted to kernel services but can use any service available at the user level. This scheme has been used to build a type manager that interconnects the DOMAIN file system with a generic Berkeley 4.2 Unix file system, and another that connects to a VAX/VMS file system.

Underlying Facilities

Many facilities provided in the DOMAIN environment made the implementation of TTM and Extensible Streams possible. These facilities make it possible to write OS–like functions in user space.

The underlying virtual memory system — which allows objects to be mapped into the virtual address space — is needed to give type managers low level, yet controlled, access to the raw data in objects. The virtual memory system allows more flexible access to the address space than that allowed by *sbrk(2)*. These calls take the name of an object, map the object into the address space, and return a pointer to (i.e. the virtual address of) the mapped object. The address space of a process can be characterized solely in terms of what objects are mapped where. Processes are not allowed to make memory references to parts of the address space to which no object is mapped.

The read/write storage (RWS) facility is a flexible and efficient storage allocation mechanism. It is implemented in user space in terms of the virtual memory primitives; it maps temporary objects into the address space and allocates storage from that part of the address space. It allows storage to be allocated from multiple **pools**. One pool corresponds exactly to the type of storage allocated by *malloc*. Another pool is similar, except its state is not obliterated by *exec* calls. Type managers must use storage from this pool to hold per-process state information since open streams must survive calls to *exec*.

RWS also provides a global storage pool. The global pool is a place where storage that can be viewed from all processes' address spaces can be allocated. The allocation call returns a pointer to the allocated storage, and this pointer is valid in all processes. Type managers must use storage from the global pool to maintain things like the current position (i.e. offset from beginning-of-file) of an open stream. If a process opens a stream to an object, forks, and then the child does I/O to the stream it was passed, the parent sees the position of *its* stream change too. Thus, position information must be in storage accessible to both parent and child. Because type managers run in user space, they need a user space global storage allocater for this purpose.

The dynamic program loader allows the system to load managers as they are needed. Managers for types that are not used by a given process do not take up any virtual address space in that process. The loader is implemented in user space in terms of the RWS facility (to allocate space for static data) and the mapping calls. The pure parts of executable images are simply mapped into the address space before execution, because the compilers produce position–independent code. In 4.2bsd, only the kernel can be dynamically linked to; all other subroutines must be statically bound to the program which uses them.

The eventcount [8] (EC2) facility is the basic process synchronization mechanism. Eventcounts are similar to semaphores: eventcounts are associated with significant events, and processes can advance an eventcount to notify another process that an event has occurred, or wait on a list of eventcounts until the first event happens.

A design principle for all DOMAIN interfaces is that for every potentially blocking procedure in an interface, there is an associated eventcount that can be obtained through the interface and that is advanced when the blocking procedure would have unblocked. This always allows programs to wait for multiple events (say, input on a TTY line and arrival of a network message) simultaneously. The 4.2bsd *select(2)* system call is implemented in terms of eventcounts. However, unlike *select*, eventcounts can also be used to wait on non–I/O events, such as process death.

The mutual exclusion (MUTEX) facility is a user-state library that contains calls that allow multiple processes to synchronize their access to shared data (i.e. data in objects that are mapped into multiple processes). MUTEX is implemented in terms of EC2. MUTEX defines a lock record that consists of a lock byte and an eventcount. Typically, applications embed a record of this type in a data structure over which mutual exclusion must be maintained. A MUTEX lock is set by calling *mutex_$lock,* which attempts to set the lock byte (using the hardware test-and-set instruction). If it fails to set the lock byte, it waits on the eventcount; when the wait returns, *mutex_$lock* repeats the attempt to set the lock byte. *mutex_$unlock* unlocks a MUTEX lock by clearing the lock byte and advancing the eventcount. Type managers use shared storage to maintain various kinds of information. To control access to this data, managers use the MUTEX facility.

The shared file control block (SFCB) facility allows multiple processes to coordinate their access to the same object. There is various dynamic information that processes might want to keep about an object. For example, type managers need to maintain information about the object's current length, whether the object is being accessed for read or write, and whether other processes should be allowed to concurrently access the object. Since this information must be accessed by multiple processes, it must reside in global storage. The first process to access the object can allocate the storage, but how are other processes to find the virtual address of that storage? The SFCB facility addresses this problem by maintaining a table translating object UID into global virtual address. (The table is in global storage at a well-known location.) The *sfcb_$get* call takes an object UID and returns a pointer to a piece of global storage (called the SFCB). If no storage was "registered" with SFCB prior to the call, an SFCB is allocated and registered under the specified UID; otherwise, a pointer to the existing storage associated with that UID is returned and a use count field in the storage is incremented to reflect the additional "user" of the storage. *sfcb_$free* decrements the use count and, if it reaches zero, frees the storage.

Examples

Extensible Streams allows a number of special-purpose types to be defined. For example:

- History objects: objects that contain many logical versions, only one of which is presented through the open stream at a time. The residual text is used to specify a particular version; if omitted, the most recent version is presented. Useful for source control systems.
- Circular objects: objects that grow to a certain size and then have their "oldest" data discarded when more data is written to them. Useful for maintaining bounded log output from long-running programs.
- Structured documents: objects that contain document control (e.g. font and sectioning) information but which can be read through an open stream as if they were simple ASCII text. Useful for using conventional text processing tools (e.g., Unix "grep") [9].
- Gateways to non-DOMAIN file systems: objects that are placeholders for entire remote file systems. The residual is used to specify a particular file on the remote system. The manager implements whatever network protocol it chooses to access the remote system's data.
- Distributed, replicated data bases: objects that, for reliability reasons, are distributed across a network of machines. A Yellow Pages [10] manager would eliminate the need for the *ypcat*

command, and allow any ordinary user to access a Yellow Pages data base without modification and without having to bind to a special library (the type manager, in effect, is the library).

TTM can be used independently of Extensible Streams. For example, the DOMAIN graphics library may be converted to use TTM. Currently, the graphics library has code for all the display hardware types it must support. A TTM-based implementation would define multiple types, one for each type of display hardware, a trait that contains graphics operations (e.g. *move, draw, trapezoid_fill*), and a set of managers, one per type. This approach would make it possible for only the code necessary for a particular display hardware type to be loaded into the system, and for the graphics library to be easily extensible to new hardware types.

Experience

While the original Streams library was written with the idea of types and type managers in mind, the actual implementation had to be restructured substantially to take advantage of TTM. We took this opportunity to redesign the interface to managers and the interface presented to applications that use the Streams library.

The decision to implement the Berkeley socket calls in terms of a trait turned out to be a good one. On a standard Berkeley Unix system, defining and implementing a new domain (address family) is a fairly difficult task — it requires working inside the kernel. With Extensible Streams, you need only create a new type and implement the SOCKET trait in the manager for that type. We have already implemented a manager for "DOMAIN domain sockets". Currently, this domain supports only datagram-oriented sockets (SOCK_DGRAM) because our short-term goal was merely to allow access to specific, low-level DOMAIN networking primitives using the generic, high-level socket calls.

The nature of the address family space made our task a bit more complicated. Address families are identified by small integers in a space over which there is no central authority. As a result, one has to simply pick an address family out of thin air and hope no one else has picked it too. It is interesting to contrast this state of affairs with the type UID approach we took in TTM, since the small integer address families are essentially type tags. The type UID approach does not have the problem of more than one person picking the same type tag. We did not have the option to change the way address families are identified, so we used a scheme in which address families are translated into type UIDs.

The socket creation primitive is called *socket_$create_type*. This calls takes a type UID (and a socket type) and returns a stream to a socket of that type. (*socket_$create_type* is analogous to *ios_$create* except that it calls the *create* operation in the SOCKET trait instead of the *initialize* operation in the IO_OC trait.) The *socket* system call converts its address family argument into a type UID by consulting an object in the file system that contains a table translating address families into type UID. It then calls *socket_$create_type*. Note that we could have simply hardcoded a "switch" statement on address family into the implemention of *socket*, but this would have meant that *socket* would not have been as extensible as we would like. (User-defined sockets could have been created via *socket_$create_type*, but not by *socket*). The scheme we implemented is less than ideal in that it requires both that the type be created and that the address-family-to-type-UID object be updated, but it was the best we could do.

One difficult problem that we have not adequately addressed is that of expanding wildcards in an extended name. For example, using our VMS gateway type manager, one would like to type the name:

```
/gateways/my_vms_sys/dra0:[rees.*]mail.txt
```

If `my_vms_sys` is a gateway object to a VMS system, and `dra0:[rees.*]mail.txt` is a VMS file specification, this specification should be expanded to include files named `mail.txt` in all subdirectories of `dra0:[rees]`. Unfortunately, the agent doing the wildcard expansion (typically the Unix shell) has no knowledge of the syntax of the extended part of the name, and so has no way to expand the wildcard. We considered implementing a "wildcard trait," but this is difficult to specify in a general way, and every program that does wildcard expansion would have to be modified to use this trait. Instead, we require that standard Unix hierarchical names with "/" separators be used whenever wildcards are being expanded, but we also allow non–standard syntax (as in the example above) if there are no wildcards.

The semantics of certain Unix operations turned out to be fairly obscure. For example, suppose a program sets the `FAPPEND` flag (via *fcntl(2)*) to "true", then forks, then the child sets the flag to "false". Is the change to the stream state seen by the parent as well? We were frequently obliged to look at Unix kernel source or to write sample programs and run them on a standard Unix system to answer our questions. As we discuss below, we are led to believe that the task of producing exact semantic specification is a forbidding one. The various Unix standards committees have their work cut out for them if they intend to do a complete job.

Another interesting experience gained during the implemention of TTM and Extensible Streams relates to the problem of documentation. The goal of Extensible Streams is to make it possible for people who are not employees of Apollo Computer to write new type managers without having access to Apollo source code. This means that the specification of the semantics of the operations must be very precise — it must completely characterize the expectations of application programs that do I/O. The creation of this specification turned out to be a non-trivial task.

Acknowledgements

In addition to the authors, James Hamilton, David Jabs, and Eric Shienbrood worked on the implementation of TTM and Extensible Streams. John Yates was involved in some of the early design work. Elizabeth O'Connell wrote most of the documentation.

References

[1] Paul J. Leach, Paul H. Levine, Bryan P. Douros, James A. Hamilton, David L. Nelson, Bernard L. Stumpf, "The Architecture of an Integrated Local Network," *IEEE Journal on Selected Areas in Communication,* SAC–1, 5 (November 1983).

[2] Paul J. Leach, Paul H. Levine, James A. Hamilton, Bernard L. Stumpf, "The File System of an Integrated Local Network," *Proceedings of the ACM Computer Science Conference,* New Orleans, La. (March 1985).

[3] E. I. Organick, *The Multics System: An Examination of Its Structure,* M.I.T. Press (1972).

[4] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, S. C. Purcell, "Pilot: An Operating System for a Personal Computer," *Communications of the ACM,* **23**, 2 (February 1980).

[5] R. E. French, R. W. Collins, L. W. Loen, "System/38 Machine Storage Management," *IBM System/38 Technical Developments,* IBM General Systems Division (1978).

[6] Paul J. Leach, Bernard L. Stumpf, James A. Hamilton, Paul H. Levine, "UIDs as Internal Names in a Distributed File System," *Proceedings of the 1st Symposium on Principles of Distributed Computing,* Ottawa, Canada (August 1982).

[7] M. J. Rochkind, "The Source Code Control System," *IEEE Transactions on Software Engineering* (December 1975).

[8] David P. Reed and Rajendra K. Kanodia, "Synchronization with Eventcounts and Sequencers," *Communications of the ACM* (February 1979).

[9] J. Waldo, "Modelling Text as a Hierarchical Object," *Usenix Conference Proceedings,* Atlanta, Ga. (June 1986).

[10] B. Lyon and G. Sager, "Overview of the Sun Network File System," Sun Microsystems, Inc. (January 1985).