Application of Meta-Protocols to Improve OS Services

Douglas B. Orr
University of Utah
Salt Lake City, Ut. 84112
E-mail: dbo@cs.utah.edu

Abstract

User control of the implementation of OS services is the target of many current OS research efforts. The majority of these efforts require the OS to be substantially modified to expose its previously internal interfaces, with applications modified to reference these interfaces. Recent work in languages has centered on the use of meta-protocols to improve application implementations. Meta-protocols are application protocols that specify how *an application works, as opposed to* what *the application does. Meta-protocols permit application implementations to be specialized to their expected operating conditions, eliminating OS provisions for uncommon or impossible cases. We believe these specializations will result in significant performance improvements, transparent to the application. We present a set of tools for realizing this per-application specialization.* [1]

1 Introduction

The uses of abstraction and modularity relieve programmers of the burden of understanding the details of the implementations of the software components they use. By restricting access to software objects through well-defined interfaces, many possible sources of programming error are eliminated. In modular software, the internal workings of the module are not accessible to its clients, so the implementor is free to change those specifics as needed without affecting its clients. The use of general, abstract notions in software permits development of flexible, reusable programs. Abstraction permits a program to be used with specific data types that may not have been forseen by the program developer.

While the mechanisms of abstraction and modularity are a boon to software engineering, they also have some drawbacks. These mechanisms hide unnecessary specifics from application developers. These mechanisms may also inhibit the developer from taking advantage of specific system features that could improve an implementation. For example, the use of Unix file descriptors[RT78] allows an application to manipulate I/O objects in an abstract fashion; the client need know little about the object it manipulates. A file descriptor will necessarily provide a more generic interface than might best access a given object, however, since not all objects support all access methods (e.g., *"open disk file using mapping and plan on sequentially touching every page"*).

Meta-protocols[KdRB91] describe *how* a system feature works, rather than what it does. For example, the implementation of a storage allocation system may work dramatically better if the system is customized to allocate large blocks or small blocks, if the system is encouraged to optimize for space instead of time, or if the system is optimized to allocate blocks of a particular size. None of these attributes is related to the allocator interface. A meta-protocol for this storage allocation system would be used to improve its implementation without affecting its interface.

We have constructed a set of tools to describe and implement meta-protocols within our linking, loading, and Unix service facilities. This system provides the means to experiment with new implementations that are specialized to the particular configurations of services, objects, and working environments. The system uses meta-information to guide specific choices that determine the common-case a program is tailored for, and/or the uncommon cases it is prepared to handle. The system is used to optimize the execution of applications that access Unix facilities via a Unix server executing on top of the Mach[ABB+86] microkernel.

2 Meta-protocol Architecture

In this system, applications are constructed from *modules*. A module is a collection of interfaces and the corresponding program code that implements them. A meta-protocol is specified as a set of assertions about the behavior of modules (e.g., "this module doesn't spawn processes", "this module requires restartable I/O", or "this module is a long-running server"), and actions to be taken on the basis of those assertions. The assertions are made during the construction of the program from modules. Some assertions may be inferred by examining the client symbol table to determine which services it accesses; others are made explicitly.

Operations that enact meta-protocols are made at three different points in a program's lifetime: link time, exec time, and service time. At those times, some decision is fixed that would otherwise be either made at a future time or left completely undecided. For example, when a program can make use of a cus-

[1] This research was supported in part by the Advanced Research Projects Agency under grant numbers DABT63-94-C-0058 and N00014-91-J-4046, and by the Hewlett-Packard Research Grants Program.

tom service (e.g., a specialized version of an I/O package), and this condition is completely determinable at link time, that program can be statically bound to a service implementation that best matches its needs.

The system uses the OMOS server (described below) to detect conditions that permit overriding or augmenting service implementations at link time, based on the declared characteristics of the client application program. The system also provides facilities for specialized initialization and termination of programs, both in the client Unix application, and in the Unix server. And, for services whose nature is best determined at run-time, the system provides per-service enactment of meta-protocols. This facility may take the form of a specialized service provided to the client at run time. A given meta-protocol may be implemented using one or all of these strategies. None of these specializations reflect a change in the application interface.

3 Meta-protocols and OMOS

A key component for the meta-protocol system is the OMOS programmable linker/loader[OMHL93, OM92, OBLM93] which the system uses to provide our system loading facility. OMOS is used by the Unix server in our system to load programs and shared libraries.

OMOS provides a level of indirection between a module name and its implementation. OMOS maintains a namespace whose elements include executable fragments (e.g., a.out format files), and *module specifications* (also known as *meta-objects*). Module specifications are small extension language programs whose execution results in the generation of executable fragments. Although OMOS may cache these executable fragments for speed, in general, invocation of a module could result in the execution of an arbitrary series of operations to generate that module.

OMOS uses the Stk[Gal93] subset of Scheme[Ste78] as its extension language. Stk has been extended with datatypes that represent OMOS module specifications and program fragments. The primitive operations in this extension language include a set of operations designed to manipulate object file symbols and interfaces. These operations are based on a formal model described in [Bra92, BL92], that provides a basis for relating modularity and inheritance. Operands to module operations may be executable fragments or the results of executing other module operations. OMOS also defines the primitives `properties` and `has-property?`, which can be used to declare attributes of a given module, or query the existence of attributes, respectively.

With its extension language, OMOS can perform arbitrary transformations on modules. Module operations provide a powerful mechanism for replacing, hiding, overriding module interfaces, or implementing inheritance. The system uses them to wrap procedures to augment procedure functionality, to replace procedures, or to hide procedures in order to overload procedure names. Using these module operations, the standard Scheme facilities and the persistence of the OMOS store, OMOS can perform conditional linking and adaptive transformations that are the mainstay of link-time meta-protocols. In addition, OMOS generates the support code that establishes exec-time and service-time portions of meta-protocol implementations.

/bin/ls:

```
;; name our modules and libs
(let ((obj-list '("/lib/crt0" "/obj/ls.x"
                  "/lib/libc")))

  ;; merge them all together
  (merge (map resolve obj-list)))
```

Figure 1: Sample Module Specification

Figure 1 demonstrates a simple module specification that links a Unix application (`ls`) with an implementation of `crt0` and `libc`.

4 Examples of Meta-protocols

Specializing the use of file descriptors is a good example of an application of a meta-protocol optimization in Unix. File descriptors provide a useful abstraction for accessing I/O devices. The full file descriptor semantics greatly limit the options for providing enhanced implementations of their interface, however. For example, most of the semantics associated with file descriptors work nicely when implemented in a user-space client library, since user-space implementations may be able to avoid calls to Unix, either via local caching, or redirection. However, file descriptor semantics require that the file descriptor state be maintained across calls to the `fork()` or `exec()` system calls, rendering it difficult to implement them in user-space.

Through use of of meta-protocols, implementation tools can choose their fights prudently. Within this system, those modules that don't use the process spawn facilities can be declared to have the property `no-spawn`. The C library shown in Figure 2 provides kernel and user-space implementations of its I/O primitives. The specific implementation of the low-level I/O portion of the library is chosen based on whether the `no-spawn` property is present or not in the module with which it is being linked[2]. This decision is an example of a *link-time* specialization.

The user-space file descriptors found in this smart C library can be used to implement various optimizations. To complement the user-space file descriptors, the Unix server provides a variant of the file `open` call that returns file type information and some constructor data that can be used to construct an instance of a

[2]In reality, this example is complicated by the fact that the user-space version of the fd library will almost certainly need to reference the kernel fd services, and react to whether libc services reference `exec` or `fork`. OMOS provides for this possibility by, essentially, deriving the user file descriptor class from the base kernel file descriptor class. The details are omitted for brevity.

/lib/libc:

```
(deflib (m)
  ;; name our modules and libs
  (let ((common-list '("/obj/printf.o"
                       "/obj/scanf.o"
                       "/obj/stat.o" ...))
        (kern-fds '("/obj/open.o"
                    "/obj/read.o"
                    "/obj/write.o" ...))
        (user-fds '("/obj/u_open.o"
                    "/obj/u_read.o"
                    "/obj/u_write.o" ...)))
    (merge m
      (cl-project m
        ;; merge "m" with the
        ;; modules we reference out of...
        (map resolve
          (append common-list

            ;; special i/o libs
            (if (has-property? 'no-spawn m)
              user-fds
              kern-fds)

            ;; wants page-aligned allocations
            (if (has-property? 'page-io m)
              "/obj/page_malloc.x"
              "/obj/malloc.x")))))))
```

Figure 2: Simple Libc Transformation

user-space I/O manager. These I/O managers implement instances of *service-time* specializations. Based on some property descriptions, OMOS encodes attributes in executables which are conveyed to the Unix server at program startup time. These attributes may guide *service-time* optimizations; they are examples of *exec-time* specializations.

For example, applications that expect to issue moderate to large read requests are annotated with the attribute `page-io`. At *link-time*, these applications are linked with a special version of `malloc` that returns page-aligned buffers for large allocation sizes. This attribute is then asserted to the Unix server at *exec-time*. Subsequently, if this application opens large files using the user-space file I/O system, the fact that it has declared its interest in `page-io` will guide the Unix server to return a type whose manager specializes in doing large I/O via vm operations. On execution, the common case for which I/O is optimized would match up with the typical data. Figure 3 demonstrates use of the `page-io` annotation within the Unix `cat` program.

/obj/cat.m:

```
(properties '(no-spawn page-io)
  (resolve "/obj/cat.x"))
```

/bin/cat:

```
(let ((obj-list '("/lib/crt0" "/obj/cat.m"
                  "/lib/libc")))

  ;; merge them all together
  (merge (map resolve obj-list)))
```

Figure 3: Annotated Unix "cat" Program

As another example, we can address the traditional softare engineering problem of maintaining multiple, similar versions of a module which handle slightly different variations of a given interface. For example, the Mach `malloc` call requires a slightly different implementation if it is to be used in a multi-threaded environment. By establishing a mechanism for declaring attributes of a program, this framework provides a natural way to select variations of library routines on a functional basis. The Mach implementation of a malloc library would check to see if it is being linked with a multi-threaded program and return the appropriate implementation.

5 Discussion

In all the given examples, the decisions made at specialization time could have been replaced with some sort of a run-time check made by a more complicated version of a given service. One could imagine a user-space `read` that queries the size of the file and does different things based on the file size and amount of data requested, and also transmits its relevant state back to the Unix server if the process tries to `fork` or `exec`. This service, however would end up doing unnecessary work, if the appropriate path could be established earlier in the application's lifetime (at link or exec time, for example), or with fewer operations. In addition, the code base of this sophisticated `read` routine might include I/O algorithms for different situations that were be totally unrelated and non-overlapping, violating software engineering principals. By elevating these decisions to the meta-protocol level, however, we can provide appropriate implementations, avoiding the extra run-time operations and checks often needed to choose an appropriate code path.

To implement the meta-protocol, it is necessary to have a sufficient understanding of the internal workings of the given module's implementation that its different operational characteristics can be identified and separated out. The implementor of a module should be able to quantify these characteristics as they often play a crucially important role in the actual use of a module. Meta-protocols provide a valuable mechanism for describing and acting upon these characteristics.

6 Related Work

A number of Unix emulation projects could benefit from the use of meta-protocols to restrict the problem domain of their implementations. The Apollo Domain system used user-space implementations for I/O objects[LLD+83, ROS87]; it used a complicated protocol to marshal and unmarshal file descriptor state before and after `fork` operations. The GNU Hurd[Bus94] project is following a similar path. The

Mach Multiserver[JCS+91] project experienced similar issues in the domain of Mach ports. Related work was done by [Jon90] exploit the use of different implementations of system calls as a way of specializing or extending system behavior.

A great deal of work has gone into understanding meta-protocols in the LISP community[KdRB91]. Much work is currently underway in the operating system community to provide more complete descriptions of the service requested to service providers[BCE+94, DPH92].

We have presented a system for implementing meta-protocols in an extended Unix system, using link-time, exec-time, and run-time support. We have demonstrated how this system could be applied to problems specific to several Unix services. Since meta-protocols are designed to describe *how* a service is to used, independent of *what* the service is, they provide the necessary entreé for systems implementors to specialize the implementations of their services to match the needs of their clients. Many different kinds of specializations could be implemented in the context of meta-protocols and this promises to be a useful tool for future investigation.

Acknowledgements

We would like to thank Jay Lepreau and John Carter for their help in reviewing this document.

References

[ABB+86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proc. of the Summer 1986 USENIX Conference*, pages 93–112, June 1986.

[BCE+94] Brian Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemyslaw Pardyak, Stefan Savage, and Emin Gun Sirer. SPIN – an extensible microkernel for application-specific operating system services. Department of Computer Science and Engineering TR-94-03-03, University of Washington, March 1994.

[BL92] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *Proc. International Conference on Computer Languages*, pages 282–290, San Francisco, CA, April 20–23 1992. IEEE Computer Society. Also available as Technical Report UUCS-91-017.

[Bra92] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* PhD thesis, University of Utah, March 1992. 143 pp.

[Bus94] Michael I. Bushnell. Towards a new strategy of OS design. In *GNU's Bulletin*, Cambridge, MA, January 1994.

[DPH92] Peter Druschel, Larry L. Peterson, and Norman C. Hutchinson. Beyond microkernel design: Decoupling modularity and protection in Lipto. In *Proc. of the 12th International Conference on Distributed Computing Systems*, pages 512–520, Yokohama, Japan, June 1992.

[Gal93] E. Gallesio. Embedding a Scheme interpreter in the Tk toolkit. In Lawrence A. Rowe, editor, *First Tcl/Tk Workshop, Berkeley*, pages 103–109, June 1993.

[JCS+91] Daniel P. Julin, Jonathan J. Chew, J. Mark Stevenson, Paulo Guedes, Paul Neves, and Paul Roy. Generalized emulation services for Mach 3.0 — Overview, experiences and current status. In *Proc. of the Second USENIX Mach Symposium*, pages 13–26, 1991.

[Jon90] Michael B. Jones. A toolkit for interposing user code at the system interface, May 1990. Thesis Proposal – Carnegie Mellon University.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, Cambridge, MA, 1991.

[LLD+83] P.J. Leach, P.H. Levine, B.P. Douros, J.A. Hamilton, D.L. Nelson, and B.L. Stumpf. The architecture of an integrated local network. *IEEE Journal on Selected Areas in Communications*, SAC-1(5):842–857, November 1983.

[OBLM93] Douglas Orr, John Bonn, Jay Lepreau, and Robert Mecklenburg. Fast and flexible shared libraries. In *Proc. USENIX Summer Conference*, pages 237–251, Cincinnati, June 1993.

[OM92] Douglas B. Orr and Robert W. Mecklenburg. OMOS — An object server for program execution. In *Proc. International Workshop on Object Oriented Operating Systems*, pages 200–209, Paris, September 1992. IEEE Computer Society. Also available as technical report UUCS-92-033.

[OMHL93] Douglas B. Orr, Robert W. Mecklenburg, Peter J. Hoogenboom, and Jay Lepreau. Dynamic program monitoring and transformation using the OMOS object server. In *Proceedings of the 26th Hawaii International Conference on System Sciences*, pages 232–241, January 1993. Also available as technical report UUCS-92-034.

[ROS87] Jim Rees, Margaret Olson, and J. Sasidhar. A dynamically extensible streams implementation. In *Proc. of the Summer 1987 USENIX Conference*, pages 199–207, Phoenix, AZ, June 1987.

[RT78] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 57(6):1905–1930, July/August 1978.

[Ste78] G.J. Steele, G.L.; Sussman. The revised report on scheme, a dialect of lisp. AI Memo 452, MIT AI Lab, Cambridge MA, January 1978.