Ada Implementation of an X Window System Server

Stuart Lewin

Sanders Associates, Inc.
P.O. Box 2035
Mailstop MER15-1120
Nashua, NH 03061-2035
lewin@savax.Sanders.com

ABSTRACT

Sanders is in the second year of a two year project to implement an X Window System server using the Ada programming language. X is a highly portable, network transparent display management system which was developed at MIT's Project Athena and has emerged as the industry standard for windowing systems. Our objectives are to implement a production-quality base windowing system suitable for use in Ada-based real-time systems, and to examine Ada's applicability as an implementation language for graphics software. We are currently running a partially implemented server which dispatches incoming protocol requests, performs basic windowing operations and executes basic graphics functions. This paper explains why we chose to do an Ada implementation, describes our implementation approach and server design and relates several lessons learned about using Ada in such an application.

INTRODUCTION TO THE X WINDOW SYSTEM

A display management system (of which the X Window System[1] is an example) is analogous to the operating system of a general purpose computer. Display management systems provide a centralized mechanism for the sharing of resources between potentially competing users. In much the same way that the operating system manages access to processor cycles, peripheral devices and file systems, the display manager manages screen space, colors, fonts, cursors, and any input devices attached to a workstation.

A windowing system is a form of display manager that implements the desktop metaphor. The desktop metaphor gets its name from the analogy between the screen and a user's desktop. In the same way a person would shuffle papers around on a desk while performing daily duties, the workstation running a windowing system provides multiple windows (the "pieces of paper") that can be resized, moved and restacked as the user changes focus in the course of working. This is the most common user interface running on workstations today.

The X Window System was developed at MIT's Project Athena. It is an ambitious project to link together the 10,000 or so various workstations on the MIT campus with a single, unified network. The X Window System, or simply "X", was developed to provide a standard, workstation independent interface for application programs as well as the student user community. Because it was to be used in a heterogenous networked environment, the designers hid the implementation details, increasing portability and providing network transparency.

Version 11 is the most recent implementation of X, and the one which is currently undergoing standardization. During its development at MIT, the following design goals were established:

1. High Portability - The system had to be highly portable to be used with the variety of workstations found at MIT. This portability was to be in the areas of (1) graphics hardware it supported, (2) network communication protocol it ran on, and (3) host operating system it ran under. This has been achieved,

[1] The X Window System is a trademark of the Massachusetts Institute of Technology.

as may be seen by the wide variety of X implementations currently available. All the major workstation vendors have ported X to their machines, and there are versions running on UNIX[2], VMS[3] and even Apple's native Macintosh operating system.

2. Good Performance - The system had to have performance suitable for use in what may be one of the most demanding user environments: students using computer aided instruction (CAI) applications. As MIT migrated towards the use of computer simulations to replace laboratory exercises for teaching concepts by example, the underlying display software had to be fast and responsive in order to make this feasible.

3. Extensibility - At the time the systems engineering was underway on the development of the protocol definition for Version 11 of X, the architects realized that they were not going to be able to predict all users' needs or foresee all the capabilities which might be required to meet those needs. In order to provide for future extensions to the core X functionality which remained compatible with the base system, a standard mechanism needed to be provided. This goal was met, and several "standardized" extensions are already under development (including double buffering and one for the use of PHIGS/PHIGS+[4] inside an X window).

4. "Mechanism, Not Policy" - In order to provide maximum flexibility in its use, the system was designed to provide a wide enough variety of underlying mechanisms to allow implementation of almost any user interface policy on top of it. This goal was acceptably met, as is evidenced by the large number of different window managers that run on top of X, each implementing its own, sometimes widely divergent, man-machine interface.

In summary then, the X Window System is a working, well-tested, highly portable, public domain, multidisplay, network-model windowing system. In addition to the protocol specification defining X, a sample, reference implementation (developed at Project Athena) is distributed by the X Consortium at MIT. This reference implementation is written in the C programming language and runs in a UNIX environment. The reference implementation supports graphics hardware from Apollo, Apple, DEC, IBM, Silicon Graphics and Sun Microsystems.

SYSTEM OVERVIEW OF THE X WINDOW SYSTEM

In order to establish a common understanding of Version 11 of the X Window System, a short synopsis of its salient features follows. X consists of two parts, an application interface called Xlib and the actual display manager program called the server (see Figure 1). Applications tasks (referred to as clients) communicate with the X server via a byte stream protocol which is accessed by the procedural interface contained in Xlib. Clients can either reside on the same processor as the server or on other processors within the network. Xlib converts the various window system requests into

[2] UNIX is a trademark of AT&T Bell Labs.

[3] VMS is a trademark of Digital Equipment Corporation.

[4] PHIGS, or the Programmer's Hierarchical Interactive Graphics System is a draft ANSI standard graphics interface and a draft ISO standard which includes support for 3D. PHIGS+ extends this standard to include support for light source shading, depth-queuing and additional output primitives.

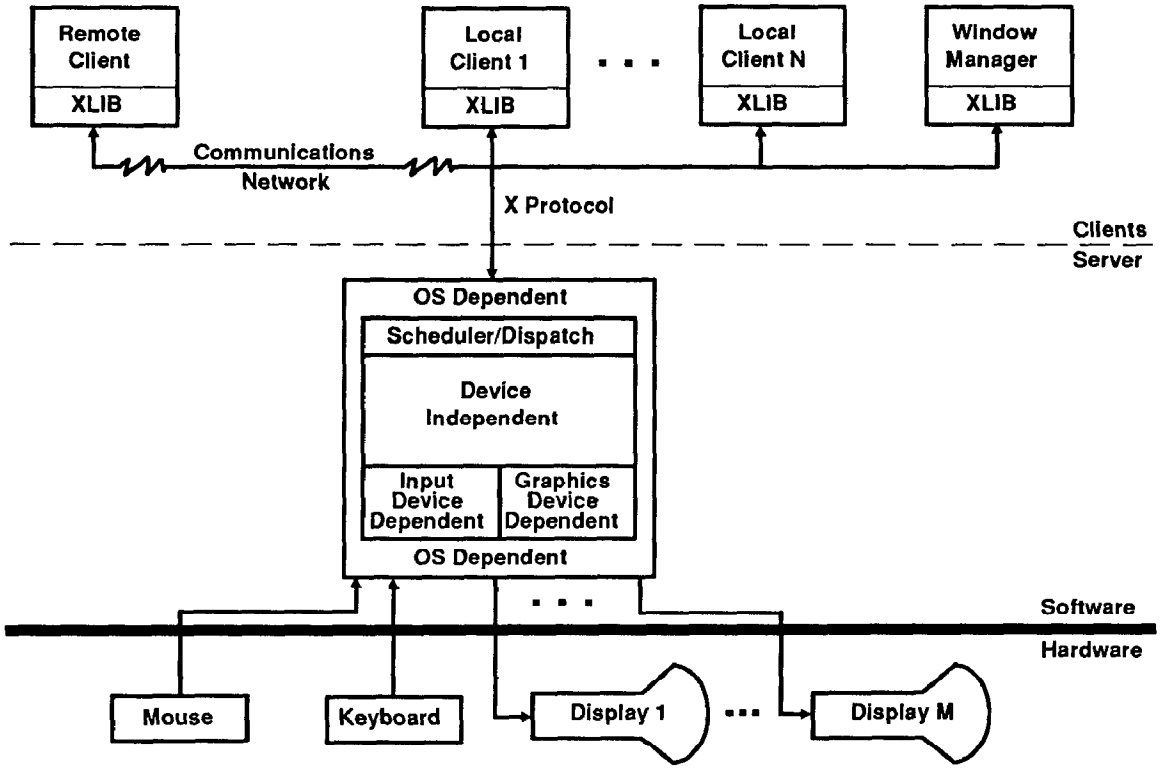

Figure 1. X Window System Model

correct op-codes and data formats, and manages the buffering and transmission protocol clients use to communicate with the server.

The X server itself provides two major services to clients, display management and input management. In addition, it provides a large number of related services, including window positioning (both physically on the display and within the hierarchy)[5], irregularly shaped window clipping (due to overlapping, opaque windows), resource management (fonts, cursors, colormaps, etc.), and a collection of services designed to support window managers and interclient communication.

Structurally, the server can be thought of as consisting of three layers: the scheduler or dispatch, the device independent, and the device dependent. These three layers, which exist for both display and input management, are encased by an operating system dependent shell.

In the display management portion of the X server, the scheduler is responsible for fairly handling X protocol requests from multiple clients and dispatching them to the device independent layer. This layer performs any required device independent portions of the operations before passing the graphics requests on to the device dependent layer. Here, whatever processing necessary to display the graphics requests is performed. For frame buffer graphics hardware, the rendering is performed by the host processor; for smarter graphics processors, the requests can be converted to a form understood by the processor and passed on to it for rendering, relieving the host of this computational burden.

In the input management portion of the X server, the device dependent layer handles the interface to the devices (an alphanumeric keyboard and a cursor position and selection device), passing input events to the device independent layer. There, the events are converted as necessary to make them conform to the X protocol and are passed to the scheduler for distribution to the clients. The input manager also provides the cursor management on the display(s).

The operating system shell is responsible for the interface between the server and the operating system under which the server is running. This shell is the same for both the display and input management sides of the server. Interfaces to the client-server communication mechanism are provided here, as well as to the file system and memory management routines. I/O routines for the various input and display devices are also contained here.

WHY DO A SERVER IMPLEMENTATION IN ADA?

Information Systems Division at Sanders Associates has been building display-oriented systems for over ten years. Time and time again, we have had to work with requirements for some form of display management software, which was always custom implemented. Because of the types of systems we build, it is highly likely we will continue to have similar requirements in the future.

In one recent program, Sanders was again faced with display management requirements. In order to avoid implementing yet another custom solution and to capitalize on the benefits realized through their use, we wanted to proposal a standards-based solution. After performing a trade off study to examine alternative windowing systems, we concluded that the X Window System provided the optimal solution. It has sufficient performance capabilities, no unacceptable limitations and offered a natural path for future extensibility.

Since that time, it has become even more evident that X can be applied to a wide variety of applications. Rather than continuing to invent and apply unique solutions to each new system, we want to be able to capitalize on the advantages and benefits obtained through the use of X in our future system design. This meant that Sanders had to come up with an X server suitable for use in the type of systems we build (increasingly Ada-based, with unique requirements for graphics hardware and host environment).

Another pragmatic reason for using X is that our government customers are becoming very interested in X. Along with the push for the use of Ada as the implementation language of choice for future software systems, we have seen several government agencies requiring X as the graphics interface. We would expect this trend to continue as they increase the emphasis placed on standards. This project incorporates both requirements into a single solution.

PROJECT ADDRESSES SEVERAL KEY PROBLEMS

Sanders' Independent Research and Development (IRAD) project to implement an X Window System server in Ada simultaneously addresses four areas concerning software development and the use of standards on Ada programs. As software development and maintenance costs have continued to rise at ever increasing rates, a push for the use of standards as a solution to the problem has developed. The Department of Defense (DoD) has addressed the problem in the use of programming languages through its mandate for the use of Ada. DoD Directives 3405-1[6] and 3405-2[7] both require Ada as the programming language for future real-time, embedded systems. The use of a standard programming language, as with other standards, increases portability and maintainability of software.

The newness of Ada, however, has created a shortage of existing code for (re)use on programs. This is especially true of software standards. Although some Ada implementations of standards are becoming available, the low quality and general lack of availability of these implementations make it difficult to produce systems programmed completely in Ada. The resulting hybrid systems are in direct conflict with the DoD mandates and serve to minimize the advantages realized through the use of a single, standard programming language. These advantages include reduced non-recurring engineering experience, shorter development cycles, decreased software maintenance costs and increased system extensibility and modifiability. By developing systems implemented entirely in Ada, these advantages could be realized.

[5] In the same way papers can be stacked on a desk, a windowing system allows the ordering of windows such that those closest to the user obscure the ones below (they overlap).

[6] "The Ada programming language shall be the single, common, computer programming language for Defense computer resources used in intelligence systems, for the command and control of military forces, or as an integral part of a weapon system."

[7] "... waivers from using Ada, [shall be permitted] only on a specific system or subsystem basis. For each proposed waiver, a full justification shall be prepared."

Another problem introduced by the newness of Ada is a lack of practical experience in several key areas. For example, effective use of Ada in solving graphics problems with demanding DoD response time requirements which lie close to the hardware will come about only when the issues involved are understood. MITRE raised this concern over lack of experience in a study for the FAA regarding the use of Ada on the Advanced Automation System[8]. The study was undertaken by MITRE to examine whether it was feasible to require the use of Ada in the implementation of a very large real-time system. One of the risk areas identified by the study concerned the feasibility of doing the real-time graphics required on AAS using Ada. This risk was identified due to a perceived lack of practical experience in the area. Other areas which caused MITRE concern included the implementation of both low-level device drivers and Boolean bit manipulation using Ada. The IRAD project to implement an X Server in Ada is providing Sanders with valuable hands-on experience in these areas, decreasing our risk on future Ada programs with significant graphics content and real-time performance requirements.

The third problem area involves the difficulties associated with program integration and maintenance in mixed language environments. Although it is possible to obtain language waivers for existing code, as we had done when planning to use a C implementation of the server, many problems can occur while trying to incorporate even a single non-Ada program into an otherwise native Ada environment. This is especially true when porting the non-Ada program to the bare machine runtime environments typical of Ada real-time systems. Multiple languages also require multiple toolsets for the host hardware, as well as multiple knowledge bases in the programming staff. By implementing a server entirely in Ada, we eliminate these problems and make a total Ada solution possible.

The final problem area addressed by our project concerns the maintainability and reliability of the reference server distributed by the X Consortium. Although much effort went into developing a usable and stable reference implementation, it is just that: a "reference" implementation. The server was implemented in the C programming language by several diverse groups, with no accompanying coding standards or design documentation produced. And although there is now a consortium of companies[9] dedicated to maintaining and extending the X Window System, this group has claimed no real responsibility for correcting problems that may crop up in the reference implementation.

The reference implementation contains ports to common workstation platforms, but performance optimization was not a factor in its development. In fact, its performance is relatively poor as a result of the goal to provide ease of portability to new target graphics hardware. In order to meet that goal, the server was implemented in a very general fashion, without making use of any hardware dependent speed ups. In addition, because it serves as the reference implementation, the main focus was to ensure correctness rather than performance. The reference implementation is also currently supported only under the UNIX operating system. Sanders will have requirements to retarget the server for other graphics hardware and port it to different operating systems depending upon the system under development.

[8] *Use of Ada for FAA's Advanced Automation System.*

[9] The X Consortium, established in January of 1988, currently consists of over 50 members and is based at MIT. This includes all the major workstation and graphics hardware vendors.

In recognition of the DoD objectives, and to favorably position ourselves in a competitive environment, Sanders undertook the Ada implementation of an X server. This will also allow us to improve our capability to maintain and retarget the server, as well as increase our responsiveness to unique customer requirements.

PROJECT PLAN AND APPROACH

Sanders is currently in the second year of a two year IRAD project[10] to implement an X server in Ada. Last year we expended slightly less than one man-year's time (using two engineers). This year we have staffed up to five engineers, and are planning to complete the project with a total of about 4 man-years of effort expended.

The project can be divided into four distinct efforts. In the first effort, currently underway, a complete Ada implementation of the X server is being produced. We are employing formal Ada design methodologies during the top level and detailed design phases. During the design phase, Ada packaging provides us with a natural route to produce a compilable and integrated design. The implementation phase then consists of completing the subprogram bodies.

The protocol specification for X defines the requirements for the server design. It establishes the structure and content of all traffic between the server and clients, as well as the semantics associated with each protocol packet. We are also using the reference implementation of the server contained in Release 3 from the X Consortium for guidance; however, the protocol specification provides the final word on what the requirements are.

The project is also serving as a practical proving ground for the various design and implementation methodologies being espoused for use with Ada within Sanders. Object oriented design[11] is commonly used with Ada systems. We have combined that with a methodology for real-time systems design developed at Hughes Aircraft Company.

The second effort consists of validating the server implementation for correctness. A series of validation tests and benchmarks will be run, and the implementation changed as necessary to correct any problems. These tests will come from the test suite being developed by the X Consortium to validate X implementations (in a manner similar to the way Ada compilers are validated using the ACVC Test Suite). In addition, since the protocol is language independent, it will be possible to run the set of clients distributed on the release tape against our Ada server. These clients include four different window managers, a terminal emulator, some demonstration programs and several games. The clients are varied enough in application to provide coverage of the majority of protocol requests, giving further verification of the correctness and robustness of our implementation.

The third effort is aimed at improving the performance and implementation quality of the server. One of the goals of the project is to gain insight into the use of Ada for near real-time graphics and display management in an environment that is close to the underlying hardware and run-time system. In order to meet that goal,

[10] DI-88-26 and IS-89-88, Ada Implementation of an X Window System Server.

[11] Actually, a subset of OOD referred to as data abstraction and presented by Grady Booch in his book, *Software Engineering with Ada.*

we are undertaking a 100% Ada implementation to uncover any mismatches of the language with this application. During the testing and validation stage of the project, however, areas may be uncovered where alternate implementations need to be considered in order to meet the required performance. Although we will be revisiting these areas for alternate Ada implementations as dictated by performance or functionality concerns, some areas may still require eventual implementation in some other language. Clarification of these areas will enable us to predict where we will have to ask for language waivers or make special plans on projects involving this type of problem domain.

As mentioned previously, one of the design goals of X was to provide for extensibility. In order to demonstrate that our Ada implementation is extensible and to make it more usable in government applications, we will be examining and implementing selected extensions which typically would be used in systems we build. These extensions include features such as transparent windows (X currently only supports opaque and input-only windows) and a method for display recording and playback (coincidentally, the server provides a natural synchronization and pick-off point for this function), as well as the standard extension to support double buffering (which smoothes out the display presentation on a large screen).

The fourth and final effort involves documenting the results of the project. In addition to the final project report required, we are producing a subset of MIL-STD-2167A documentation to document our design and implementation. A set of Software Development Folders (SDF) is being kept, and the Software Design Document (SDD) and Interface Requirements Specification (IRS) will be written. Only a subset of the full set of documentation in 2167A is being produced to keep the IRAD costs down. However, it will be sufficient to support use of the software on future projects within Sanders and Lockheed.

The project is using a compiler self-hosted on a network of Sun-2 and Sun-3 workstations. Although the server is being developed using a single vendor's compiler, we are following established guidelines to ensure the portability of the finished server.

The target hardware platform we are initially using is a Sun-3 processor with a standard Sun mouse running UNIX. Many of our customers want high performance graphics that necessitate using a graphics processor. In addition, based on previous performance analysis (as discussed in the *Ada "Got-cha's"* section), we anticipated problems with using Ada to generate graphics in a memory mapped frame buffer. We decided early on to avoid the performance problems on this IRAD project by using a graphics processor.

During our efforts on a previous program where Sanders had proposed the use of X, we went through an exhaustive specification and selection process to obtain the best match of X device-dependent functionality in the selected graphics hardware. At the conclusion of the competition, Matrox Electronic Systems, Ltd. of Canada was selected. Because their boardsets also provide a suitable interface for the Ada server, we are targeting a Matrox VG-1281 graphics processor connected to a Sony 1280 x 1024 color display.

CURRENT STATE OF AFFAIRS

The most common way of measuring progress on any software project is through line of code counts. As of mid-July, we have 25,400 Ada statements contained in 639 library units (package specs, package bodies and separate subprogram bodies). Because Ada tends to be a verbose language, and we are using a style guide that requires in-line program design language and comments, this represents 93,700 actual source lines.

Our approach to implementing the server consisted of starting at the core with the protocol dispatch and device independent resource type managers. This is where the bulk of the server code resides and has allowed us to defer the implementation of the operating system and graphics hardware-dependent portions. This strategy allows us to get the core of the server up and running while deferring some of the more difficult implementation issues. At this point, we have completed the protocol dispatch packages, as well as the corresponding resource type managers. We have concentrated on the threads for displaying and configuring windows on the screen and for displaying rudimentary graphics. This allows us to get a demonstratable server running relatively early in the project.

We also have the interface at the device-independent to device-dependent display portion defined and stubbed out. As discussed previously, we have decided to target a graphics processor. Although we have a simple UNIX device driver (written in C) in place to open the board and memory map it into the Ada server's address space, the rest of the code to handle the board is being developed in Ada. Several of the device-dependent portions of the resource type managers have been implemented and a library of routines to send commands and parameters to the board is complete.

Two important tools have been developed to aid in the testing and integration of the server. The first is an interactive program which allows the generation (and optional recording) of X protocol packets. This program also takes input from ASCII command files, which simplifies the generation of the protocol "test scripts". The second tool is a special interface to the server's dispatch processing that allows prerecorded protocol to be injected directly into the server, bypassing the client-server communication mechanism. This has allowed us to get the core part of the server up and running in parallel with the outer layer of interfaces to the clients and operating system. It also allows for greater repeatability of "client" inputs during implementation and testing.

OVERVIEW OF ADA DESIGN

Although the exact nature of our server design remains proprietary at this point, a brief description of the top level design and protocol processing follows.

Figure 2 shows the Boochgrams for the core of the server processing, or that represented by the dispatch process. It can be thought of as consisting of three layers. The top layer contains the packages which deal with the various kinds of protocol packets. The approximately 120 different kinds of protocol have been grouped by the server resource to which they are related. In these routines, each protocol packet is checked for lexical correctness. Then, one or more operations in the resource managers themselves are called to actually perform the work.

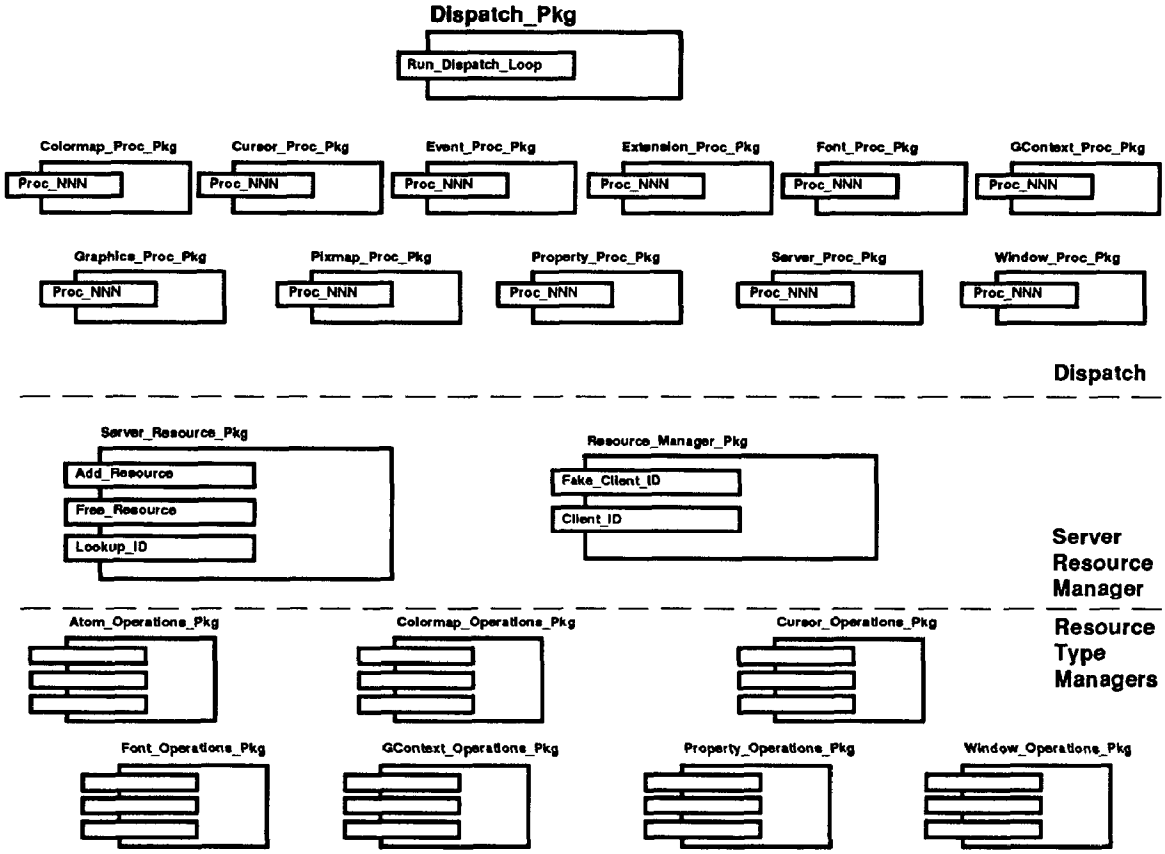

Figure 2. Device Independent Dispatch Packaging OOD

The second layer consists of the server resource manager. One of the key changes to the protocol in X Version 11 was to make resource creation asynchronous, eliminating a round trip in the protocol. Because of this, the application program (or client) only knows about its resources through the ids that it assigns to them. As the server may use a different notation to identify a resource (for example, the access value to a dynamically created resource), some method must be provided to map between client ids and server resources. The server resource manager provides operations to Add and Free resources, as well as to look up a resource for a given ID from a client. This package also helps to prevent errors caused by using access values to resources that have been destroyed.

The third layer consists of the various resource managers themselves. We have implemented the resource managers in the server using a combination of both type and object managers. Most of the server resources (windows, cursors, graphics contexts, etc.) are implemented as dynamically allocated objects. Server resources tend to have fairly long life times and there is no way to reliably guess the number required ahead of time in order to statically pre-allocate them. Access functions to set and retrieve components of the resource are implemented as Pragma Inline to preserve the data abstraction while minimizing the overhead.

Figure 3 shows a Boochgram for the program structure involved with processing a typical protocol request dealing with a resource. In this case, the server dispatches to the correct protocol processor based on the message content. For this example, we will assume it is to create a window.

The first step is to convert the request as defined by the standard header into the specific request type. This is accomplished via unchecked conversion. The protocol processing subprogram and any associated operations from the individual resource managers are then responsible for checking the integrity of the received data. The data checking is necessitated not only by the unchecked conversion, but also because the protocol is received from a source external to the Ada program. Thus, the use of unchecked conversion does not result in any additional work.

One of the checks performed in this case is to verify that the resource id provided by the client is valid. As mentioned earlier, clients allocate resource ids and, before accepting an id, the server must ensure that it is not already in use and is really an id from the requesting client. After other checks for a valid request length and the right number of attributes being supplied to override the defaults, the protocol processor calls the window resource manager to create a window.

In the window operations package, a structure for the window is dynamically created. We are using dynamic memory allocation for some objects as they tend to have fairly long lifetimes and there is no way to calculate the number required during either system build time or at startup. The values sent by the client for attributes of the new window are checked for validity and stored in the window object. The window is then returned to the protocol processor.

The last phase of creating a new window is to store the association between the window's id (as it will be known to the client) and the

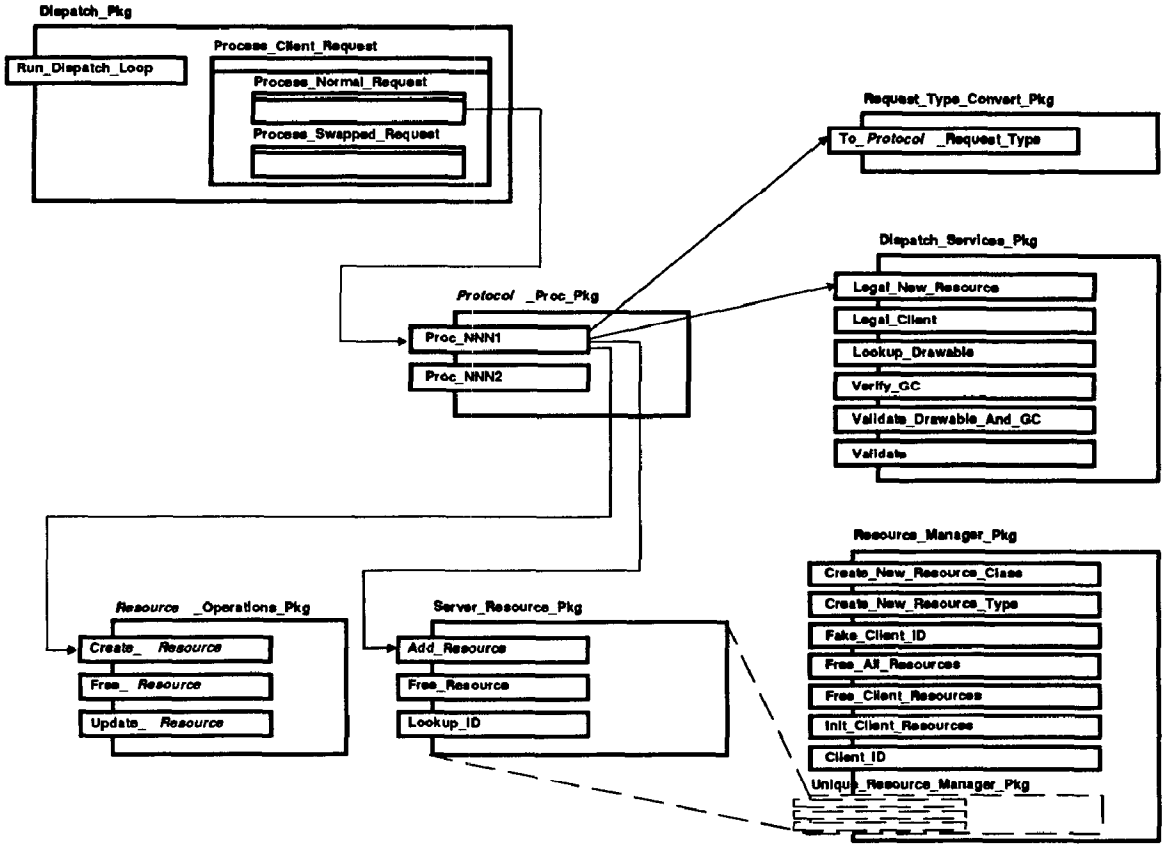

Figure 3. Typical Protocol Processing OOD

object itself (as the server will operate on it). The server resource package consists of a series of instantiated generics for each type of resource. In this case, a call is made to add the resource to the database. At a future point, calls to lookup the object or free it when done can be made.

Processing now returns to the dispatch loop, where the next client request is obtained and the process starts over again. If at any point errors are uncovered (in values received from the client) or local resources are exhausted (in memory allocations), exceptions are used to cause an error message to be sent to the client and processing of this request to be terminated.

DIFFERENCES BETWEEN C AND ADA DESIGNS

There are four major areas of difference between the existing C reference implementation and our Ada implementation. The first is in the area of error handling capabilities. The reference C server distributed by MIT has its internal interfaces defined to return error indications where required. However, in many instances, this information is not used by the calling routine. In fact, C's capability to use a function returning an error condition as a procedure when it is called, and thereby ignore the returned error condition, is prevalent in the reference server code. This has introduced the unpredictability and fragility one would expect, especially in the case of failed operations on a resource. Ada, on the other hand, makes no such allowances, forcing the calling routine to at least acknowledge the error condition.

In addition, the use of Ada exceptions to propagate unexpected error conditions has become a prominent feature of our implementation. Instead of implementing the interfaces as functions that return a status, interfaces can be defined as procedures with exceptions raised on errors. With the use of exceptions, error conditions must be dealt with at the point of occurrence, and cannot be ignored. This improved method of dealing with internal program errors will provide a substantially more robust server, and increase the possibility of recovering from errors that would have been fatal in the existing C implementation. Examples of this include the ability to back out of failed new resource allocations, or operations on a resource that fail at the beginning of a block of code.

The second area of difference concerns the methods used to implement type abstractions supported by the two languages. Ada is a strongly typed language; C is not. This accounted for numerous problems due to two major factors. The first of these is that incomplete type declarations are allowed to be more general in C than in Ada[12]. This means that the format and content of Ada's abstraction of an object will not match the C representation. The second factor was that various types had different sizes in the protocol and the internal data structures used to store them (and even between different data structures). Although this does not present a problem to the completed full Ada implementation (since it is internally consistent), trying to utilize existing C routines for integration during development was made impossible.

[12] The C programming language allows pointers (equivalent to Ada's access types) to be declared without giving, in the same compilation unit, the content or makeup of the structure they point to.

Our original intent was to first implement our system framework design in Ada and plug in existing C code where possible to do the actual work. In this way, we would have been able to prove our design at an early stage and have a working implementation of the server at all phases of the project. Each package of operations would then be designed and implemented in Ada to replace the existing C code. However, it quickly became obvious this would not work due to the incompatibilities of the object representations between the two languages.

One possible solution to the problem was to write small intermediary routines to pack/unpack the Ada and C data structures and provide the required compatibility. Upon closer examination, we decided that even this approach would be infeasible due to the vast differences in the two representations, and the large amount of "throw away" code it would require. We feel that the strong support for typing provided by Ada will make our implementation more robust, with increased portability, and the problems inherent with implicit and explicit type casting will be avoided.

A third area of difference is the support for concurrency that is supplied by the Ada language itself. The capability of providing a multithreaded server presupposes the availability of light-weight processes, and our server design, by incorporating tasking, provides the foundations for this feature. This capability is not possible with the existing C implementation and requires a non-standard interface to the underlying run-time environment if implemented using kernel features. A multithreaded server approach allows for the prioritized scheduling of client requests, rather than the round robin approach now used in the C server. With this capability, a server could be tuned to prioritize its display activity.

The final major area of difference is concerned with the use of dynamic binding and inheritance in an object oriented design. This is one area of true object oriented design that is unsupported in Ada. The C implementation follows an object oriented approach to its design. This allows the processing operations performed on various resource objects within the server (windows, graphics contexts, etc.), as well as operations to actually render graphics objects on the display, to change as the attributes of the object change. This capability is provided through extensive use of "subprogram pointers"[13]. Another area where this capability has been exploited is in the generation of vector tables for dispatching on various items.

Ada, on the other hand, provides no such capability and requires calling sequence and semantics to be determined at compile time. Therefore, we have had to resort to alternate implementations. In most instances, the dispatching must be handled by using large case statements. Depending on the compiler quality, this can introduce a large amount of overhead. This is a special problem in the protocol dispatch area, as that is the most heavily exercised part of the server. One side effect that has fallen out of this fundamental difference between the two programming languages is that we were given further "encouragement" to do a reimplementation in Ada, rather than a straight code conversion.

[13] The C programming language allows calls to subprograms via an address that has been previously stored. By changing this stored address, different implementations with the same syntactic interface can be provided.

ADA "GOT-CHA'S"

One of the objectives of this project is to examine the feasibility of using Ada to do high performance graphics in a distributed system environment. Several mismatches between the language and the application have been uncovered so far.

Bit Manipulation - Ada provides no underlying bit-level manipulation mechanisms (such as the logical operators and shifting or rotation of bits). Instead, bit masks are implemented as arrays of Booleans. Although this is conceptually a very nice model for dealing with what bit masks may represent, it introduces a very large amount of overhead. The bits must be combined by shifting and masking them one at a time. In the X server environment, which requires a large amount of bit manipulation to generate graphics in a frame buffer (or any other environment close to the physical world), the performance breaks down and presents a clearly unacceptable solution. Early recognition of this problem encouraged us to target a graphics processor rather than a frame buffer.

There is an alternative abstraction for bit masks using record types with Boolean, one-bit components for each place in the mask. This substantially reduces the overhead required, as the compiler can make certain optimizing assumptions while generating code. However, it still provides no capability for performing the masking and shifting operations on the object as a whole and fails to satisfy the requirements found in areas such as graphics generation.

This definition of bit masks is built into the language specification and, hence, is unlikely to change. Given the semantics of the specification, it is also unlikely that compiler code generation can be optimized sufficiently in this area. Recognizing this, some compiler implementations provide a library of services to perform this type of operation more efficiently (by interfacing to the underlying operating system or an alternate language implementation). Because of the prevalence of this type of operation in our application, we also will be examining alternate implementations of these operations, most likely in C or assembler language.

Client-Server Communication - The X server communicates with clients using any implementation of a reliable byte stream (in the reference server, TCP/IP is used). Each protocol element is mapped into an untyped packet of bytes for transmission to the server. When received by the server, this group of bytes must be mapped back into the correct data structure. Due to system timing and other external influences, only a piece of a protocol element may be available for reading into the server at any given time. Because Ada is such a strongly typed language, there is no real support for reading untyped, variable sized blocks of data. This has obvious implications for any kind of application involving "middle level" networking.

Compiler Technology - Ada is a very large and complex language, as evidenced by the Language Reference Manual[14]. Because it is also a relatively new language, Ada compilers are not yet mature and stable enough to support the entire language definition. This is especially true in those areas closest to the underlying physical

[14] *The Ada Programming Language Reference Manual*, ANSI/MIL-STD-1851A-1983.

environment as defined in Chapter 13. As compilers become more mature and complete, this will be less of an issue. In the meantime, it is important to keep in mind that problems can occur when using the newer, rarely used features of the language.

We have run into several problems with our Ada compiler in the areas of generics and correct implementations of representation specifications, both of which have necessitated alternate implementations. For this reason, we have endured the difficulties of acting as a beta test site for new compiler releases. This allows us access to the vendor's most current compiler technology and helps to minimize the impact of compiler bugs.

Another area that has caused problems for us is in compiler optimization. We are using a memory mapped interface to the graphics generator where data is sent to the board by writing to a single address which represents the head of a FIFO type queue. Compiler vendors, in general, do not provide much control over degrees and types of optimization performed. Hence, the compiler treats the series of statements writing to the FIFO as dead code and eliminates all but the last one. In order to prevent this problem, the code must be isolated and compiled unoptimized (which generates horrible code), resulting in rather clumsy interfaces to things.

Design Methodologies - There are several different design methodologies available for use with Ada, with a somewhat smaller number applicable for implementations requiring real-time performance or appropriate for the design of concurrent programs. One problem with several of the methodologies is that they produce an inordinately large number of Ada tasks. Although this may provide a conceptually clean design, with concurrency issues correctly handled, it presents a performance problem. The implementations of the Ada tasking model provided by most Ada runtime environments falls short of providing one suitable for use in real-time environments due to the overhead involved with task context switches. This is especially true when the Ada runtime is implemented on top of some other underlying operating system which does not have real-time characteristics, as opposed to a bare-machine environment.

Code Size - Although the original estimate of the amount of code to be developed was based on the C implementation (roughly 24,000 lines of code), it has become evident that our Ada implementation will be substantially larger (on the order of 45,000 lines of code). One reason for the larger size is our use of Ada's separate compilation capability (through the use of package specifications with separate subprogram bodies). In doing this, it is necessary to generate large amounts of Ada source code at the start of a project with little executable code being created in the process. Although this code does not really show up in the count of truly "executable" code, it takes time to develop and must be included in any schedule estimates.

Development Environments - As stated earlier, this project is being developed on a group of Sun-3 file and compute servers. We have discovered that the compiler and associated tools (linker and debugger) tend to be quite large. This can limit the number of simultaneous users a Sun file/compute server can support as it becomes unable to allocate enough virtual memory to support them. In addition, because so many files need to be read and written during compilations, we have found, during benchmarked library builds, that the CPU is idle for 50% of the time waiting for disk I/O. Both of these point to a need for careful consideration of host development environments and supporting hardware.

Foreign Language Interface - One unique feature of Ada is its ability to interface to other languages (i.e., C, FORTRAN, or assembler) in a language defined, compiler supported fashion. This is achieved through use of the Pragma Interface. Although this does present an approach to using existing code written in other languages, one has to be wary of thinking this feature of Ada is a panacea. One of the things we quickly discovered on this project is that, lacking the rigors enforced by Ada, other developments may be inconsistent enough as to prevent their use. This point should be considered by all established product bases planning a migration to Ada.

SUMMARY

Our project is well on the way to a projected conclusion by year's end. We currently have a majority of the dispatch routines and device-independent resource type managers completed and have started integration with our targeted graphics processor. Valuable experience has been gained in the use of Ada for near real-time graphics and display management in an environment that is close to the underlying hardware and run-time system. In addition, we are finding the areas to watch out for as Sanders migrates towards the use of Ada in applications with substantial graphics content. At the conclusion of the project, we will be able to offer a total Ada solution on programs that require the industry standard X Window System.

BIBLIOGRAPHY

Angebranndt, S., Drewry, R., Karlton, P., Newman, T., *Definition of the Porting Layer for the X v11 Sample Server*, MIT, March 1988.

Booch, Grady, *Software Engineering with Ada*, The Benjamin/Cummings Publishing Company, 1987.

Department of Defense, *Military Standard: Defense System Software Development*, MIL-STD-2167A, 29 February 1988.

MITRE, *Use of Ada for FAA's Advanced Automation System*, MTR-87W77, April 1987.

Nielsen, K., Shumate, K., *Designing Large Real-Time Systems with Ada*, McGraw-Hill, 1987.

Scheifler, R., Gettys, J., Newman, R., *X Window System: C Library and Protocol Reference*, Digital Press, 1988.

United States Department of Defense and American National Standards Institute, Inc., *The Ada Programming Language Reference Manual*, ANSI/MIL-STD-1851A-1983.