Mach and Matchmaker:
Kernel and Language Support for Object-Oriented Distributed Systems

Michael B. Jones
Richard F. Rashid

September 1986
Technical Report CMU-CS-87-150

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

Mach, a multiprocessor operating system kernel providing capability-based interprocess communication, and Matchmaker, a language for specifying and automating the generation of multi-lingual interprocess communication interfaces, are presented. Their usage together providing a heterogeneous, distributed, object-oriented programming environment is described. Performance and usage statistics are presented. Comparisons are made between the Mach/Matchmaker environment and other related systems. Possible future directions are examined.

Keywords

Object-Oriented Systems, Object-Oriented Languages, Object Capabilities, Multi-Processor Operating System, Interprocess Communication, Remote Procedure Call, Multi-Targeted Compiler, Interface Specification Language, Distributed Systems.

This paper also appears in the Proceedings of the 1st Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), September, 1986.

THIS RESEARCH WAS SPONSORED BY THE DEFENSE ADVANCED RESEARCH PROJECTS AGENCY, DEPARTMENT OF DEFENSE, ARPA ORDER 3597, MONITORED BY THE AIR FORCE AVIONICS LABORATORY UNDER CONTRACT F33615-81-K-1539.

THE VIEWS AND CONCLUSIONS CONTAINED IN THIS DOCUMENT ARE THOSE OF THE AUTHORS AND SHOULD NOT BE INTERPRETED AS REPRESENTING THE OFFICIAL POLICIES, EITHER EXPRESSED OR IMPLIED, OF THE DEFENSE ADVANCED RESEARCH PROJECTS AGENCY OR THE U.S. GOVERNMENT.

Table of Contents

Mach and Matchmaker:

Kernel and Language Support for Object-Oriented Distributed Systems

Michael B. Jones
Richard F. Rashid

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

Mach, a multiprocessor operating system kernel providing capability-based interprocess communication, and Matchmaker, a language for specifying and automating the generation of multi-lingual interprocess communication interfaces, are presented. Their usage together providing a heterogeneous, distributed, object-oriented programming environment is described. Performance and usage statistics are presented. Comparisons are made between the Mach/Matchmaker environment and other related systems. Possible future directions are examined.

Keywords

Object-Oriented Systems, Object-Oriented Languages, Object Capabilities, Multi-Processor Operating System, Interprocess Communication, Remote Procedure Call, Multi-Targeted Compiler, Interface Specification Language, Distributed Systems.

1. Introduction

Many different object-oriented systems have been built and proposed. Some are inapplicable to distributed environments. Design and implementation considerations have often been driven by abstract issues rather than the concrete requirements of a distributed system. Many systems assume that all code will be written in a single programming language, ignoring the diversity of existing applications and programming tools already available. Some do provide multiple languages, but allow inter-language and inter-machine interfaces to be coded by hand in an ad-hoc manner. Many systems ignore protection issues, particularly for distributed applications. Others fail to provide efficient mechanisms for passing objects, and invoking operations. Finally, while an object-oriented programming style is supported for "applications code," the operating system itself is often not extensible, or implemented using the same abstractions.

Mach [1] [2] [3] is a multiprocessor operating system currently under development at Carnegie Mellon University. Mach provides

- a uniform object reference mechanism,
- protected object capabilities,
- efficient cross-domain object communication, and
- efficient cross-domain object operation invocations.

Matchmaker [13] is an interface specification language and compiler used in the Mach environment, providing programming language support for distributed, object-

oriented programming. Specifically, it provides

- support for multiple, existing programming languages,
- language support for object references,
- language interfaces for object operations,
- language and machine independent operation interface specifications, and
- automated interface code generation from interface specifications.

Mach and Matchmaker do not provide a "pure" object-oriented environment; not every piece of data is an object, unlike systems such as ECL [6] and Smalltalk [11] [9]. The only true objects present in the Mach/Matchmaker world are ports (and other types defined as ports). Both "normal" data and object references are supported. Many other object-oriented systems have taken this approach. For example, Simula [22] provides both Algol types and Classes, the Lisp Flavors [14] system allows both normal Lisp values and flavors, and Hydra [28] objects contained both capabilities and data.

Neither Mach or Matchmaker explicitly provide for inheritance of operations between "related objects," as do many object-oriented systems such as the Lisp Flavors system. Nonetheless, some servers do provide for inheritance. For example, the *Remote Method Invocation* mechanism of the Flamingo [23] window manager provides for "method" operations to be inherited by derived objects. Hence, related objects can still share common operations (and even share pieces of their implementations when appropriate) within the Mach environment. In this respect, Mach took the same decisions as did other systems such as Demos [5], StarOS [12] and iMAX [10].

Mach is currently being used by a number of research projects at CMU, and is being run on machines ranging from personal workstations to multiprocessors. The Matchmaker language has been in use for several years as the normal means of defining and generating interprocess interfaces for code within the Accent[1] [17]network operating system environment, and is now being used for Mach.

In this paper, we will discuss both the Mach operating system and the Matchmaker language. A brief description of their features will be presented, including some examples of their use. These examples serve to illustrate the object oriented nature of the features available and demonstrate the flexibility of the resultant system.

2. Mach discussion

2.1. Mach: An extensible object-oriented kernel

Mach is a multiprocessor operating system kernel currently under development at Carnegie Mellon University. The design of Mach draws heavily on CMU's previous experience with the Accent network operating system. Important aspects of the Mach design are

- kernel objects are referenced via object capabilities,
- all system services are provided via a capability-based interprocess communication mechanism,
- all systems abstractions allow extensibility to multiprocessors and networks of uniprocessor or multiprocessor nodes,
- the underlying mechanism for communication provides support for protection as well as network transparency,
- access to virtual memory is simple with no arbitrary restrictions on allocation, deallocation and virtual copy operations and yet allows both copy-on-write and read-write sharing, and
- any support for parallelism allows for a wide range of tightly coupled and loosely coupled multiprocessors.

In addition to satisfying these goals, Mach provides compatibility with existing environments at CMU. It is binary compatible with Berkeley 4.3 bsd, and is source compatible with existing Accent code. This allows existing program development tools and programming environments to continue being used, while also providing appropriate support for distributed, object-oriented systems.

Mach was conceived in 1985 as an Accent-like operating system which would provide complete UNIX[2] [18] compatibility. Experience with Accent showed that a message and capability based network operating system, properly designed, can compete with more traditional operating system organizations. The advantages of this

[1] Accent is a trademark of Carnegie Mellon University

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

approach are system extensibility, protection and network transparency. It was also designed to better accommodate the general purpose shared-memory multiprocessors which appear to be succeeding traditional general purpose uniprocessor workstations and timesharing systems.

As of June 1986, Mach currently runs on most VAX[3] architecture machines (VAX 11/750, 11/780, 11/785, 8600, μVAX I, and μVAX II) and the IBM RT PC[4]. In addition, Mach runs on a four (11/780 or 11/785) processor VAX 11/784 with 8 MB of shared memory. Mach is currently in production use by CMU researchers including projects for multiprocessor speech recognition and building parallel production systems. The same binary kernel image runs on all VAX uniprocessors and multiprocessors. Ports are in progress to the Sun-3, and the Encore MultiMax multiprocessor, with other ports in the works.

2.2. Kernel abstractions and operations on objects

The Mach kernel supports five basic abstractions:

1. A *port* is a communication channel – logically a queue for messages protected by the kernel. Ports are used as protected capabilities for all objects within the Mach environment. Ports are the reference objects of the Mach design.

2. A *message* is a typed collection of data objects used in communication between threads. Messages may be of any size and may contain pointers and typed capabilities for ports. All message data other than ports is passed by value.

3. A *task* is an execution environment in which threads may run. It is the basic unit of resource allocation. A task includes a paged virtual address space and protected access to system resources (such as processors, port capabilities and virtual memory). The UNIX notion of a *process* is, in Mach, represented by a task with a single thread of control.

4. A *thread* is the basic unit of CPU utilization. It is roughly equivalent to an independent program counter operating within a task. All threads within a task share access to all task resources.

5. A *memory object* is a unit of virtual memory which may be mapped into the address space of a task.

Send and *Receive* are the only primitive operations. Operations on all objects other than messages are performed by sending messages to ports which are used to represent them.

A message is sent by a thread to a port in order to cause an operation to be performed on the object represented by that port. The receiving thread will determine the object represented by the port, and perform the proper operation, depending on the object type, the message type and the parameters contained in it. It may also send a reply message containing results from the operation performed. Thus, messages sent to ports may be correctly viewed as object operation invocations.

An example from the Mach kernel serves to illustrate the use of ports as object references, and messages as operation invocations. The act of creating a task or thread returns access rights to the port representing the created object. This port can be used to manipulate the created object. Messages representing such operations as *Suspend*, *Resume* and *Terminate* can be sent to the port, causing Mach to perform the requested operation upon the task or thread, and to send a reply message containing results from the operation.

Many different types of objects can share common operations. *Suspend* is meaningful for both tasks and threads, and potentially other types of objects as well. While both objects in the example were provided by the kernel, they could equally well have been provided by distinct servers.

2.3. Extending object primitives to a network environment

The majority of message communication occurs between entities running on the same machine. This "normal" case is illustrated in figure 2-1. Nonetheless, messages can be sent to processes on remote machines in the Mach environment.

---

[3] VAX is a trademark of Digital Equipment Corporation

[4] RT PC is a trademark of International Business Machines Corporation

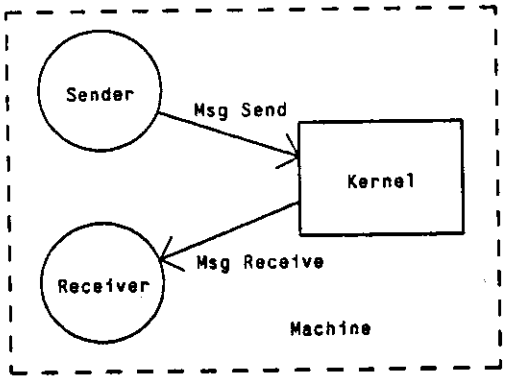

Figure 2-1: Local Message Communication

By the nature of the port and message abstractions, it is possible to insert transparent intermediary process between a pair of communicating processes which forward messages between the processes. Yet this level of indirection is undetectable by either party, since the processes with access to a port cannot be determined from the port.

Transparent network interprocess communication with preservation of capability protection across network boundaries is provided by a "message server" [19] running on each machine. Message servers extend the local port name space into a network global port space by representing each relevant remote port with a local port held by the local message server.

Likewise, the message servers provide the ability to transparently send messages to remote ports. For each remote port known on the local machine, the local message server actually holds a local port which is used locally as a surrogate for the remote port.

When a message server receives a message on a port representing a remote port, the message is encapsulated into a network message representation, and transmitted to the remote message server. The remote server then reconstitutes the message received over the net, and sends it to its intended destination on behalf of the original machine. The "remote" case is illustrated in figure 2-2.

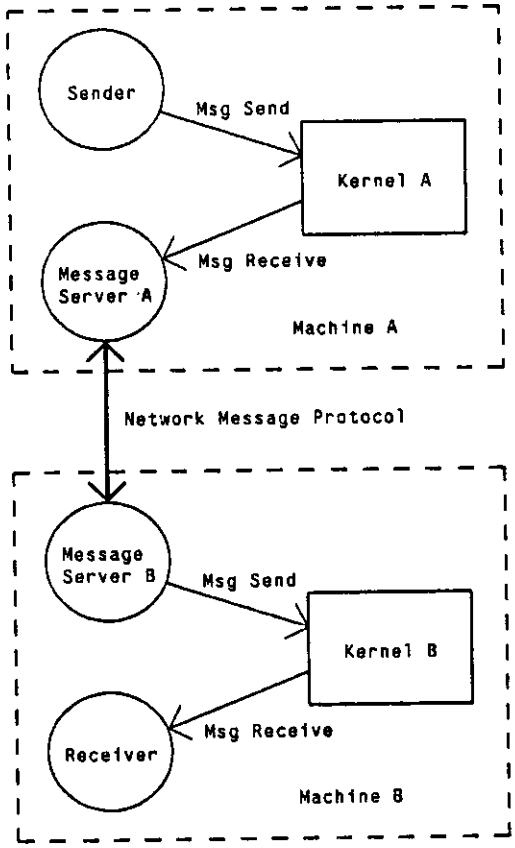

Figure 2-2: Transparent Network Message Communication

Port deallocation, and all port capabilities (*Send, Receive,* and *Ownership*) are correctly handled by the message servers. Data will be encrypted over the net when necessary [20]. Thus, the full Mach object semantics are transparently preserved across the network.

3. Matchmaker discussion

3.1. Matchmaker: Language support for distributed object interfaces

A variety of languages for distributed programming have been developed, such as PLITS [7] and Argus [15]. Likewise, a number of object-oriented languages such as Smalltalk [11] and the Lisp Flavors [14] system have come into use. Rather than being another new "distributed" or "object oriented" programming language, Matchmaker is an interface specification language for use with existing

programming languages. Matchmaker is a product of experience gained building the Accent environment during the CMU SPICE [25] project. It provides

- a language for specifying object-oriented remote procedure call (RPC) and asynchronous interfaces between communication processes, and
- a multi-targeted compiler which translates these specifications into interface code for each of the major languages used within the Mach environment, including C, COMMON LISP [26] and PERQ Pascal[5] [4].

Matchmaker allows interfaces between cooperating computing entities to be specified and maintained independent of specific languages or machine architectures. The Matchmaker code provides communication, runtime support for type-checking, type conversions, synchronization and exception handling.

3.2. Matchmaker background

Matchmaker was started in 1981 as part of the SPICE project. It was built to automate some of the coding for the Accent operating system kernel [17] message interface. It has since evolved significantly in its syntax, data representation semantics and communication semantics. At each point of change, decisions about Matchmaker design and implementation were driven by specific requirements of the SPICE and Mach environments.

Over the years, Matchmaker has proven to be a valuable tool. It has

- eased implementation and improved the reliability of distributed programs by detaching the programmer from concerns about message data formats, operating system peculiarities and specific synchronization details,
- improved cooperation between system programmers working in different languages,
- enhanced system standardization by providing a uniform message level interface between processes,
- provided a language rich enough to express any data structure which can both be efficiently represented in messages, and reasonably represented in all target languages, and
- reduced the cost of reprogramming interfaces in multiple languages whenever a program interface is changed.

Today, Matchmaker interfaces define the vast majority of interprocess communication in the SPICE and Mach environments, including the kernel interfaces. To date, Matchmaker has been used as the distributed programming support environment for over 500,000 lines of code written in four major languages. Matchmaker has evolved from a simple programming aid into the effective definition of interprocess communication within the SPICE and Mach environments.

3.3. Matchmaker computational model

Matchmaker builds upon the facilities provided by Mach. Objects are still represented by ports, and object operations are still invoked via messages. With Matchmaker though, the programmer no longer *programs* using messages. Matchmaker hides the underlying message passing mechanisms.

The Matchmaker language is used to specify procedural interfaces for sets of operations upon objects. Instead of requiring programmers to invoke object operations via message sends and receives involving parameter manipulations only slightly less intricate than assembly language, Matchmaker allows programmers to invoke (and receive invocations of) operations via corresponding procedure calls.

Each declared operation contains an object port parameter, and a list of *in* and *out* operation parameters. The object port parameter is used to pass the object on which the operation will be performed, and the operation parameters are used to pass data specific to each operation invocation. Parameters may themselves be other object references. All parameters other than object references are passed by value. Thus, the Matchmaker language also provides object-oriented interfaces to services, but at a higher level than bare Mach does.

3.4. Matchmaker features

The syntax of Matchmaker specifications is fairly close to the Pascal or Ada syntax for the analogous objects. Constants of various types can be declared, new data types can be constructed from built-in types (within certain

[5] PERQ is a trademark of Perq Systems Corporation

constraints), and remote procedures can be declared with a syntax fairly similar to Pascal procedures or functions. The invocation of an operation on a port in a given target language is usually represented as a procedure call in that language, with the port as the first procedure parameter.

The built-in data types provided by Matchmaker are: **Boolean**, **Character**, **Signed** and **Unsigned Integers** of various bit sizes, **Integer SubRanges**, **Strings**, **Communication Ports**, and **Reals**. New data types can also be constructed with some restrictions. Type constructor functions supported are: **Records**, fixed and variable-sized **Arrays**, **Enumerations**, **Pointers** to the above types, and certain kinds of **Unions**.

Several semantically different kinds of remote procedure call interactions can be specified in Matchmaker. The process normally initiating an operation is called the *client process*, and the process normally receiving requests is called the *server process*. The RPC paradigms provided are:

- **Remote_Procedure:** Generates code for a client process to send a request to a server, and to receive reply parameters back from the server. Timeout values can be specified, and the wait for the reply can be made asynchronous as well.

- **Message:** Generates code for a client process to send a single request message to a server without a reply.

- **Server_Message:** Generates code for a server process to send a single message to a client process.

- **Alternate_Reply:** Generates code for a server process to send a reply message back to a client process in response to a **Remote_Procedure** which is different than the normal reply message. **Alternate_Reply** messages are meant to be used for signaling exception conditions which occurred during execution.

Each of these varieties of calls except for Alternate_Reply takes a port as a parameter to which the request is sent. Thus, "binding" is done dynamically on the basis of ports, and not by using some compile-time or link-time discipline. Or restated, each operation takes the object on which it is to be performed as an explicit parameter.

4. Handling heterogeneity

There are many sources of heterogeneity which have had to be confronted to make the Mach environment work. Some specific sources are

- networks
- processor architectures
- multiprocessor architectures
- operating systems
- programming languages, and
- display devices and user interfaces.

4.1. Programming language diversity

The Mach environment currently supports four programming languages – each with its own notions of procedural invocation, data representation and exception handling: COMMON LISP, C, Ada and Pascal. Both clients and servers can be built in any of these languages.

The Matchmaker language is used to mask language differences by compiling object-oriented interprocess communication interface specifications into client and server remote procedure call code implementing those interfaces for each target language. Matchmaker handles differences in language syntax, type representations, record field layout, procedure call semantics, and exception handling semantics.

Certain semantic restrictions are placed upon the Matchmaker data types to allow efficient passing of arguments in messages. In particular, pointers, variable-sized arrays and unions can only occur in top-level remote procedure call declarations, and may not be used when constructing other types.

4.2. Data representation issues

In the Mach environment, data representation issues take several forms:

- byte ordering,

- hardware and languages differences in the packing of simple data types into aggregates, and

- actual differences in the representation of simple data types.

Differences in type representations by various programming languages within each machine are handled by Matchmaker. Data representation issues across machine

boundaries are handled through a different mechanism.

All inter-machine communication within the Mach network is handled through message server [19] processes which transparently forward messages between processors. Messages consist of a standard format header with a variable size body. All data passed in a message is typed. Pointers may be passed, but all data is transferred by value and only one level of indirection is permitted. Byte reordering and machine specific conversions are performed by the message servers with the responsibility for conversion always resting with the receiving host.

Using message servers for all network communication has a second advantage. Since code other than the message servers themselves use message passing for remote communication rather than network interfaces, all differences between various networks are hidden from programs in the Mach environment.

4.3. Operating system diversity

In most cases, Mach processes depend only on the Mach interprocess communication facility (IPC) to communicate with each other and perform their tasks as servers. The Mach kernel specifically provides this facility as the basis for building distributed, object-oriented applications. Nevertheless, it is possible to implement the Mach IPC facility as a service under other operating systems.

To date, we have taken advantage of this capability largely as a way of integrating our Accent and Mach based environment with various versions of VAX UNIX. Early in 1980 support was provided for an Accent/Mach-style IPC facility within VAX UNIX (32V). Currently message passing is supported under Berkeley 4.1BSD and 4.2BSD. Remote file access, name-lookup, remote process invocation and process control operate between Accent, Mach and UNIX hosts allowing substantial interoperability.

4.4. Multiprocessor differences

The Mach task, thread and virtual memory mechanisms were designed to allow for efficient implementation on a wide class of tightly and loosely coupled shared-memory multiprocessors. The thread mechanism should provide a high processor utilization on multiprocessors by increasing the number of schedulable computing entities available to the kernel.

Another problem addressed by Mach is the differences inherent between uniprocessor and multiprocessor architectures. Mach was designed such that all systems abstractions can be implemented on uniprocessors, as well as allow extensibility to multiprocessors and networks of uniprocessor or multiprocessor nodes. The success of this approach is illustrated by the fact that all VAX family processors, from μVAXes to 8600s and a four processor multiprocessor 780, run exactly the same kernel binary, providing the same features.

4.5. Handling diverse user interfaces

One of the problems we have encountered has been a proliferation of user interfaces both on Mach and UNIX workstations. We are currently building a flexible, object-oriented user interface facility called Flamingo [23] which will allow many different windowing and management paradigms to be in use on the same workstation's display, potentially at the same time. Currently both the Accent Sapphire [21] interface and ITC's (CMU Information Technology Center) Andrew [16] interfaces are being implemented using Flamingo.

5. The Sapphire window manager: Using the advantages of object capabilities

One of the often cited advantages of oriented systems is the ability to hide the implementation of an object from the object clients. This allows the implementation of operations on objects to be changed, either at compiletime, or even at runtime, without effecting the client system. The Sapphire [21] window manager developed under Accent, and emulated by Flamingo under Mach, utilizes these advantages in its implementation.

In some respects Sapphire is a fairly traditional window manager in the abstractions which it provides; *Rectangles* are rectangular arrays of pixel memory, *Viewports* are rectangular pixel arrays which may potentially be covered by other viewports, and *Windows* are viewports with borders and menus. Each Sapphire abstraction, however, is represented by an object capability, similarly to the Smalltalk window system [11]. A Matchmaker interface is provided for each Sapphire abstraction, which makes the object operations available to clients via procedure calls (and calls the server operation implementations in a similar fashion).

The viewport implementation takes advantage of the ability to dynamically change the implementation of an object in a capability based system. Ordinarily, viewports are implemented by Pascal code which processes requests received on the ports representing viewports. This code remembers the ordering of viewports, determines clipping boundaries, and performs microcoded raster operations for each visible rectangular subregion of the viewport which needs to be changed.

An important special case, however, is handled through a completely different mechanism. When Sapphire determines that a viewport is completely uncovered – that no clipping within the viewport needs to be done, then the right to receive operations on the viewport object is passed from Sapphire to kernel driver code directly implementing rectangle raster operations. The driver code operates many times more efficiently than the corresponding Sapphire code since it has far less to do. The application program is unaware that the handler for the viewport capability has been changed (unless, of course, timings are being done).

The Sapphire viewport example clearly shows the advantages of being able to change the handler for operations on an object. The ability to dynamically provide a service via the "best" server available at that time allowed for an end-to-end approach to be taken, instead of relying on a strictly layered implementation. Thus, an order of magnitude efficiency gain was possible in an important special case.

6. Status and future directions

6.1. Status

As of June, 1986, the only major feature of Mach not yet fully implemented was the Mach thread facility. It is expected that this will be completed by summer of 1986.

Matchmaker has been the primary definition of inter-domain interfaces for SPICE since 1982. The current Matchmaker implementation has been running since early 1985. It is used to specify and build all SPICE and Mach interfaces on the PERQs, VAXes, and IBM RT PCs in C, PERQ Pascal, COMMON LISP, and Ada. Work is currently in progress allowing Matchmaker to better handle differences in data structure layouts on different machines.

6.2. Future directions

Once the Mach thread implementation is completed many of the system servers can be changed to take advantage of it. One potential area of research is studying how much parallelism is gained by having threads available. Another area of upcoming work is the development and tuning of pager and memory object manager processes which use an object-based kernel message interface, rather than being imbedded in the kernel.

Currently, some small Matchmaker changes are in progress which allow Matchmaker to handle some problems of byte ordering within records which are not currently handled by the message servers. These changes should be complete by Fall of 1986.

In the longer term, research is underway to attempt to generalize the existing Matchmaker work in several ways. While Matchmaker is already a useful and fairly ambitious tool, the current implementation suffers from the following limitations and deficiencies:

- Knowledge about target programming languages is scattered throughout code generator implementations, rather than declared.
- Knowledge about target machines is scattered throughout various pieces of code, rather than declared.
- Knowledge about message formats is scattered throughout the Matchmaker front end, rather than declared.
- Knowledge that Accent/Mach messages are the underlying communication mechanism is pervasive throughout Matchmaker.
- Little flexibility is provided in type conversion choices for target languages.
- All representation choices are made at compile-time; one might want to be able to pass data via more than one representation.

Many of these restrictions should be able to be removed if the new tool is properly structured.

Possibly the most ambitious, but the most interesting of the above issues is abstracting the object-oriented Matchmaker specifications (and generated code) from message passing as a transport mechanism. As we have already seen, Matchmaker interface routines supplant

messages as the effective means of invoking object operations. Thus for many interfaces it should be possible to replace message passing as a transport medium with a different transport medium, while preserving the same interfaces. Even if this is not reasonable, it should still be possible to build a tool capable of producing interface code for a diversity of environments, from Mach to the Xerox Courier data interchange standard [29], for example.

7. Conclusions

Mach and Matchmaker represent an object-oriented programming environment real enough to serve as the basis for a large body of research software development, including the Sesame [27] distributed file system, the Flamingo [23] window system, the TABS [24] distributed transaction facility, as well as supporting existing UNIX applications. By combining the notion of object references and invocation with communication ports and message passing, Mach allows for simple transparent extensions of the object paradigm to a distributed environment. Matchmaker serves to bridge the barriers between heterogeneous programming environments and the Mach object facilities.

The granularity of Mach/Matchmaker object operations is considerably coarser than "pure" object language systems such as Smalltalk. Operations typically require on the order of 2-5 milliseconds [13]. This has proven adequate, however, for a range of systems tasks including transaction processing, window management, and even speech recognition. Overall performance of Accent, for example, is in the range of comparable traditional operating systems such as UNIX [8].

By extending the object paradigm to the lowest levels of the system and system services we have gained a uniform, protected capability name space, efficient interprocess message communication, an extensible, capability based kernel, integrated interprocess communication and virtual memory management, and transparent, protected extensions of the system primitives to a network environment. Matchmaker, likewise, not only has made using the Mach object paradigm easier to use, but has done it so well that it has become *the* working definition of object operation interfaces in the Mach environment.

Through use in real object-oriented distributed systems, Mach and Matchmaker have proven their worth as kernel and programming language support for building object-oriented applications in a very demanding distributed environment.

I. Example Matchmaker interface specification

The text which follows is a fictional Matchmaker interface specification for a "display server" process.

```
Interface Screen = 15000;        ! Base Msg ID is 15000

Constant
    Max_X       = 132;
    Max_Y       = 40;

    Inverted    = true;
    Normal      = not Inverted; ! A constant expression

Type
    Screen_Array = packed array [Max_X * Max_Y] of Character;

    Char_Vector  = ↑ packed array [*] of Character;

    Screen_State = record
                       x          : byte;
                       y          : byte;
                       Reverse    : boolean;
                   end record;

    Screen       = port;    ! Port object ref for each screen

Message DisplayChars(
                           : Screen;          ! Object parameter
        x                  : byte;
        y                  : byte;
        chars [num]        : Char_Vector;  ! Note size parameter
        ) : No_Value;

Message PutChar( : Screen; c : Character) : No_Value;

Message ClearScreen( : Screen) : No_Value;

Remote_Procedure GetWholeScreen(
                           : Screen;          ! Object parameter
   out  ScreenArray        : Screen_Array;
   out  Current_X_Size     : byte;
   out  Current_Y_Size     : byte;
        ) : GR_Value;

Remote_Procedure SwapScreenState(
                           : Screen;          ! Object parameter
  inout State              : Screen_State;
        ) : No_Value;

Alternate_Reply No_Such_Screen;

End Interface
```

II. Example client code for the example specification

The following text is a fragment of a C program using the example interface. It would be coded by the programmer using the service, but it calls client interface code generated by Matchmaker.

```
#include <Screen.h>     /* Get interface type definitions */

static Screen myscreen; /* Holds screen object reference */

clear_hello()
/* Clear myscreen and say hello */
{
    register int i;
```

```
    static hello = "Hello!\n";

    ClearScreen(myscreen);          /* Call interface routine */
    for (i = 0; hello[i]; i++) {}/* Get length of C string */
    DisplayChars(myscreen,          /* Pass object reference */
        30, 12, hello, i);          /* and parameters */
}
```

III. Example server code for the example specification

The following text is a fragment of a PERQ Pascal program implementing the services described in the example interface. It would be coded by the programmer implementing the service, but would be called by the Matchmaker generated interface code.

```
imports ScreenDefs from ScreenDefs;     { Type definitions }

procedure DisplayChars(
                ScreenPort : Screen;
                x, y       : byte;
                chars      : Char_Vector;
                num        : long);
  var
    i : integer;
    scr : screen_ptr;
  begin
    { Resolve object reference to data structure ptr }
    scr := port_to_screen_ptr(ScreenPort);
    if scr = NIL then exit(DisplayChars);

    position(scr, x, y);            { Position Cursor }
    for i := 0 to num-1 do
        display(scr, chars^[i]);    { Access char vector }
  end;
```

Acknowledgments

Mach was the brainchild of many including Avie Tevanian, Mike Young, Bob Baron, and Rick Rashid. David Golub and Bill Bolosky are also working on the implementation. Matchmaker was the product of many good people, among them Doug Philips, Mary Thompson, Rob MacLachlin, Keith Wright and Mike Young, as well as the authors. The network message server was originally written by Rick Rashid, and is being rewritten for Mach by Dan Julin, Robert Sansom and Ed Zayas.

This research was sponsored by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-81-K-1539.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

References

1. M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. Proc. Summer 1986 USENIX Technical Conference and Exhibition, June, 1986.

2. Robert Baron, Richard Rashid, Ellen Siegel, Avadis Tevanian and Michael Young. MACH-1: An Operating System Environment for Large-Scale Multiprocessor Applications. IEEE Software, IEEE, July, 1985.

3. Robert Baron, Richard Rashid, Ellen Siegel, Avadis Tevanian and Michael Young. MACH-1: A Multiprocessor Oriented Operating System and Environment. In *New Computing Environments: Parallel, Vector and Systolic*, Arthur Wouk, Ed., Siam, Philadelphia, PA, 1986. Also available as at CMU CS technical report Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, April, 1985.

4. Miles Bartel, Michael Kristofic. PERQ Pascal Extensions. In *PERQ Software Reference Manual*, Three Rivers Computer Corporation, 1982.

5. Forest Baskett, John H. Howard, John T. Montague. Task Communication in DEMOS. Proc. 6th. Symposium of Operating System Principles, ACM, November, 1977, pp. 16-18.

6. *ECL Programmers Manual.* Cambridge, MA, 1974.

7. Jerome A. Feldman. "High Level Programming for Distributed Computing". *Comm. of the ACM 22*, 6 (June 1979), 353-368.

8. R. P. Fitzgerald and R. F. Rashid. "The Integration of Virtual Memory Management and Interprocess Communication in Accent". *ACM Transactions on Computer Systems 4*, 2 (May 1986).

9. A. Goldberg, D. Robson. *Smalltalk-80.* Addison-Wesley, Reading, MA, 1983.

10. Kahn, K.C. et al. iMAX: A Multiprocessor Operating System for an Object-Based Computer. Proc. 8th Symposium on Operating Systems Principles, ACM, December, 1981, pp. 127-136.

11. Daniel. H. H. Ingalls. The Smalltalk-76 Programming System Design and Implementation. Xerox Palo Alto Research Center, Palo Alto, CA, 1980.

12. A. K. Jones, R. J Chansler, I. E. Durham, K. Schwans, and S. Vegdahl. StarOS, a Multiprocessor Operating System for the Support of Task Forces. Proc. 7th. Symposium of Operating System Principles, ACM, December, 1979, pp. 117-129.

13. Michael B. Jones, Richard F. Rashid, Mary R. Thompson. Matchmaker: An Interface Specification Language for Distributed Processing. Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ACM, January, 1985. Also available as Technical report CMU-CS-84-161, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1984.

14. Sonya E. Keene, David A. Moon. Flavors: Object-oriented Programming on Symbolics Computers. Common Lisp Conference, December, 1985.

15. Barbara Liskov. Overview of the Argus Language and System. Programming Methodology Group Memo 40, MIT Lab. for Computer Science, February, 1984.

16. J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, F. D. Smith. "Andrew: A Distributed Personal Computing Environment". *Communications of the ACM 29*, 3 (March 1986), 184-201.

17. Rashid, R. F. and Robertson, G. Accent: A Communication Oriented Network Operating System Kernel. Proceedings of the 8th Symposium on Operating Systems Principles, December, 1981, pp. 64-75.

18. D. Ritchie. "The Unix Time-Sharing System". *CACM 17*, 7 (July 1974), 365-375.

19. Robert D. Sansom, Daniel P. Julin and Richard F. Rashid. Extending a Capability Based System into a Network Environment. CMU Computer Science Department, April, 1986.

20. Robert D. Sansom. Security in a Network Operating System. Securicom 86 - 4th Worldwide Congress on Computer and Communications Security and Protection, March, 1986.

21. *User's Guide to the Sapphire Window Manager.* PERQ Systems Corporation, 1984.

22. Dahl, O.-J. and K. Nygaard. "Simula – An Algol-Based Simulation Language". *Communications of the ACM 9*, 9 (Sempember 1966).

23. E. T. Smith and D. B. Anderson. Flamingo: Object-Oriented Abstractions for User Interface Management. Proceedings of the Winter 1986 USENIX Conference, January, 1986, pp. 72-78.

24. Alfred Z. Spector, Jacob Butcher, Dean S. Daniels, Daniel J. Duchamp, Jeffrey L. Eppinger, Charles E. Fineman, Abdelsalam Heddaya, Peter M. Schwarz. Support for Distributed Transactions in the TABS Prototype. Proceedings of the 4th Symposium on Reliability In Distributed Software and Database Systems, October, 1984. Also available as Carnegie-Mellon Report CMU-CS-84-132, July 1984..

25. CMU Computer Science Department. Proposal for a Joint Effort in Personal Scientific Computing. Carnegie-Mellon University, August, 1979.

26. Guy L. Steele Jr.. *COMMON LISP: The Language.* Digital Press, 1984.

27. Mary R. Thompson, Robert D Sansom, Michael B. Jones, Richard F. Rashid. Sesame: The Spice File System. CMU-CS-85-172, Carnegie-Mellon University, December, 1985.

28. William A. Wulf, Roy Levin, Samuel P. Harbison. *HYDRA/C.mmp: An Experimental Computer System.* McGraw-Hill Advanced Computer Science Series, 1981.

29. *Courier: the remote procedure call protocol.* Xerox Systems Integration Standard 038112, Xerox Corporation, Stamford, Connecticut, 1981.