Workshop: *Object-Orientation in Operating Systems*

Organizers:
Vince Russo (Purdue University)
Marc Shapiro (INRIA)

1 Introduction

Due to the unexpectedly large number of participants, it became necessary to organize this workshop a bit more than what the participants (and organizers) might have preferred. Since there were approximately 50 attendees, the chosen format was 4 sessions representing general themes, with 4 attendees presenting their position for each session. The first presentation of a session was longer than the rest and was intended to set the tone for the rest of the session. Each session is summarized below by a brief synopsis of the presentations from the session chairman. Each session ended with a question/answer period. The minutes of these periods are included with each session. The final session of the workshop was a panel discussion attempting to answer the question: "what is an object-oriented operating system?"

2 First Session: Operating System Issues

Michael Jones, chair

2.1 Operating System Support for Objects in a Distributed Environment,

Vinny Cahill (Trinity College, Dublin)

The Oisin system is implemented as part of the Esprit-I Comandos project. Oisin is a native distributed system operating system kernel supporting objects. It provides transparent access to both local and remote objects which may be either recently created or long-lived persistent objects. Such persistent objects are stored in the distributed storage system.

Most of this presentation dealt with the implementation of the Oisin object model. In particular, the Cluster abstraction. Clusters are groups of related objects collected together to help granularity. A Cluster is the unit of virtual memory transparent to the programmer. Clusters can be mapped into multiple address spaces. All objects are referenced as offsets from the beginning of the cluster. A special table distinct in each address space contains addresses of the code objects. Each cluster knows the externally visible (mature) objects. An immature object is promoted to a mature object when it is assigned to a global reference.

Oisin supports two invocation schemes: inter- and intra-Cluster. Inter-Cluster invocations are about 10 times the cost of intra-Cluster.

2.2 Object Oriented Platforms

Jose Alves Marques (INESC)

This presentation related more experiences with the Comandos system's object model. In particular it discussed the programming model further. Their goal is to use existing languages and compilers. They extend the languages by adding system classes for concurrency and naming. The model is basically that of C++ with restrictions. In particular, the restrictions prohibit pointers to instance data. The programmer is also aware of the difference between language and system objects. An "invoke" primitive is used on system objects and is supported by the runtime system.

This system supports:

- Dynamic object loading
- Dynamic loading of classes
- Cross-context invocations

The sharing model is that of shared memory tied to the clustering model discussed in the previous talk.

2.3 Operating System Support for Omniscient Object-Oriented Databases,

Peter Schwartz (IBM Almaden Research Center)

This talk focused on problems of database management and the correlation of unorganized information by providing an object-oriented access to any information. Their goals are to make no assumptions about direct links between data, and to provide a language independent data model.

Their object model is planned to use a variant of the Emerald conformance model. They will use access lists for access control.

2.4 The MUSE Object Architecture

Yasuhiko Yukote (SONY, Tokyo)

The MUSE system addresses granularity of objects, and the differences between language and system objects. It attempts to provide a self-advancing system based on a reflective computation model. The MUSE system uses a Meta-Object model to support the reflective computation model.

2.5 Discussion

Q(to Yukote): What is the difference between objects and metaobjects?

A: Metaobjects support virtual machines. Mailer/Memory/Scheduler are all metaobjects. Metaobjects provide an environment for their sub-objects. A metaobject is sort of like a proxy.

Q(to Cahill): Are clusters a win? (without programmer visibility/tweaking)?

A: No—the default mechanism makes one huge cluster and is bad. It would probably be best to have the programmer/language libraries do it. Could use automatic reclustering mechanism (under study).

Q(to Cahill): I have two questions—I like the segmented memory model, but isn't it hard to implement on flat VM hardware?

A: Implementation tries to map computation model down to hardware. Constrains things too much. Hardware support (i.e. 64-bit pointers) would be a win.

Q: Each physical page can be in multiple VM spaces. What about distributed virtual memory?

A: You can do "diffusion" with special objects to handle communication.

Q: There is a war between single virtual address across the Universe camp and the "that's impossible" camp. Where do you stand?

A: The model for each applications programmer is a single global objects space—this must be mapped down to conventional hardware. There should be no great performance penalty for doing this, if it's implemented efficiently.

Q: I think the concern about language- versus OS-objects is unnecessary. (1) Fortran doesn't even have language level objects. (2) Aren't OS objects just instances of subclasses?

A: We don't want to force the applications programmer to choose which objects are visible to the OS. Sometimes the programmer has to choose—especially for multiple language support. We could put this into a library (proxies) and hide it from the programmer. We don't *need* application programmer – OS interaction.

Q: Memory-management for short-lived objects has been done. Where does the OS come into play for memory management/protection?

A: Security is a separate issue. Once you gain access to a context, you get to do anything. Separate address spaces should be used for protection in cross-context invocation. Dynamic linking within an address space is not shared. Code objects are not shared. You can take advantage of the language's typing mechanism to some degree, but yes, it is a problem.

Q(to Schwarz): Can users have an Open Systems model?

A: Users can defines types and groupings. The query language matters.

Q(to all 4 panelists): How open can an object-oriented system be?

A: [laughter]

Q: Does metaness address openness?

A: Yes. Metaobjects can inspect sub-objects directly

A(Schwarz): Since a query is over a set of entities you must limit the set of queriable objects (like relational databases do), so it can't be totally open.

A(Marques): Efficiency is compromised by openness. SOS has a well-defined interface (moving objects), but objects can attach their own semantics.

Q: As a strawman, I would assert that sh/awk/sed/grep is a kind of brain-damaged query language already in use today—how are you better?

A(Schwarz): You can get a lot more out of it. Objects have methods—having only structure to grep through is not enough.

Q: Encapsulation is broken by VM mapping. Object invocation might protect a bit; but what about using VM when the language doesn't provide protection?

A: Hardware can protect me (RO access, for example) even from stuff that's mapped into my address space. Multiple (hardware) levels of protection would be nice.

Q: You ... an "object kernel" (runtime or teal OS). What VM and IPC support do you need?

A(Cahill): Not much—just need user control of VM mappings. There is a micro-kernel for this, and we build our kernel out of it. We need TCP/IP, segmented memory and can build the rest.

A(Marques): An interface to the fault-handler is handy too. There are both policy and mechanism issues.

3 Second Session: Objects and Distribution

Jose Marques, chair

3.1 Fragmented Objects: A Building Block for Object-Oriented Systems,

Yvon Gourhant (INRIA)

The main objective of this work is the development of a tool that within an object oriented framework provides efficient support for the programming of distributed services.

The author claims that most of the work on distributed object oriented systems has been based on the analysis of single objects accessible from different address spaces and do not consider the most general cases where a service is implemented by a set of cooperative server objects, by a server exporting proxies to clients, or by several objects encapsulating the manipulation of a protected resource.

The universe of Fragmented Objects (FO) is a set of elementary objects instantiated in different address spaces cooperating closely with one another, without interference from other objects. From the outside world, these objects are seen as a single one, the location of the elementary objects constituting the fragmented object is transparent to users. However, transparency is only considered on the user side, the implementor of the FO is aware and may exploit the knowledge associated with distribution.

An FO has a distributed public interface, its type, a distributed implementation containing code and data, plus a protected internal interface defining the means for a safe internal cooperation.

To provide support for the FO a specific language was developed together with communication facilities to ensure safe interaction. To program an FO one has to define its type which is implemented by a fragmented class. Fragmented methods are the class methods which provide the same behavior but are implemented in different fragmented classes and executed at different fragmented instances. A fragmented method has at least one server component and a client component. Syntactically the language is an extension of C++.

Server and proxy methods communicate through typed channels. A channel represents a virtual connection between a proxy and one or several servers. The channel supports group communications and both synchronous and asynchronous invocation.

The main strength claimed by the FO implementors is to provide the appropriate levels of visibility, hiding implementation details from users but not from implementors. The FO concept was used in SOS to program a distributed object manager, communication protocols, a distributed name server and some distributed applications.

3.2 Scalable Support for Autonomous Networks,

Peter Dickman (University of Cambridge)

The objective of the project is to investigate and demonstrate scalable mechanisms to build applications for complex large-scale computer supported cooperative work (CSCW). Motivations for the project are:

- management autonomy
- cooperative working
- wide availability of communication facilities.

The main problem in considering collaborative work has to do with the difficulty of establishing inter-administration facilities that do not conflict with internal autonomy and allow external users to access resources held by a particular administration domain.

Object-based techniques are favored as an approach to manage the complexity inherent in such CSCW applications. The project will analyze the appropriate set of underlying mechanisms needed to support the methodology. A part-emulated part-simulated system is under construction. CCLU (concurrent CLU) is used as implementation language.

The system is based on three abstractions: objects, references to objects, and gateways.

The system defines a very simple object model upon which several object-oriented models are supported by embedded objects. Language heterogeneity is a hard issue out of the scope of the system mechanisms. However the language implementation is simplified if all languages are build on top of the same object location and message passing mechanism.

Gateways are used to separate organizations and may be processors, data structures, or logical concepts.

References which supposedly indicate an object in a different organization point to a proxy in the gateway.

Migration of objects involves leaving proxies behind and eliminating unnecessary links.

Component networks only know the part of the system they interact with, eliminating the need of very general name-server.

Current garbage collection techniques are inadequate for the scale of the system considered, therefore a hybrid garbage detector-based on three traditional approaches is being developed.

Support for specific load balancing is also being investigated.

3.3 Distributed Invocation in Multimedia Object-Oriented Systems,

Nigel Davis (Lancaster University)

The objective of the project is to provide support for multimedia applications in an heterogeneous environment.

The work is based on the conceptual model provided by the Ansa project. In the Ansa model objects make themselves available by exporting an abstract data type signature to a trader. The user imports the object and gets a type template. Importing returns a ticket (capability) but does not bind the object to a particular location. Binding is separated from object importation and it translates to locate the object in the storage hierarchy and optionally moving it to a more appropriate location for invocation.

Different techniques concerning local and distributed invocations and the introduction of persistence on the invocation mechanisms were discussed.

In the Lancaster architecture an invocation manager is currently being explored as a major element in the system architecture. The invocation manager has a number of tools at his disposal: activation/passivation of objects, procedure calls, remote procedure calls, and migration of services. The main task of the invocation manager is to select the most appropriate techniques for a given series of invocations.

The invocation manger intervention is at binding time making objects fixed for a series of invocations. With this approach object invocation carries no extra overhead and would map either on local procedure calls or remote invocations.

3.4 Use of Object-Oriented Technology in the Implementation of a Distributed Operating System,

Paulo Guedes (OSF)

The requirements for an object oriented programming environment to support the development of the system services on top of the Mach-3 micro-kernel were discussed in this presentation.

The architecture of the system, under development in CMU and OSF, has three main components: the "pure" micro-kernel, a set of generic system servers and the emulation libraries.

The system servers execute as user processes and provide the functionality of the operating system which is not directly related to the abstractions implemented by the micro-kernel (e.g. files, protocol management).

The emulation library executes in the user address space and provides the OS. interface. The emulation library and the servers communicate through the Mach/Ipc.

This layer is currently written in an object-oriented environment called MachObjects providing typed objects, delegation, and a generic transparent RPC.

The basic objective of the work reported was the definition of the requirements for a new technology for building the servers interface. Some of the requirements are:

- to provide backward compatibility with existing languages,
- efficiency,
- distribution transparency in the access to remote objects,
- dynamic linking

Distribution was discussed in more detail. Transparency in the access to remote objects is considered to be an important feature in a distributed system. Remote invocation have been usually implemented by using stub routines. A generalization in an Object-Oriented framework is to consider stubs class (proxies) as a representative of the remote one providing the same interface. Optimizations are possible by transferring to the client side part of the functionality of the server object. One drawback of this approach is that although proxies are a good encapsulation concept they are difficult to manage: each remote class needs a proxy; the binding to the proxy introduces a certain overhead which is significant if interaction only takes place once and proxies are hard to program. An alternative implementation would be to use a generic run-time routine to handle all remote invocations. The routine must interpret some template describing the parameters at run-time to generate messages. In the presentation the author related that his experience in writing the library showed that although more complex proxies are more simple to integrate in a statically typed checked language like C++.

Although several possible choices could be considered for the implementation language, C++ was the one with more industrial impact. Multiple inheritance also helped to construct a class hierarchy that supported the interface virtual classes and the implementation classes for clients and servers in a coherent way.

3.5 Discussion

Q: How do managers differ from metaobjects in MUSE?

A: I don't know.

Q: Can you take advantage of hardware latency/intelligence?

A: Our network is smart. Download code and let it chew on it, freeing the CPU for other stuff.

Q: Is invocation an adequate mechanism?

A: No. We need streams too (consider a camera/VCR). Objects interact in more ways than just via invocation.

Q: Proxies seem to be hard to manage and give you a performance hit, don't they?

A(Gourhant): You can't use proxies for everything.

A(Guedes): C++ lends itself to proxies, though the creation/deletion overhead hits you.

Q: I think Fragmented Objects is the right abstraction. You need a typed interface. Proxies are one way to do it. Proxies out to be lightweight sometimes, shouldn't they?

A: There are instantiation costs—both for proxies and SVC.

Q: Proxies and stubs are different, aren't they?

A(Jones): A proxy can possess local intelligence.

Q: The word proxy means "on behalf of..." so intelligence seems like it could be bad, especially if the proxy does the Wrong Thing, wouldn't you say?

A: A proxy is a way to get at a fragmented object. It can do local things like caching answers in some cases.

Q: Can't you use fragmented objects to break authentication, since its code is in my address space?

A(Gourhant): The client only sees the proxy.

A(Guedes): In Mach, ports are capabilities.

Q: But when Joe User creates a server, couldn't it potentially do nasty things?

A: Trusted Computing Bases is a different issue (policy vs. mechanism).

Q: How is Fragmented Objects different from [unintelligible]'s Distributed Objects?

A: I have never heard of [unintelligible].

Q: Mapping into more than one address space breaks encapsulation, doesn't it? Is this bad?

A: Different objects should live in different address spaces—otherwise it "feels" wrong. There are optimization and encapsulations, and you need to distinguish between the two.

Q: What about uniform address space/uniform I/O model? I think that's dumb. Pretending there is no distribution is naive and won't scale. Even 64-bit identifiers won't help. You need to translate. It is not part of the OS.

A: Each administrative domain can have its own model—the objects in it just answer messages. You can make global names and compare them. Reference formats can be changed along the way as messages move around the network and cross administrative boundaries. Of course, you have to think about how you use references—don't just expect them to refer to the same object.

3.5.1 Additional Summary

Three of the papers talked explicitly about proxies, the initial discussions were centered on this topic. There were three main questions regarding the use of proxies:

- What is the fundamental difference between proxies and plain stubs?
- Proxies seem hard to manage and give you a performance hit.
- Do proxies compromise security by executing in the user address space?

In the first question there are two aspects to consider. Proxies have a well defined class and therefore its interface can be considered within an abstract object model, the type model is therefore more elaborated than simple routines hiding message passing mechanisms. The second issue, which is more correlated with the proxy designation, has to do with the possibility of delegating some of the functionality from the server objects to the user. Although the intelligence transferred to the proxy is a mere qualitative classification proxies are supposed to have some internal intelligence allowing to optimize the transfer of information. Stub classes are a mere limit case of a proxy.

The performance penalty is clearly on the association/dissociation with the proxy, which involves binding and sometimes dynamic linking. A generic run-time routine could avoid these overheads but its integration on existing programming environments particularly on static type checked languages like C++ is not obvious.

Proxies are mapped in the user address space therefore there is no protection to the information which they store. The general approach is that the user is free to do whatever he pleases as he will only hurt himself. This is essentially coherent with the Unix and Mach view of libraries.

To store protected information in proxies one may use capabilities if the underlying system provides support for them. For instance Mach ports are used as identifiers in the emulation library.

A different issue concerns the knowledge of the internal interface between client and server and being able to use this to break the server protection mechanisms. This problem is not more relevant to proxies than to normal stubs and relates to trusted computing bases.

The second discussion topic was centered around the model of identification provided to the application programmer. There are two opposite approaches to consider: a global identification scheme or to assume heterogeneous management domains where identifiers are local and translate whenever used. The discussion was very controversial but essentially everyone agreed that the unique identifiers may be efficiently used inside management domains where unicity can be easily guaranteed, this may provide efficient invocation mechanisms and solutions for handling persistency and heterogeneity. The problem remains on the definition of the gateways for crossing management domains.

Each administrative domain can have its own model—the objects in it just answer messages. You can make global names and compare them. Reference formats can be changed along the way as messages move around the network and cross administrative boundaries. Of course, you have to think about how you use references—don't just expect them to refer to the same object.

4 Third Session: Using the Object-Oriented Approach in OS Design and Implementation,

Sabine Habert, chair

4.1 LIPTO: A Dynamically Configurable OS Kernel,

Norman Hutchinson (University of Arizona)

LIPTO is a dynamically configurable, language-independent, object-oriented kernel. It provides communication services as a collection of protocol objects. Those protocol objects can be used to compose different protocol suites. That set of protocols is a user-extensible collection of interfaces within realms.

A realm is an interface, which is defined at kernel configuration time. As protocols are stackable objects, interfaces are paired (downcall interface, upcall interface). Protocol objects explicitly support one or more realm interfaces, and they may be dynamically loaded in any address space at any time. There are proxy objects for remote access.

The system supports processes, address spaces and inter-address space communication. The only objects that are known by the kernel and that may be passed through an interface are: addresses, messages, protocol objects, and names that are composed of an address-object pair.

4.2 The Choices Approach to Object-Oriented Operating Systems,

Roy Campbell (University of Illinois at Urbana-Champaign)

Choices is an object-oriented operating system written in C++. The position of the authors is that the OS should be object-oriented through and through, from user applications down to devices. All system entities are thus modeled as objects. User-visible kernel objects are represented by ObjectProxies in user address spaces.

The authors hope that an effort of standardization will be made by the object-oriented operating systems communities, both academic and industrial. The definition of an "open" inheritance hierarchy for standard libraries would help to port applications between different systems.

4.3 Towards Object-Oriented Structures for Open Operating Systems,

Clement Szyperski (ETH Zurich)

There is "sex appeal" in UNIX's "everything is a file" approach. Smalltalk says "everything is an object, and so belongs to some class." But what is an object? A conceptual integration, but not a useful classification. Introducing classification allows to develop abstractions.

Ethos, an experimental object-oriented operating system, is based on the carrier/rider separation for the canalizing and formatting of I/O to multiple devices. The carrier/rider separation allows to separate data format from access and avoids the multiple-inheritance dilemma. Carriers and riders classes have strongly-typed interfaces. Several riders can be connected to a single carrier.

4.4 Object Orientation in Operating Systems: SPRITE

Brent Welch (Xerox PARC)

The Sprite system is written in C, but uses object orientation to manage complexity. Sprite is 4.3bsd compatible. It has a distributed filesystem, process migration, and supports heterogeneous networks.

There are two main classes of objects: pathnames and I/O-streams. To goal is to provide general access to all objects in a distributed system. Processes migrate, but not objects.

There are several ways to handle pathname processing (user upcall, locally, RPC). A remote file is basically a proxy that does caching with delayed write of dirty cache blocks, process migration and recovery. The proxy state duplicates server state and it is used to rebuild volatile server state. The server detects conflicting state at recovery time.

4.5 Discussion

Q(to LIPTO): Why did you require explicit realms?

A: We don't want names to determine types—if operation "ping" does different things for a network than for process management, things break. Name collisions hurt, otherwise.

Q(to Choices): Can you inherit from new classes?

A: Access via nameserver gets you at classes as well as objects. There is protection between the user and the kernel.

Q(to Choices): Would the user only see a subprogram library? Why is your system object-oriented?

A: That's where language independence lives. Users can have objects other than OS-objects, of course. The system interface lets you build objects.

Q(to Ethos): What relations exist between carrier and rider?

A: Some kinds of matches make no sense. The objects should cooperate at the highest possible level (for instance, a byte-munger ought to work on generic streams, not just files).

Q(to Sprite): What is the benefit of object-orientation in your system?

A: Classification. It allows to keep track of who is providing what services. (There is a type-hierarchy but no code inheritance.)

Q(to Sprite): Why not code reuse?

A: Specifications are reusable, not code (between different machines). Interfaces should be developed—code will change.

Q: You are writing an OS in o-o style to make it extensible. How does this compare with the microkernel and user stuff?

A(Campbell): We're moving stuff out of the kernel into user space, and implementing multilevel protection.

A(Welch): I/O are good reasons (efficiency paramount among them) for things to be in the kernel.

A(Russo): Kernel calls are optimized by the hardware, but what is really needed is to optimize multiple space crossing paths. Moving things around is a tuning issue.

Q: Microkernels are a win with object-orientation, aren't they?

A: It's nice to standardize the low-level interface. But for performance reasons, some things should be in the kernel. A dynamically configurable kernel is a win.

Q: What is a good interface? [Discussion about specifying behavior broke out at this point with people flinging things around too fast to write down verbatim. *ed*]

A: Locking strategies should be part of the interface.

A: Interfaces should not just be collections of names and types. You may want, for example, to choose your implementation of the type. A good type system is important.

Q: Don't you think that it's also important to know what that interface does not guarantee?

A(Russo): It is uncheckable, but we should still try to specify what it's for, what it does, what it doesn't do—can use pre- and post-conditions to describe behavior.

Q: What do you gain from a thin object-oriented level? I say it gives a consistent way to talk to the OS.

A: The OS needs to be object-oriented to support this well/flexibly.

5 Fourth Session: User-defined vs OS-defined objects,

Brent Welch, chair

5.1 Stackable Filesystem Layers

Jerry Popek (UCLA)

Unfortunately the folks from UCLA didn't show up to talk about stackable file system layers.

5.2 A Toolkit for Interposing User Code at the System Interface,

Michael Jones (CMU)

Mike Jones described a toolkit for the interposition of user code at the system call interface. The basic idea is to make the system call intercept mechanism more accessible to programmers. There are several useful applications of system call interception including emulation of system calls in library code, forwarding of system calls to user-level server processes, tracing and debugging aids, and protected execution environments for suspect programs. While system call intercept mechanisms exist in several commercial operating systems, the main obstacles to such applications are the lack of programming tools to deal with system call interception. Jones proposes a toolkit that understands the various abstractions associated with the system call interface. This includes file descriptors, signals, and processes. The toolkit takes care of the mundane details of intercepting system calls and it provides a higher-level, object-oriented (i.e. files, signals, processes) interface to the system calls.

5.3 Multiple Views of [the] RESC Distributed Object System,

Yasushi Shinjo (Tsukuba)

Yasushi Shinjo described the notion of multiple views in the RESC operating system. System services are cast into classes in this system, and they can be subclassed to provide different views of the services. The examples described in the talk were a compressed file system and a tape archive (UNIX tar) file system. The service subclasses can be layered, or "stacked," to achieve a compressed tar image for a file system. Clients interact with servers through an RPC system, and they will transparently be routed to the correct server for the version of the filesystem they are accessing. The current implementation is layered on top of UNIX and uses SUN RPC for client-server communication. A question was raised by the audience about maintaining consistency among different views. The answer was that it could be achieved with the help of file modification callbacks from the supporting operating system [or the other service sub-classes], but since these don't exist yet then there is no support for consistency.

5.4 Operating System Extensions to Support User Objects,

Andy Olson (Chemical Abstract Service)

Andy Olson described "Operating System Extensions to Support User Objects" in the context of a network management expert system implemented on top of SunOS. Knowledge about network management was encoded in a class hierarchy of about 50 classes and about 100,000 objects. Objects are mapped into virtual memory, and the sheer size of the system leads to practical problems such as limitations from the underlying operating system. Objects are grouped into "accounts" that are mapped in together and share access protection and consistency. Objects can reference other accounts and objects via universal, opaque IDs that are managed by the runtime system. The system supports distributed and concurrent access to the object database. An "account manager" is in charge of arbitrating access to accounts, responding to update notices, checkpointing an account, and doing rollback for recovery. The manager is implemented as a coroutine by the runtime system. Recovery is needed because of network failures and because of deadlock. Deadlocks are broken based on a priority scheme. In addition to mapped files, which are provided by most modern operating systems, this object system would benefit from modification callbacks so that independent agents could keep track of changes to the database more efficiently.

5.5 Discussion

Q (to Jones): You have to trust the interposition toolkit, don't you?

A: Yes, but it's just another application as far as the OS is concerned. It doesn't have special privileges/access.

Q: Who decides when to interpose?

A: It's explicit at application launch time. Like I/O redirection.

Q: How do you do coherent caching in the ZFS complicated example?

A: It's hard. Keeping track of derived objects is tricky, but you can use "changed" messages to invalidate caches. Either need bidirectional communication or keep checking modification information all the time.

Q(to Olson): You have callback for modification, right?

A: They synchronize between themselves before writing, then other objects and accounts are notified.

A(Shinjo): Object IDs—a client can send messages without knowing the location of its recipient (via a nameserver). The nameserver knows about the existence of different sites, so it can serve names for them. This permits replication.

Q(to Jones): Do you have/need a toolkit for running untrusted binaries?

A: Could put agents in other address spaces so they couldn't be clobbered (let the agents read/write the untrusted application's memory).

Q(to Olson): What OS features do you need?

A: Need notification.

Q: How do you go about sharing database stuff?

A: Only long-term IDs are used inside objects, so you can map them into VM in a location-independent manner. This is all hidden from the application.

6 Panel discussion: What is an Object-Oriented Operating System?

Alan Snyder, chair

6.1 Ross Finlayson (Apple)

A virtual machine for application level software. It does protection, communication, and is defined by its exported virtual machine. So an OOOS exports an o-o virtual machine regardless of how it's implemented.

Objects = State + Behavior + Messages + Inheritance of interface + Data Abstraction + Polymorphism (which you get "free" from inheritance).

There are many kinds of OOOSs—there is no one "right" approach. Need to choose criteria relating directly to o-o nature of the system, such as: object addressing (need VM for protection), support for programming-language object references, active vs. passive objects. Inheritance of interface *plus* behavior? VM relates to IPC.

6.2 Marc Shapiro (INRIA)

I'm going to make a provocative statement: Mach and Amoeba are not object-oriented. There is no predefined set of objects. Things within processes are named by convention only—really it's server-oriented.

Choices is object-oriented.

SOS supports objects: data + code + type, etc. Naming, upcalls, persistent-, mobile- and fragmented objects. SOS is an "object support system" but is not object-oriented since it's ugly inside.

SOUL = OSOS + OOOS. Nucleus + object management components. Multiple subsystems.

6.3 Vince Russo (Purdue University)

Working on "Better Choices" [now called Renaissance—*ed*].

Remember, this is OOPSLA, not SOSP. Objects, classes, inheritance, polymorphism are very important.

OOOS = conventional OS (VM, devices, filesystem, network, processes, etc.) but must be object-oriented in implementation—everything is an object. The object-oriented interface must be exported to application, so the applications can take advantage of the OS frameworks. Otherwise, what good do the object-oriented internals do. This was meant as a rhetorical question. There are many software engineering advantages to OO internals that should not be overlooked as well.

6.4 Richard LeBlanc (Georgia Tech)

Clouds may or may not be an OOOS. Let's see. Objects + Threads. Objects are virtual address spaces.

There are two versions: version 2 is in C++ with microkernel—o-o all the way down.

Objects are not fine-grained. Persistent objects hold collections of fine-grained language-level objects.

Programming in extended C++ and Eiffel.

6.5 Jeff Chase (University of Washington)

I disagree. The goal is application integration. The object model is the basis for communication/sharing objects. Can integrate stuff into the language (typing); transparent invocation; uniform naming. I.e., make life easier for the programmer.

There are fundamentally different kinds of objects, in terms of granularity, access control, sharing characteristics, etc.

Data objects are not the same as resource objects. For data objects, clusters are good, and relying on the language for encapsulation is ok. Not true for resource objects. Need ports so they can hide in other address spaces.

Let's figure out how to identify, select and combine different kinds of objects.

6.6 Discussion

Q: Clouds isn't bad. Just different. So we should use different words.

Q: How does an object in Clouds differ from a process?

A: Its interface and long-lived nature.

Q(to Shapiro): What is object-oriented support?

A: Support user-defined arbitrary objects. Interface based.

A(Chase): Ports are enough.

A(Russo): A Port is just one kind of object.

A(Finlayson): Marc wants too much (persistence, migration).

Q: There is object-oriented programming in an OS. There are object based OSs. There are things made with OO techniques (like OO analysis) in whatever language. So where is the OOOS/OODB boundary?

A(Finlayson): Support for OODB is VM plus files. We should support DB needs. Also object management people. OS = protocol suite, not a bunch of code.

Q: What are the key bits?

A: There is a panel later to discuss this later in the conference.

A(Shapiro): Code reuse is good. Common components are good, especially for distributed systems—like garbage collection, naming and invocation, VM.

A(Russo): Should be driven by applications, not by OS designers. The two groups need to talk to each other.

Q: Is one thing about an OOOS the ability to create new object types?

A: Yes.

Q: You said OOOS and object management system are different. You need query support. Won't they become one eventually?

A: Yes.

Q: Get requirements first, then go build the system. So we need to support higher layer abstractions. The PS is there to support applications. Don't you just need protocol support?

A(Campbell): Specialized kernels are doable. If you let applications twiddle OS stuff it's okay.

Q: What are the right abstractions?

A: I'd like to see synchronization, etc., etc.

Q: Are the OS and the kernel different? There is code that's good to share—is that necessarily part of the OS?

A(Shapiro): The OS is a common set of things with the kernel inside ...