On Secure, Dynamic Customizing of a Meta-Space-Based Operating System

by

Michael Horie
B.Sc., University of Victoria, 1994

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

We accept this dissertation as conforming
to the required standard

Dr. Eric G. Manning, Co-Supervisor (Departments of Computer Science and Electrical and Computer Engineering)

Dr. Gholamali C. Shoja, Co-Supervisor (Department of Computer Science)

Dr. Mantis H.M. Cheng, Departmental Member (Department of Computer Science)

Dr. Kin F. Li, Outside Member (Department of Electrical and Computer Engineering)

Dr. Yasuhiko Yokote, External Examiner (Sony Architecture Lab)

Co-Supervisors: Drs. Eric G. Manning and Gholamali C. Shoja

ABSTRACT

Continuing advances in hardware and in software applications are pushing traditional operating systems beyond their limits. This is largely due to the fact that these advances, and their associated requirements, were not foreseen at operating system design time. This becomes particularly apparent with multimedia applications, whose demands for guaranteed quality of service differ considerably from those of most traditional applications. To ensure that many future requirements will be met, along with many existing demands, one solution is to allow applications to customize their operating system throughout its life-time. However, opening up an operating system to application-initiated changes can compromise the integrity of the system, suggesting the need for a security model. Like any other aspect of a customizable system, such a security model should be securely customizable, too. Therefore, this dissertation introduces *MetaOS*, a securely- and dynamically-customizable operating system which has a securely- and dynamically-customizable security model.

MetaOS employs four types of building blocks: meta-levels, meta-spaces, meta-objects, and meta-interfaces. Meta-levels localize customizable system services. Meta-spaces act as firewalls which prevent custom alterations from affecting unrelated meta-spaces and their applications. Meta-objects help to modularize meta-spaces into smaller, easier-to-maintain components. Finally, meta-interfaces provide the heart of the secure customizing model.

MetaOS meta-interfaces are strictly divided into declarative and imperative interfaces, providing a basis on which to distinguish between calls which only affect the invoking application (i.e., local-effect calls), and calls which could affect other applications as well (i.e., meta-space-wide-effect calls). By giving free access to the former, but limiting access to the latter, a basic balance between flexibility and security can be struck. Additional flexibility is achieved by allowing new local and meta-space-wide-effect calls to be added dynamically, by permitting untrusted applications to negotiate changes with trusted meta-space managers, and by allowing untrusted applications to migrate to cloned meta-spaces and alter them as necessary.

Examiners:

_____

Dr. Eric G. Manning, Co-Supervisor (Departments of Computer Science and Electrical and Computer Engineering)

_____

Dr. Gholamali C. Shoja, Co-Supervisor (Department of Computer Science)

_____

Dr. Mantis H.M. Cheng, Departmental Member (Department of Computer Science)

_____

Dr. Kin F. Li, Outside Member (Department of Electrical and Computer Engineering)

_____

Dr. Yasuhiko Yokote, External Examiner (Sony Architecture Lab)

Table of Contents

List of Tables

List of Figures

Acknowledgments

Like most dissertations, this work was not written in isolation. I particularly want to thank my two supervisors, Dr. Eric Manning and Dr. Gholamali Shoja, for spending many hours listening to my progress reports, reviewing my papers, and helping me to stay focussed on the essentials of MetaOS. I also want to thank them for their financial support, which helped me to attend a number of important conferences, and to meet several key people in the area of customizable operating systems. Many thanks also to Dr. Mantis Cheng for thoroughly discussing security issues with me, and to Dr. Kin Li and Dr. Yasuhiko Yokote for their review of this work. I would furthermore like to thank Kenichi Murata for spending many hours explaining Apertos concepts to me, and for helping me understand Apertos source code. Thank you also to James Pang for writing a SFQ scheduler, and to Robert Bryce for helpful foundational discussions. Finally, I would like to thank all members of the PANDA group for their suggestions, and NSERC for their financial support.

Dedication

To Mom, Dad, and my Sister, Ruth

Chapter 1

Introduction

1.1 Benefits of Customizing

Continuing advances in hardware and in software applications are pushing traditional operating systems beyond their limits, largely because these advances, and their associated requirements, were not foreseen at operating system design time. In the past, this could be observed with page-replacement policies: Initially, only a least-recently-used (LRU) page-replacement policy was provided, because of its suitability for many types of applications [1,2]. However, it turned out to be inadequate for database applications which access data sequentially; these fare much better under a most-recently-used (MRU) page-replacement policy. Indeed, Cao et al. demonstrated that customizing the page-replacement policy for such applications can result in efficiency gains of up to 45% [2]. For this reason, many systems now support both types of paging policies.

More recently, the problem of unanticipated advances has been particularly apparent with multimedia applications. Multimedia applications, in general, require that the system guarantee a certain quality of service (QoS) to reduce or eliminate undesirable side effects, such as delay jitter. However, a traditional operating system scheduler may not be able to support such a requirement [3-5]. Furthermore, existing LRU and MRU paging policies are also insufficient for certain multimedia applications [6], requiring yet another page-replacement policy. One solution to this on-going problem of

meeting new requirements in a timely manner is to allow operating systems to be customized throughout their life-times.

Recent work on customizing systems has focused intensely on performance, for good reasons: In addition to the above 45% gain, researchers have also reported a 52% gain in speed after customizing the synchronization mechanism [7], as well as a 72% gain after customizing disk reads [8]. Some even reported a 90% gain after customizing multiprocessor communication [9]. However, customizing operating systems should not be driven solely by performance improvements. Adding QoS guarantees, for example, is less about adding performance, and more about adding new functionality. Furthermore, some researchers have criticized current research as focusing too narrowly on performance, at the expense of other issues [10]: These include containing the ever-growing complexity of operating systems, and finding ways to propagate new operating system ideas more rapidly. Interestingly, these last two issues can also be addressed by customizable systems, provided they are structured properly. For example, if such a system is properly modularized, it can be much easier to understand than a tightly-coupled, monolithic system [11]. Moreover, if new system ideas are implemented in the form of modules, it can be easy to package them and, on request, ship and install them on other customizable systems via a network.

In addition to lack of performance and inadequate system support, failure to allow customizing has further downsides. For one, barring applications from modifying system services can result in application code which tries to circumvent operating system functionality by re-implementing parts of it at the application level [1,12]. However, this approach raises several objections: It is a duplication of code and effort; it

makes application code larger and more complex; and it is an invitation to additional software defects.

Another possibility is that code will be structured in a way that will exploit undocumented quirks of the underlying implementation [12,13]. For example, modules could be written in such a way that they fit onto a certain number of pages. However, this approach results in code that is hard to understand and maintain, as well as being unduly sensitive to the underlying implementation. Of course, a customizable system by itself will not prevent poor programming practices; but in many cases, it can prevent the type of work-arounds mentioned above.

Systems which support customizing need not be limited to extending system functionality, but can also allow modification of existing services. This permits existing, defective code to be replaced by new code which fixes the defect. As mentioned earlier, customizable systems can furthermore make it easy to distribute and install this new code.

The author does not claim that it is possible to design a customizable system which will meet the needs of all applications, past, present, and future. However, even if such a system cannot be designed, does not mean that one should give up on the idea of customizing altogether: Just a limited degree of flexibility can already allow a large number of applications to execute more effectively.

The benefits of customizing have certainly not escaped the operating systems community: Even early operating systems such as HYDRA [14,15], RC 4000 [16] and VM/370 [17] supported a limited degree of customizing. However, recent advances, such as faster and more reliable networks and multimedia applications, demand levels of

control that traditional operating systems cannot fully supply [18-20]. This realization has led to an intensified effort to find more effective approaches to customizing.

1.2 Benefits of Dynamic Customizing

A fundamental issue which must be addressed when creating customizable systems, is deciding the point at which customized changes can be introduced [21]. If customizing is allowed only at boot-time, the designer must make sure that a system administrator can easily augment or replace existing code upon system initialization. However, complications associated with run-time changes, such as preventing applications from introducing customized system code which attempts to gain additional, unauthorized access, will not be a concern.

Unfortunately, it is not always clear at boot-time which applications will be run, and what their requirements will be. One solution is to install a large number of different, customized policies, but this wastes resources, and there is always a chance that a crucial policy will be missed. Another solution is to shut down a system whenever it is to be customized, but there is a cost associated with bringing down a system, re-configuring it, and re-starting it. In some cases, this may only be a nuisance, but for systems supporting mission-critical applications, such as on-line reservation systems, telecommunication systems, and industrial control systems, the cost is much more substantial. In these cases, customizing at run-time is more attractive, provided that the additional security and run-time versioning costs can be borne.

1.3 Contributions of This Dissertation

While customizing operating system services is helpful, a problem arises when concurrently-executing applications try to customize the same service in different ways, or try to exploit the flexibility of the system to gain unauthorized access to resources. Operating system designers can approach this customizing problem in numerous ways [21]: One way is to ignore it altogether. This has been popular with some single-user microcomputer operating systems, such as DOS and MacOS, but it ultimately fails in the presence of multiple users, some of whom may try to sabotage or steal resources from fellow users. The other alternative is to provide some type of mutual protection among competing modifications, i.e., to support secure customizing. It is the goal of this work to introduce such a secure customizing model.

A few secure customizing models, also called *security models* throughout this work, already exist. These models range from simple schemes which limit applications to a small list of predetermined policies [2], to models which include a transaction mechanism in the kernel to undo any harmful customization [22]. The novelty of the secure customizing model introduced by this work is that it, too, can be customized, even at run-time. There are two main reasons why security should be customizable: First, just like any other aspect of a system, one type of security may not be suitable for all cases. For example, a highly-sensitive system, such as a system supporting the company payroll, may require two independent users to authorize a change to the operating system, whereas in a small, trusted research environment, research group membership may suffice. Second, just like any other aspect of a system, the security policy or mech-

anism of a system may be defective, and thus be subject to upgrades. Finally, for the same reasons that any other aspect of a system should be changeable at run-time, i.e., convenience or the mission-critical character of the system, security should also be securely customizable without having to shut down the system.

To fulfill its goal, this work first refines some existing operating system structuring ideas to provide a highly-modular, customizable system, called *MetaOS* [23,24]. It then introduces the dynamically-customizable security model, along with a default security mechanism and policy.

The system presented in this work is largely a centralized system which has been designed with best-effort and soft real-time applications in mind, such as video-conferencing, web servers, and payroll systems. The author is not aware of any aspect of the security model which would hinder its applicability for distributed or hard real-time control systems, such as those found in nuclear reactors, but a detailed study of this was left as future work. Finally, the main emphasis of this work is security, and not performance, meaning that performance, too, was left for future work.

1.4 Structure of This Dissertation

The rest of this work is structured as follows: Chapter 2 surveys prominent approaches to customizing operating systems. Chapter 3 discusses the MetaOS building blocks and security model. Chapter 4 analyzes the security model of MetaOS. Chapter 5 shows how the building blocks and security model can be used to implement traditional system services. Chapter 6 introduces the Java-based prototype. Chapter 7 discusses several

key experiments conducted with this prototype. Finally, Chapter 8 concludes this work with a brief summary, a list of future work, and some general thoughts.

Chapter 2

Related Work

2.1 Customizable Systems

There are many different ways of customizing operating systems, so for the sake of brevity, the following discussion is limited to seven of the most prominent customizable systems documented in the literature.

2.1.1 Spring

Spring [25-27] takes the microkernel approach [28] when it comes to customizing: Basic CPU scheduling, inter-process communication, capabilities [29], and memory management are locked away in the kernel address space. Naming, paging, network communication, and other services, however, are implemented by user-level servers, each of which may reside in its own address space for the sake of security. By replacing these servers, a system administrator can customize Spring.

Spring servers export their interfaces by means of an interface definition language, which allows services to be declared in a language-independent manner. Spring furthermore ties subcontracts to these interfaces. Subcontracts are in charge of connecting a client to a server, and therefore deal with marshalling and unmarshalling arguments and other transfer-related issues. More specifically, if a client application invokes a method exported by a server, the appropriate subcontract is activated to transfer the

arguments to the proper destination. This transfer will fail if the client does not hold the appropriate capability.

Subcontracts can be replaced by a system administrator to allow for different types of method invocation, marshalling and unmarshalling, transaction management, etc. Ultimately, however, all subcontracts depend on the highly-optimized, kernel-supplied inter-process communication mechanism to transfer arguments between address spaces.

Although Spring offers much flexibility in some areas, in others it is rather restrictive. CPU scheduling, for example, is locked away in the kernel. This poses a problem for multimedia applications, because they may depend on a scheduling policy which differs from the one which is provided. Another problem is that replacing servers and subcontracts is limited to the system administrator, leaving the average user with significantly less flexibility.

2.1.2 Cache Kernel

The Cache Kernel [30], illustrated in Figure 2-1, is essentially a microkernel, but for the sake of efficiency, it allows certain types of objects to be cached in the kernel's address space, namely, thread objects, address-space objects, and kernel objects. These will be discussed below in more detail. Directly above this kernel can be a number of application kernels, to provide applications with a full range of operating system functionality which the Cache Kernel itself does not offer.

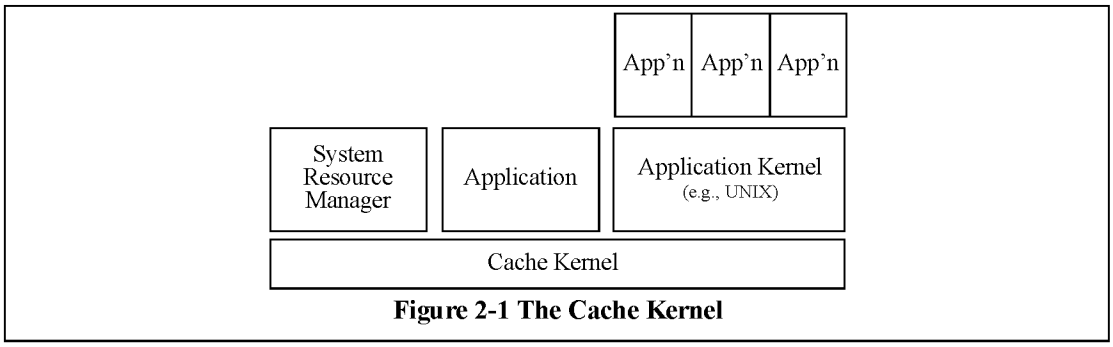


Figure 2-1 The Cache Kernel

Each application kernel thread is represented by a thread object, which becomes eligible for execution when it is loaded into the Cache Kernel. By controlling which of its thread objects are loaded, an application kernel can effectively control its scheduling policy as well as the scheduling policy of any application it supports. Should a thread block on an operation, the Cache Kernel can unload the thread, and ask the application kernel to save the thread's current state. To avoid unloading a scheduler thread object, every application kernel can lock a certain number of thread objects in the Cache Kernel.

The address space of an application kernel is represented by address-space objects. By loading such an object into the Cache Kernel, the data is transferred from disk to primary memory. Should there be no room left, address objects can be unloaded by the Cache Kernel in consultation with the applicable application kernels. Similarly, should a page fault occur, the Cache Kernel asks the applicable application kernel to load the appropriate address object. In this way, paging can be controlled. Address-space objects are also used to implement inter-process communication.

Finally, kernel objects provide some degree of security. Each kernel object determines the address spaces of an application kernel, its trap and exception handlers, and the resources that it can use, including the physical pages, percentage of CPU time,

and number of locked objects. The first application kernel that is loaded is generally considered the system resource manager, and only it can create new kernel objects.

Note that unloading objects is rather costly, partly because of the way they interrelate. For example, if an address space object is unloaded, all thread objects running in this address space object must be unloaded as well. Another problem is that the underlying kernel mechanism cannot be modified, including the basic fixed-priority scheduling algorithm. As a matter of fact, the designers mention that it has been burned into PROM. On the other hand, given its 139 kilobytes total size for code and data, the Cache Kernel itself is pleasantly small.

2.1.3 Aegis

Aegis [31,32] takes the microkernel approach much further than Spring or the Cache Kernel. Rather than implementing some abstractions in the kernel, Aegis was designed to implement none at all. The Aegis designers call this approach the *exokernel* approach. The only goal of an exokernel is to export hardware securely, including CPU, memory, interrupts, and so on. All ordinary operating system functionality, such as paging, must be provided by user-level library operating systems.

Hardware is protected by *secure bindings*, a *visible revocation protocol*, and an *abort protocol*. The secure bindings allow the kernel to restrict which library operating system has access to what hardware resource, such as a page of memory or the CPU. This also allows the kernel to ensure that library operating systems do not interfere with each other. Note, however, that libraries are not protected from the applications they

support. The visible revocation protocol allows the kernel to negotiate with a library operating system in case one of the system's resources must be allocated to another library system. If the library system refuses to cooperate, the abort protocol allows the exokernel to take control away forcefully from the library system.

As the designers of Aegis point out, creating an interface for an exokernel is very difficult. For one, it is highly hardware-dependent. For another, it is very difficult to export raw hardware securely. The abstraction-less approach also has to deal with the lack of sufficient information: If a packet arrives from the network, Aegis does not know which application owns this packet. Thus applications must provide packet filters for the kernel. Finally, multiplexing between library operating systems is still under kernel control. Thus, a completely policy-free approach was not achieved.

Aegis is probably the most flexible of all systems discussed in this chapter. This is because it does almost nothing towards implementing operating system functionality, leaving the operating system designer nearly total freedom. However, because Aegis largely focuses on removing services from the kernel, rather than implementing them, there still is a need to resolve the issue of securely customizing shared operating system services.

2.1.4 Fluke

The Fluke [33] system combines the microkernel approach with the virtual machine approach. In particular, the Fluke microkernel provides low-level CPU scheduling, inter-process communication, and memory management services, as well as interfaces

that outline higher-level operating system services. Fluke virtual machines then use and re-export this higher-level interface, also known as the *virtual architecture* or *Common Protocols API*. Because interfaces among virtual machines stay the same, virtual machines implementing complementary resource management policies can be composed by stacking these virtual machines on top of each other.

Fluke virtual machines do not have to implement all the features provided by the virtual architecture. Rather, they can selectively interpose on parts of the interfaces, relying on lower-level virtual machines to handle other parts. Moreover, calls to microkernel methods can be directly sent to the microkernel, allowing virtual machines to bypass the virtual machine hierarchy entirely. Thus, Fluke virtual machines are less like VM/370 virtual machines, and more like stackable user-level servers.

To ensure security, each virtual machine has direct control over all the virtual machines layered on top of it, i.e., its *children*. It accomplishes this by donating some of its resources to its children with the help of the low-level microkernel interface, and by ensuring that children cannot use any other resources without authorization. Since higher-level virtual machines can bypass lower-level virtual machines for efficiency reasons, Fluke provides lower-level virtual machines with a means to intercept these bypasses, if required.

Fluke emphasizes a hierarchical model, but at the same time, allows parts of the hierarchy to be bypassed for efficiency reasons. This dichotomy complicates security, while performance still suffers: Adding a virtual machine layer reduces performance by up to 30% [33]. Moreover, interfaces among virtual machines are fixed, so it is unclear how services that require methods and arguments beyond those provided by the fixed

interfaces can be added. Finally, the arbitrary layering of virtual machines is problematic. For example, if a virtual machine wants to offer persistence, but its supporting virtual machine delays writing to disk, the abstraction will not work properly [34]. Unless the supporting virtual machine can be securely modified, or the virtual machine hierarchy can be securely customized without breaking anything in the process, this problem remains to be addressed.

2.1.5 SPIN

The SPIN [35,36] operating system can be dynamically customized by intercepting calls to kernel methods, called *events*, and delegating them to application-specific implementations, called *event handlers*. Since applications may customize the same kernel methods in different ways, *dispatchers* decide which event handlers to call when a particular event occurs. For example, if a packet arrives over the network, SPIN raises the associated packet-arrival event. Then, if there are several different event handlers eligible to process the packet, the appropriate dispatcher is called to decide which handlers should be executed, and in what order they should be called.

To prevent an application's customized event handlers from interfering with other applications, four major requirements must be fulfilled: First, all customized handlers must be written in Modula-3, a language which enforces type-safety. Second, to ensure that only a trusted compiler was used, the binary code must be signed. Unsigned code will not be accepted, unless it is explicitly declared to be safe. Third, the linker ensures that customized event handlers only call kernel methods for which they have

proper authorization. Finally, the default, kernel-supplied event handler must agree to let these new, customized event handlers be installed.

Because much of SPIN's security is achieved at compile-time, SPIN is able to achieve very good performance. However, its run-time security mechanism has some drawbacks: By default, any number of event handlers can be associated with an event, bounded only by a lack of memory. This means that the latency of a kernel method is highly variable. For example, in the packet arrival scenario, there may be so many event handlers ready for execution that subsequent packets are lost. Also, event handlers are usually executed to completion, one at a time. If any one of these handlers fails to terminate, SPIN can hang. To avoid this, handlers can either be declared *asynchronous*, i.e., each time an event occurs, a new thread is created for each applicable event handler, or handlers can be declared *ephemeral*, i.e., they can be terminated. However, to avoid terminating a handler that is in the process of calling a method which alters the kernel state, and thus leave the kernel in an inconsistent state, ephemeral handlers may only call other ephemeral handlers. Unfortunately, both of these solutions do not appear to be entirely desirable: One is limited by cost, the other by flexibility. Also, requiring customized code to be written in Modula-3, or be faced with visual code inspection, poses a problem to those who possess little or no knowledge of Modula-3.

2.1.6 VINO

Like SPIN, VINO [22,37] allows applications to customize the kernel. However, VINO's security mechanism relies less on a particular language, and more on software

fault isolation [38] --- a technique which confines code to specific memory addresses within an address space --- and transaction management. In particular, any customized code, or *graft*, is modified by a trusted tool to ensure that the graft does not access memory to which it has not been given access. Similar to SPIN, the tool signs the code, and the dynamic linker is called to install the code and check that the graft does not directly call kernel methods without authorization.

While this prevents grafts from wandering out of their assigned memory space or from *directly* calling methods without authorization, it does not prevent grafts from failing to share resources properly, or from *indirectly* calling kernel methods to which they have no access. For example, the VINO designers point out that C++ virtual methods could be used to circumvent the dynamic linker's security check, which is only done at link time. For this reason, all grafts are executed as transactions. In particular, whenever a kernel method is called to which a graft is attached, a new thread is spawned to execute the graft, and a transaction is started. Should a graft take too long to execute or misbehave in other ways, VINO can abort the graft, and undo all its effects. Otherwise, the actions of the graft are committed.

VINO provides tighter default security than SPIN, but it comes at a cost. Both software fault isolation and transaction management significantly add to the complexity of the kernel and the cost of executing a graft. This is especially the case if one graft calls another, as this requires dealing with nested transactions. Furthermore, it is unclear what happens when the transaction manager itself is attacked.

2.1.7 Apertos

Apertos [39-41], a pioneer in the area of highly-flexible operating systems, has an architecture as illustrated in Figure 2-2, and gains its flexibility through meta-objects [42] and meta-spaces. An Apertos meta-object is an object which defines how a system service is provided to an application. Alternatively, one can also define a meta-object as an object which controls some part of an application's run-time environment. For example, a scheduler meta-object determines the order in which applications are scheduled. An Apertos meta-space is the collection of meta-objects which make up the primary execution environment of an application. For instance, a multimedia meta-space would consist of multimedia-capable mailer, namer, and scheduler meta-objects.

At run-time, an Apertos application is supported by one of many meta-spaces. Each meta-space must contain a special meta-object, called the *reflector*, which receives incoming requests to the meta-space and forwards them to the appropriate meta-object. For example, if an application sent a scheduling request to its meta-space, the reflector would receive this request, and forward it to the scheduler meta-object. Note that the application only sees this reflector. It does not directly have contact with --- or knowledge of --- other meta-objects in its meta-space.

Meta-objects are themselves supported by other meta-spaces. Meta-objects that make up the supporting meta-spaces are, in turn, supported by further meta-spaces. To avoid an infinite scenario, this hierarchy is limited to only a few levels, with two interdependent meta-spaces making up the bottom. The entire hierarchy is supported by a microkernel, called *MetaCore*, which mostly handles context switches and interrupts.

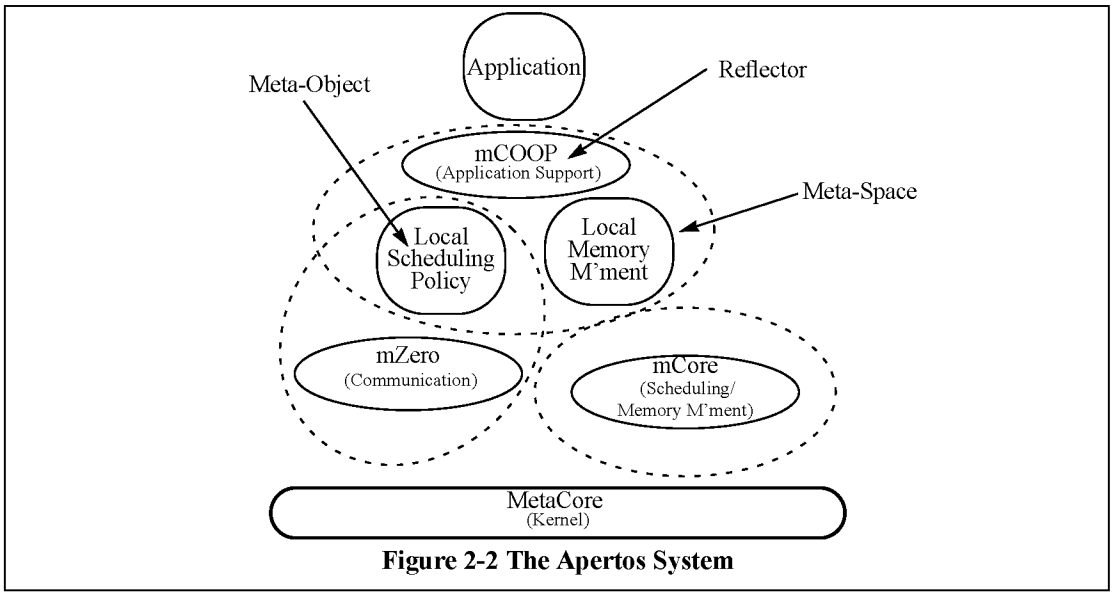


Figure 2-2 The Apertos System

One major advantage of this approach is that it gives an application the flexibility of selecting suitable system services --- in the form of meta-objects --- for its execution environment. This execution environment, i.e., the object's meta-space, can furthermore be easily packaged and migrated to other machines. However, Apertos does not prevent applications from introducing meta-objects which interfere with or sabotage the system or other applications, partly because the system was targeted for embedded devices. Because each Apertos meta-object potentially resides in its own address space, the system can also be costly. The cost has been reduced by allowing an object to lend its context to another object, but this only works under certain circumstances.

2.2 Miscellaneous Approaches

Much research has also involved customizing *parts* of an operating system. The following discusses three of the most prominent approaches.

2.2.1 ACFS

ACFS [2,43] allows applications to select their own page-replacement policies. Specifically, users can associate files with certain priorities, and then specify a page replacement policy for these priorities. For example, the user could give file F a priority of 1, and then associate priority 1 with a LRU policy. This would cause ACFS to use LRU when replacing memory pages associated with file F, or any other file with priority 1. Note that pages associated with lower-priority files are more likely to be replaced than pages associated with higher-priority files, and that ACFS currently supports only MRU and LRU.

A problem with this approach is that the relationship between priority and paging policy is not obvious. It would seem more intuitive to maintain these separately. ACFS is also rather restrictive, since it does not allow applications to use any policy other than LRU and MRU; and new policies cannot be added dynamically. On the other hand, ACFS has been shown to be rather safe and efficient.

2.2.2 Hipec

Hipec [44,45] supports a scripting language which lets applications provide their own memory page-replacement policy. In particular, Hipec provides twenty commands relating to the replacement of a memory page. The application can use these commands to write a customized paging policy, and pass it to the kernel. The kernel freezes this policy, to make sure that the application won't alter it while the kernel executes it. Then,

when an application's page must be replaced, the kernel attempts to execute the application-specific page-replacement policy.

To ensure that the user does not write a malicious policy, all script executions are timed. If a script times out, the kernel will stop executing the offending application's script, and continue with other duties. Furthermore, since the scripts are interpreted by the kernel, presumably the kernel can intervene if an application tries to steal pages it does not own. However, this is not clearly spelled out in the existing Hipec literature.

A big advantage of the script approach is that it allows kernel internals to be hidden. This makes it easier to write, and safer to execute, customized policies. However, it is difficult to come up with a scripting language which is generic enough to cover most policies. Also, not being allowed to access kernel structures is a double-edged sword. Without access to them, it is not possible to change the underlying mechanisms or policies that are not represented by the script; in Hipec, this includes the way pages are allocated among applications, as well as file prefetching.

2.2.3 Proof-Carrying Code

Proof-carrying code [46,47], or *PCC* for short, is perhaps the most challenging approach to ensure the security of customized code. PCC requires the kernel to publish a security policy. It is then incumbent upon the customized code to formally prove that it complies with this security policy. If the kernel finds the proof to be valid, it goes ahead and installs the associated customized code. This means that once the code has been

checked, no more checks are required at run-time since the code has been proven to be secure. Thus, once installed, customized code can run at full speed.

It must be emphasized that in its ultimate form, PCC should allow any customized code to automatically generate a formal proof, using first-order logic, which demonstrates its compliance with the kernel's security policy. Moreover, the kernel should automatically be able to check this proof. As such, PCC draws heavily on experience in software verification, but this automation is a challenge, and thus PCC currently requires user intervention. User intervention, of course, is a potential breach of security. Furthermore, it is doubtful whether this approach, alone, can ever be scaled to a real-world environment, especially in light of non-computable problems [48]. For example, if the kernel requires that the customized code terminate, the proof generator essentially must deal with the halting problem, which has been proven to be non-computable in the general case [48].

2.3 Summary

The systems mentioned in this chapter are all flexible, although the degree of flexibility varies. Spring and the Cache Kernel, for example, prevent the basic CPU scheduler from being changed, whereas Apertos does support such a change. The security models also differ. For one, Apertos does not even prevent applications from sabotaging each other; SPIN primarily relies on Modula-3; and VINO depends on its transaction manager for security. Yet none of these systems considered dynamically-customizable security. The next chapter introduces just such a security model.

Chapter 3

MetaOS Model

3.1 Building Blocks

MetaOS is built on the basic concepts of Apertos. In retrospect, a number of other systems could also have served as a basis for this work. However, given the fact that Apertos already allows radical run-time changes, while not providing a strong security model, Apertos turned out to be an ideal choice. This section discusses the basic building blocks of MetaOS --- meta-levels, meta-spaces, meta-objects, and meta-interfaces. The security model is introduced in the subsequent section.

3.1.1 Meta-Levels

As illustrated in Figure 3-1, MetaOS is structured as three major levels --- the base level, the meta level, and the meta-meta level [39]. The base level is intended for applications. The meta level is meant for meta-spaces, each of which supports a collection of applications sharing similar requirements. For example, multimedia applications are supported by a common multimedia meta-space. The meta-meta level is intended for the Master Meta-Space. The Master Meta-Space is similar to a microkernel: It is in charge of multiplexing hardware resources among competing meta-level meta-spaces, which in turn make them available to their applications. These multiplexing policies can be customized dynamically, as will be discussed in Section 3.1.4.

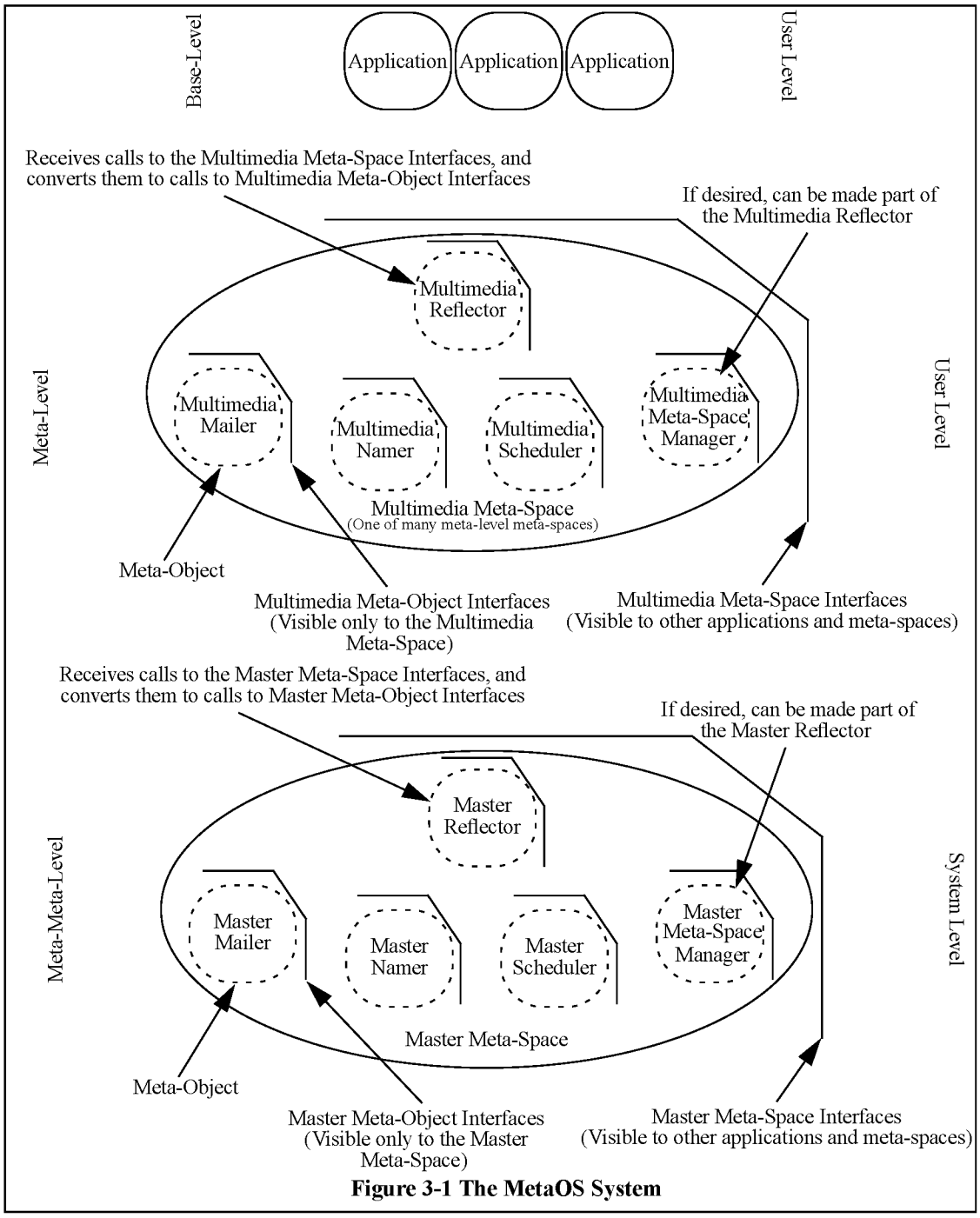


Figure 3-1 The MetaOS System

This hierarchical approach is an important first step in making MetaOS securely customizable, because the hierarchy, in combination with meta-spaces, localizes and isolates certain policy decisions. For example, by splitting the system scheduler into a hierarchical scheduler, consisting of a Master Scheduler and several meta-level schedulers, it becomes much easier to replace one of the meta-level schedulers without compromising the functionality of other meta-level schedulers. Thus, one could upgrade the multimedia scheduler of the meta-level meta-space in Figure 3-1 to a more efficient version, without affecting other meta-level meta-spaces and their applications, or the Master Meta-Space. As a consequence, application-specific control becomes less of a threat to overall system security. Also, because the system scheduler was split into many parts, each part itself is potentially simpler, and therefore easier to maintain.

The three-level approach is the first major deviation from the original Apertos design. Apertos strives to support an *infinite reflective tower* [49], by making the meta-space supporting scheduling and memory management (*mCore*), and the meta-space supporting inter-process communication (*mZero*) meta to each other. The notion of an infinite reflective tower, found in literature on reflective languages [49, 50], is important because it allows the same model of customizing to be applied recursively. For example, to change the way the base level is being executed, the reflective model demands that changes be made at the meta level. To make changes at the meta level, one must go to the meta-meta level. This sequence can potentially continue indefinitely, although meta-levels are generally only created when necessary. Apertos exhibits great generality in allowing indefinitely-large numbers of meta-levels. Still, as our study of Apertos shows, the Apertos implementation of the reflective tower increases the complexity of the sys-

tem design, and comes at the cost of a large number of context switches [51]. Furthermore, the author believes that most modifications of interest can be handled by altering the first two meta-levels: The meta level is needed to customize the execution environment of applications, and the meta-meta level is needed to provide different hardware multiplexing schemes for meta-level meta-spaces; but it is unclear how important it would be to change the meta-meta-meta level, or any level beyond that. For these reasons, MetaOS supports only three levels, simplifying the design. As will be demonstrated in Section 3.1.4, this did not result in a loss of flexibility.

Meta-levels come at a cost: Every time a system call is made, and every time system events such as time-outs are triggered, the appropriate meta-level meta-space must be contacted. This cost is similar in nature to the performance penalty which microkernel architectures face [52], and may sometimes actually outweigh the potential gains in flexibility and modularity. For this reason, MetaOS allows applications to run at the meta-level, directly on top of the Master Meta-Space.

3.1.2 Meta-Spaces

Every MetaOS application is directly supported by one of many meta-spaces, including the Master Meta-Space. Were the application supported by more than one meta-space, questions would arise as to which meta-space's scheduling policy should be used, which paging policy should be used, etc. A meta-space should provide its applications with a suitable run-time environment. If this is not the case, each application is free to attempt to customize the meta-space, or, failing that, to migrate to another, more suitable meta-

space. Note that migration in the context of this dissertation means migration to another meta-space residing on the same machine.

The basic assumption behind meta-spaces is that applications can be grouped with respect to their run-time *requirements*. For example, a video-conferencing application and a video-on-demand application share certain multimedia-inspired requirements, such as timely delivery of video and audio packets, which can be supported by a common multimedia meta-space. An on-line banking system and an airline reservations system, on the other hand, have common requirements in terms of secure transactions, which can be provided by another single meta-space. By removing such common elements from applications, and providing them through independent meta-spaces, the application developer's programming burden can be lightened, which, in turn, can accelerate the time to bring the software to market, improve the maintainability of the code, and reduce costs and numbers of errors. This can also achieved by using dynamically-linked libraries. However, it is quite likely that the requirements of similar applications are not exactly identical. For this reason, meta-spaces, unlike dynamically-linked libraries, provide a degree of flexibility, as will be discussed in Section 3.1.4.

A problem arises when there are competing requirements within an application. For example, a video-on-demand application permits the loss of video frames, to maintain timing constraints. However, the application may also require the transmission of a credit-card number, to pay for the service. This credit card data has no strict timing constraints, but it has very strict requirements regarding security, integrity, and loss. MetaOS supports two alternatives: The application can be split into two parts: One part supports video-on-demand delivery, and runs on the multimedia meta-space; and the

other part supports the sales transaction, and runs on a secure transaction meta-space. The alternative is to load a meta-object implementing secure transactions into the multi-media meta-space. MetaOS does not force either approach on meta-space or application designers.

One of the major differences between MetaOS and Apertos meta-spaces is that the former are concrete, and the latter are abstract. A MetaOS meta-space is an actual object. Each of these meta-space objects executes in its own "hardware-protected" address space. It furthermore has access to the address spaces of its applications; but applications do not have access to their meta-space's address space. An Apertos meta-space, on the other hand, is abstract: Apertos reflectors keep track of a set of meta-objects, thereby defining meta-spaces, but there is no meta-space object as such.

An advantage of our approach is that by implementing meta-spaces as objects, they become natural, memory-protected firewalls. If a meta-object within a meta-space is misconfigured, damage can then be contained to its host meta-space, and any supported application. In Apertos, on the other hand, a misconfigured, shared meta-object can affect a number of meta-spaces.

A possible disadvantage is that one damaged meta-object can affect other meta-objects within the same meta-space; but meta-objects within the same meta-space trust each other to correctly provide the services they advertise. For instance, when a message arrives for a suspended application A, a mailer meta-object depends on the scheduler meta-object to properly schedule A. From this standpoint, the mailer depends on the scheduler semantically, so there is little reason to separate the two by means of strong security like address spaces, capabilities, etc. Nevertheless, the MetaOS prototype less-

ens the security concern by depending on a type-safe language, although other techniques, such as sandboxing [38], can be used as well. MetaOS leaves this implementation decision in the hands of the meta-space designers.

There is also a performance concern: If meta-objects reside in a common address space, communication costs within a meta-space can be drastically reduced. Such communication can be based on method calls, and need not involve context switches. In Apertos, on the other hand, each meta-object is potentially an independently schedulable and directly addressable entity, meaning that any communication within a given meta-space is subject to context switches.

3.1.3 Meta-Objects

Each MetaOS meta-space consists of a set of dynamically-replaceable meta-objects. To ensure that a minimal amount of functionality is provided, this set contains at least a *reflector*, a *mailer*, a *namer*, a *scheduler*, and a *meta-space manager*, by default. Other meta-objects can be added and removed dynamically from this default set with help from the reflector.

The reflector is in charge of accepting all calls to a meta-space's interfaces, and delivering them to the appropriate meta-object via the meta-object's interfaces. If a call cannot be forwarded, the invoker receives an error message. The mailer meta-object is responsible for delivering messages according to a meta-space-specific mailing policy, e.g., delivering messages based on priority. The namer is in charge of meta-space-specific name lookup. The scheduler schedules the meta-space's applications according to

a meta-space-specific scheduling policy, e.g., round-robin. The meta-space manager is responsible for managing meta-space privileges. For example, it has the authority to grant applications new privileges.

Using meta-objects improves the resulting run-time modularity. This is because meta-objects continue to exist as objects at run-time, making them easier to be replaced. Since different parts of a system or meta-space are implemented as meta-objects, it is easier to change them at run-time, only involving the replacement of a meta-object. The trade-off is that the level of indirection adds to the overall cost.

Reflector meta-objects hide the actual contents of the meta-spaces from applications. It is possible to replace certain meta-objects without requiring each affected application to update its meta-object references [53]. The mailer, namer, and scheduler meta-objects will be discussed in more detail in Chapter 5, and the meta-space manager will be discussed in Section 3.2.

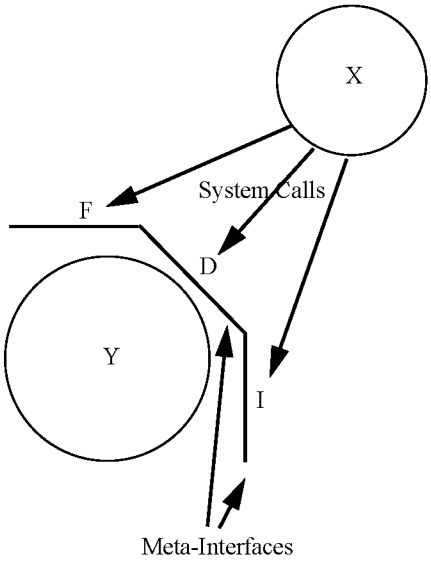
Apertos and MetaOS meta-objects are concrete objects. However, MetaOS meta-objects do not reside in their own, private address spaces. They are not visible outside the meta-space in which they reside. This means that the Master Scheduler only schedules thread objects in meta-level meta-spaces; it does not know that thread objects belong to meta-objects. Indeed, a thread object in MetaOS can actually traverse different meta-objects: It is ultimately the responsibility of a meta-space designer to determine whether a meta-space is single-threaded or multi-threaded. For the sake of simplicity, the remainder of this work will assume single-threaded meta-spaces.

3.1.4 Meta-Interfaces

One of the most important and novel features of MetaOS is its interfaces, as they allow meta-spaces to be customized dynamically in a secure manner. The following introduces MetaOS interfaces; Section 3.2 shows how these interfaces provide secure customizing.

Each meta-space and each meta-object support three types of interfaces --- a *functional interface*, a *declarative interface*, and an *imperative interface* [12,54]. The declarative and imperative interfaces are also collectively known as *meta-interfaces* [55]. This is illustrated in Figures 3-2 (a) and (b). Calls to a meta-space's interfaces are also called *system calls*, and calls to a meta-space's meta-interfaces are also known as *meta-calls*. For example, if an application wants to communicate with its meta-space, it invokes a method in one of its meta-space's interfaces, i.e., the application makes a system call. As outlined in Figure 3-1, the meta-space's reflector then translates the system call to a call on the appropriate meta-object's interface. Similarly, if an application or a meta-level meta-space wishes to communicate with the Master Meta-Space, it invokes one of the Master Meta-Space's meta-space methods, which is then translated by the Master Reflector to a call on the appropriate Master meta-object's interface.

Functional interfaces define *which* services are provided, whereas meta-interfaces determine *how* they are provided. For example, the functional interface may define a method for sending a message to a target that resides on another machine, as shown in Figure 3-3. The meta-interfaces define how this service is provided, including



F = Functional Interface
D = Declarative Interface
I = Imperative Interface

System Calls to D or I are also called *Meta-Calls*

X = Application ->
Y = Meta-Level Meta-Space or
Y = Meta-Meta-Level Meta-Space
X = Meta-Level Meta-Space ->
Y = Meta-Meta-Level Meta-Space

Figure 3-2 (a) Three Types of Meta-Space Interfaces

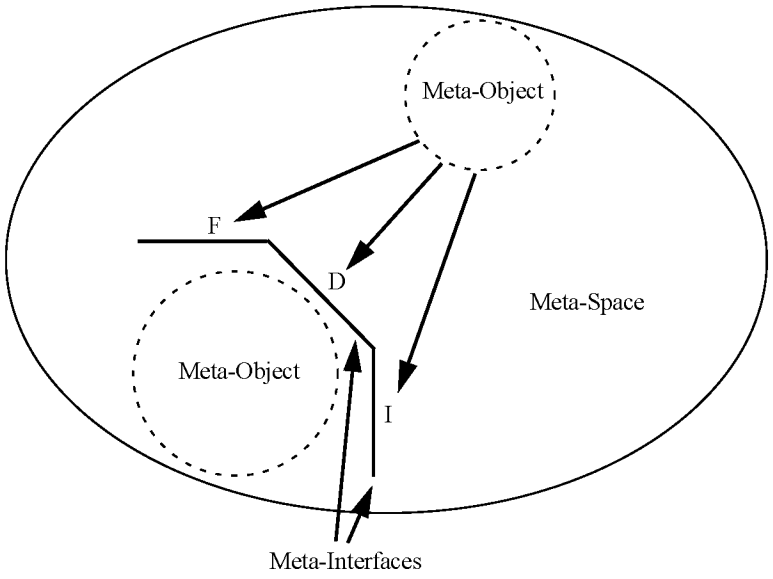


Figure 3-2 (b) Three Types of Meta-Object Interfaces

```
// Functional call to send a message
metaspace.sendMessage(message, target);
                              ...
// Declarative call to query the default protocol
TransportProtocolNamen=metaspace.queryTransportProtocol();
                              ...
// Declarative call to select an existing protocol
metaspace.selectTransportProtocol(UDP);
                              ...
// Imperative call to install a new protocol
TransportProtocol p = new rtpProtocol();
metaspace.installTransportProtocol(p);
```

Figure 3-3 Pseudo-Code for Three Types of Calls

which transport protocol should be used, whether interactions should be logged or replicated, etc.

The declarative interface differs from the imperative interface in that the former provides a list of predetermined policies and allows the status of the current implementation to be queried, whereas the latter allows arbitrary changes to be implemented. Going back to the above example, the declarative interface would provide a list of different transport protocols that the application may use, and allow selection of one of them. The imperative interface, on the other hand, would allow new protocols to be introduced.

The main reason to separate these three types of interfaces is to limit complexity. Most application programmers should be able to implement their applications by looking only at the functional interface. However, if the system does not supply effective services, application programmers would then be able to look at the declarative interface, to select another, existing policy. Only if this failed, would they have to consider the significantly more complex imperative interface. The separation also plays a role in security, and will be discussed next.

3.2 Security Model

MetaOS, like Apertos, is built on the assumption that meta-spaces are used to group common requirements, meaning that a few meta-spaces can support many different applications. To illustrate, a system could provide a multimedia meta-space for different kinds of multimedia applications, a database meta-space for various database applications, and a persistent meta-space for standard office applications, such as word-processors. MetaOS furthermore allows adjustments to be made to each meta-space, subject to the meta-space's security policy, and permits defective meta-objects to be replaced. Most applications should be able to find an existing meta-space suitable to their needs. Only in very exceptional circumstances should there be a need to create a completely new meta-space. Moreover, once applications have customized a meta-space, it is not expected that they will continue to customize the meta-space on a frequent basis. For example, after an application has selected an appropriate scheduling policy for its threads, it is unlikely that it will change this policy again.

3.2.1 Basic Principles

The most basic security principle in MetaOS can be summarized as follows:

> The default secure-customizing policy dictates that unless an application or meta-level meta-space instance has *legitimate authorization*, it should not be able to change its meta-space in a way that harms, or *significantly impacts*, any other application or meta-level meta-space supported by that meta-space.

For example, it should not be possible for an unauthorized application to change the meta-space's overall scheduling policy from a multimedia-friendly policy to a batch policy, since this would likely break other applications. It is therefore very important to protect meta-spaces from unauthorized system calls, as well as from illegal memory accesses. The latter is accomplished by letting meta-spaces reside in their own address space, but securing the interfaces requires more effort.

3.2.1.1 Basic Definitions

Identifiers play an important role in MetaOS. Assume a globally-unique identifier for every user, application instance, and meta-space instance. (While a meta-space consists of several meta-objects, meta-objects themselves do not receive a unique identifier, meaning that a system call made by a meta-object is treated as a system call by the meta-space in which the meta-object resides.) Let U_{ID} be the set of user identifiers, A_{ID} the set of application instance identifiers, and M_{ID} the set of meta-space instance identifiers. The set of identifiers in MetaOS is therefore $U_{ID} \cup A_{ID} \cup M_{ID}$, denoted by ID, where U_{ID}, A_{ID}, and M_{ID} are disjoint sets. If U is a user, then $ID(U) \in ID$ is the unique identifier of U. If A is an application instance, then $ID(A) \in ID$ is the unique identifier of A. If M is a meta-space instance, then $ID(M) \in ID$ is the unique identifier of M. Let I be A or M. Then $ID(I) \in ID$ is the unique identifier of I. Chapter 4 explains how unique identifiers are generated and maintained in MetaOS.

Every meta-object has interfaces defined by sets of methods. A system call $F(a_1,...,a_j)$ is an invocation of a method F with arguments $a_1,...,a_j$ by an application

instance or a meta-space instance. (Technically, a system call is intercepted by the reflector and transparently translated to a call on a meta-object method by the reflector, but since the reflector itself has no role in security, it can be ignored hereafter.) In general, if $F(T_1,...,T_k)$ is a method with parameter types $T_1,...,T_k$, and $D(T_1),...,D(T_k)$ are the value domain of each T_i, then the set of system calls on a method F, denoted by F^*, is

$$\{F(a_1,...,a_k): a_i \in D(T_i),\ 1 \leq i \leq k\};$$

the set of all system calls on a meta-object O, denoted by O^*, is

$$\bigcup_{F \text{ is a method defined in an interface of meta-object } O} F^*;$$

and the complete set of system calls on a meta-space M, denoted by M^*, is

$$\bigcup_{O \text{ is a meta-object in the meta-space } M} O^*.$$

Intuitively, M^* forms the set of all possible system calls that a meta-space can understand. Note that O^* and M^* are not fixed since new methods and meta-objects may be added and removed.

In MetaOS, the smallest *unit of protection* is a system call, meaning that each system call can be secured against unauthorized access. The *access list* of a system call V, $V \in M^*$, is a set of identifiers R_V; $ID \supseteq R_V$; authorized to call V. The set of system calls --- supported by a meta-space M --- which are *accessible* by an invoker with an identifier $id \in ID$, is $C^M_{id} = \{V \in M^*: id \in R_V\}$, and the complete set of system calls which are accessible by an invoker with an identifier id, is $C_{id} = C^{M_1}_{id} \cup ... \cup C^{M_n}_{id}$, where n is the total number of meta-spaces in the MetaOS system. Creating and maintaining access lists on a system call basis does not scale well, so Section 3.2.2 explains how the model

can be implemented realistically, while still remaining faithful to the ideas expressed here.

3.2.1.2 Legitimate Authorization

Saying that I has legitimate authorization to make a call V means that $ID(I) \in R_V$. For example, let I be a multimedia meta-space, M be the Master Meta-Space, and U be the user who instantiated I. Let F be the method *destroyThread(threadID)*, defined in one of M's meta-object interfaces; V_1 be *destroyThread(transactionThread)* and V_2 be *destroyThread(multimediaThread)*; and let R_{V1} contain $ID(U)$, and R_{V2} contain $ID(U)$ and $ID(I)$. The multimedia meta-space can successfully ask the Master Meta-Space to destroy the multimedia thread, but not the transaction thread. The user has the right to destroy both threads.

The above example illustrates that just because U instantiated I, and U is permitted to make a call V, does not mean that I is legitimately authorized to do so. This distinction is important. For example, a user may have just downloaded an application from the internet, and asked a meta-space to execute it. With many systems, this application will then automatically receive the privileges of its creator, meaning that anything the user can do, the application can do as well. This makes the user unnecessarily vulnerable. In MetaOS, it is the right of the user to determine which of the user's rights an application or meta-level meta-space instance will receive. For example, the user could grant the application access to files in one directory only, rather than all of the user's files. More precisely, U can give I either *full rights* or *partial rights*. Saying that I has

U's full rights means that $C_{ID(I)} \supseteq C_{ID(U)}$, that is, I has at least as many rights as U. Saying that I has U's partial rights means that $C_{ID(U)} - C_{ID(I)} \neq \varnothing$, that is, I does not have all the rights U has. Section 3.2.2 shows how access rights can be inherited in a way that does not result in bulky access lists.

The initial assignment of rights depends on an outside policy. For example, in a small research environment, any senior member of a research group may be allowed to customize a meta-space M, meaning that all access lists of M^* would contain the user identifiers of these senior members. In a larger environment, there would be a designated system administrator, meaning that all access lists of M^* would contain the user identifier of the administrator. Acquiring rights, i.e., adding elements to C_{id}, will be discussed later in more detail in Section 3.2.2.

3.2.1.3 Significant Impact

Formalizing the concept of legitimate authorization is fairly straightforward, but formalizing significant impact is very difficult because significant impact is an intuitive notion. For instance, assume I is a video-conference application, $V \in M^*$ is *destroyThread(videoconferenceThread)* and $W \in M^*$ is *selectSchedulingPolicy(Batch)*. Intuitively, I should be allowed to call V but not W, since calling W will impact applications other than I.

Informally, we can say that M guarantees a set of meta-space properties P_I for each application instance I supported by M. This set of properties may contain guarantees such as: The scheduling policy is multimedia-friendly, message transmission is

secure, etc. It is ultimately the responsibility of the meta-space designer to define this set. Calls invoked by I which affect at most P_I, but not the set of properties of any other application instance, are said to have a *local effect*. Calls invoked by I which affect the properties of other application instances as well, are said to have *meta-space-wide effect*, or *significant impact*. Calls with local effect can be made by anyone, i.e., the access list of such calls is considered to be equivalent to *ID*. Calls with significant impact can only be made by I if I has legitimate authorization to do so.

Determining whether or not a call has significant impact revolves around determining the impact a call has on guaranteed properties. Ultimately, it is the responsibility of the meta-space designer to classify calls correctly. If done correctly, a meta-space can be protected. If done incorrectly, a meta-space will have certain security loopholes which could be exploited. Some measure of protection is provided by the fact that each meta-space executes in its own address space, so if a meta-level meta-space, such as the multimedia meta-space, is compromised, this can still allow other meta-level meta-spaces to continue executing unharmed. Section 3.2.3 explains this in more detail.

3.2.2 Applying Basic Principles

The default security policy dictates that applications may not customize a meta-space arbitrarily, to ensure that the meta-space does not fail. In terms of scheduling services, this means that an application A can ask a meta-space to change the scheduling policy of A's threads, but it generally cannot change the scheduling policy of other applications' threads. In terms of communication services, an application can request a meta-space to

encrypt and compress its messages according to some particular algorithm, or even ask the meta-space to log its messages. It generally cannot change these features for other applications; and in terms of naming services, an application generally cannot be allowed to change the way the namer operates. Nevertheless, an administrator must be allowed to replace a defective scheduler, mailer, or namer with an updated version. The administrator must also be able to add previously-unavailable services to a meta-space. It is therefore essential to determine the impact of a call, and protect those with significant impact.

3.2.2.1 Differentiating System Calls

In MetaOS, it is the responsibility of each meta-object to restrict access to all of its methods which can have a significant impact. Thus, for a meta-object designer, the identification of such critical calls becomes very important. For functional calls, this is not as problematic, because they do not generally modify a meta-space service. However, meta-calls are more subtle, since they do not simply *invoke* a meta-space service, but essentially change the *nature* of a meta-space service. This would seem to imply that all meta-calls would have to be off-limits to the average applications; however, this is too strict, since it would disallow applications to make calls which affect only themselves.

In general, identifying harmful meta-calls involves differentiating between local-effect and meta-space-wide effect meta-calls. This is where the strict distinction between declarative and imperative calls is helpful. Declarative calls are predictable, and they therefore provide a basis on which to determine the overall effect of the call. Imperative calls, on the other hand, allow arbitrary changes, and are therefore unpredict-

able. By restricting access to declarative calls with meta-space-wide effect and to all imperative calls, a degree of security can be achieved; by allowing local-effect declarative calls to be freely accessible, a degree of flexibility can also be achieved.

3.2.2.2 Protecting Meta-Interfaces

By default, each meta-object is expected to use access control lists (ACLs) to protect its meta-calls which may have a significant impact. An ACL is a natural application of the access list, or $R_{I,5}$ concept described earlier. To keep the number of ACLs from becoming too large, ACLs are created on a per-method basis, not on a per-call basis. This simplification is based on the observation that a less fine-grained approach is generally sufficient for meta-calls. For example, if an application has the right to change the scheduler to a batch scheduler, it usually also has the right to change it to an interactive scheduler, making it sufficient to check access on the basis of the method which changes the scheduling policy. To simplify the use of ACLs further, granting full access rights to I by user U is a one-step process: Instead of adding I's identifier to all the ACLs in which U is listed, I's meta-space adds U's identifier to a special field in I's context maintained by the meta-space. If this field is not empty, it means that I has the same rights as the user listed in that field, i.e., I can be considered an alias of that user.

To illustrate this, consider the following example: A reflector R has just received a system call from an application instance I. R analyzes this call and translates it to a call to a method F of meta-object O in the same meta-space. R therefore calls F. Assuming F is a method which may have a significant impact, O immediately checks F's

ACL, to see if *ID(I)* is in it. If so, the method invocation proceeds. If not, *O* checks if *I* has any user *U* associated with it, and if so, checks whether *ID(U)* is in the ACL. If this is not the case, the call is rejected. Otherwise, the method invocation goes ahead. To ease the implementation of meta-objects, all meta-object ACLs can be stored in a single structure provided by the meta-space. This structure is local to each meta-space, i.e., meta-spaces do not share ACLs.

ACLs were preferred over capabilities because ACLs make it easy to revoke rights, and prevent rights from being propagated without approval from the ACL manager. Capabilities are much harder to revoke selectively, and it is difficult to prevent their unauthorized propagation. However, ACLs can grow very large. It would be counterproductive to give a large number of users, their applications, and their meta-spaces access to meta-calls with significant impact, since this would sooner or later increase the chances of a security breach. Typically, only very few should get such privilege; hence, large ACLs for meta-calls with significant impact should not be a major problem in MetaOS. Furthermore, in keeping with the MetaOS vision, such major changes are rare, meaning that the ACLs for these calls do not have to be highly optimized. In short, while capabilities can also be used, subject to the above constraints, they are not necessary, since ACLs are sufficient for this task.

3.2.2.3 Protecting Functional Interfaces

The discussion so far concerns the right to call meta-interface methods. The security of functional methods is a different matter. First, methods in functional interfaces are called much more frequently. Second, methods in functional interfaces generally

manipulate data objects, or *service* objects, not meta-objects themselves. For example, the scheduler's functional methods manipulate service objects like thread objects, not the scheduler meta-object; the mailer's functional methods manipulate service objects like message queue objects, not the mailer meta-object; and the namer's functional methods manipulate service objects like name objects, not the namer meta-object. Third, invokers are generally allowed to call significant impact functional methods, depending on which service object is targeted. For example, applications generally have the right to destroy their own threads. Hence they must be allowed to call their meta-space's scheduler method which destroys threads, as long as the target threads belong to them. Thus, for functional methods, meta-object ACLs are generally not associated with the method, but with the target service object. As is the case for meta-calls, this granularity of access is considered to be sufficient. For instance, it is adequate to check access to *threadID* in *setPriority(priority, threadID)*; it is not necessary to tie an ACL to each possible priority. Note that there are some functional calls, such as query calls, which do not have a target service object. Functional query calls are generally not secured, since they do not affect the state of the meta-object.

Ownership of service objects like thread objects are on a per-instance basis. For example, two instances of the same word-processor should not generally have access to each other's thread objects. For this reason it is important to ensure that application identification is done on a per-instance basis.

3.2.2.4 Acquiring Privileges

The previous section mentioned how system calls are protected in MetaOS. This, however, raises the question of how privileges are gained in the first place. At boot-time, the Master Meta-Space checks its policy, and adds the appropriate user identifier or identifiers to the appropriate ACLs. This would include adding the system administrator's identifier to the meta-calls which can have a significant impact. It then starts up a number of meta-level meta-spaces and applications, whose identifiers are also added to ACLs, as dictated by the existing policy. The meta-level meta-spaces check their policies, and add the appropriate user identifier or identifiers to their appropriate ACLs. For example, most meta-spaces would also add the system administrator's identifier to their ACLs. The Master Meta-Space then starts a login application, allowing users to login, following the standard login/password authentication. Once they have successfully logged in, users are then able to launch applications and meta-level meta-spaces.

When launching an application or meta-level meta-space instance *I*, a user *U* can choose to confer full rights to *I*, by asking *I*'s meta-space to associate *ID(U)* with *I*. Then, anything the user can do, *I* can do as well. However, a user can also selectively grant certain rights. The importance of this was discussed in Section 3.2.1.2. *U* does this by using a *checkAuthorization* method to determine which rights *U* has, and by using a *setAuthorization* method to transfer some of those rights to *I*. Chapter 6 discusses the *setAuthorization* and *checkAuthorization* methods.

Privilege assignment does not have to be done manually. Rather, a user or administrator can invoke a script which launches the application, and then assigns the privileges. Scripts require a user's full rights.

An interesting issue arises when new meta-objects are added to an existing meta-space. They, too, have to restrict access to certain meta-calls, but since they were not installed at the time the meta-space was created, they are not aware of any privileged users, i.e., users which are authorized to make any system call supported by the meta-space. This is a case where the meta-space manager comes into play. By querying the meta-space manager for the IDs of privileged users and applications, the new meta-object is able to add them to its own ACLs.

Finally, there is the issue of transitivity of trust. Specifically, if a meta-space *M* trusts *A*, and *A* trusts *B*, can *M* trust *B*? In MetaOS, *M* trusts *B* if *A* tells *M* to trust *B*. Therefore, the trust relation is not transitive in MetaOS. Consider: We may be willing to give a good friend the key to our house, but would be alarmed to find a stranger in the house with our key. Assurances by the stranger that the key was given in trust by our friend will not be very reassuring. However, if the trusted friend had notified us of this beforehand, possibly introducing us to the stranger, this would be a different matter. Note that the MetaOS trust relation is also not symmetric, i.e., *A* does not necessarily trust *M*.

3.2.2.5 Cloning Meta-Spaces

Meta-spaces can become outdated. For instance, a new transport protocol meta-object could have been developed after a meta-space was shipped. MetaOS allows this meta-object to be installed dynamically by a legitimately-authorized application or meta-level meta-space instance *I*. Furthermore, to ensure that other applications or meta-level meta-spaces can benefit from this change as well, the new protocol can be made avail-

able from the declarative meta-space interface as a local-effect call. To illustrate, *I* may just have installed XTP [56]. By providing a method that selects between the original transport protocol, such as TCP, and the new XTP protocol, and by exporting that method to the declarative meta-space interface as a local-effect call, *I* can let others benefit from the new protocol as well. Determining which parts of the new meta-object should be accessible from the declarative interface is the responsibility of the meta-object designer.

Meta-spaces can allow unauthorized applications or meta-level meta-spaces to request installation of certain meta-objects. For instance, a system administrator may have placed a large number of transport protocols which do not violate the security policy into a trusted and protected directory. Unauthorized applications can then ask their meta-space to install these protocols as needed, and make their selection available as local-effect declarative calls. Since these protocols were selected based on their adherence to the security policy, installing such a protocol on behalf of an unauthorized application is not a security risk.

If this is still insufficient, MetaOS allows any meta-level meta-space to clone itself. This clone is a copy of the meta-space at the time cloning was requested, including any customization introduced after the creation of the meta-space, but without data belonging to other applications. This means that if any application *A* requests cloning, the clone will not contain data such as thread objects or service objects belonging to other applications. *A* can then migrate to this clone, where the cloned meta-space will give *A* full access. For example, if researchers of the same laboratory wish to experiment with new resource management policies, they can clone an existing meta-space,

migrate their sample applications to it, and then customize the clone as they see fit, without endangering applications running on the existing meta-space. This is because the clone will execute in its own address space, and will not have any rights which can significantly impact other meta-level meta-spaces.

3.2.2.6 Migrating Applications

An application may migrate to another meta-space that best suits it, assuming the target meta-space permits this. Migration usually occurs immediately after an application starts up and realizes that its current meta-space is insufficient.

Allowing applications to migrate among meta-spaces raises several questions: For one, if an application *A* wants to migrate to another meta-space *M*, how does it know *M* is dependable? If *M* was created as part of *A*'s clone request, this is not a major concern, since *M* is basically a copy of *A*'s existing meta-space. If the target meta-space is part of the default MetaOS system, this is also not a major concern. For other meta-spaces, however, *A* can ask the Master Namer to discover who instantiated *M*, and then make a decision based on the level of trust placed in *M*'s instantiator. While this is not a perfect guarantee, knowing the instantiator of a meta-space serves as a reasonable basis for dependability.

The second question an application faces is whether or not the target meta-space is compatible. This is where the declarative interfaces become important, because they allow status information to be obtained. An application can, for example, query what kind of scheduler is being used, who wrote it, and what version number it is. Based on this information, the application can know whether or not the meta-space is compatible.

Note that MetaOS requires all meta-objects to implement certain standard interfaces, to allow the query itself to succeed. Chapter 6 will focus on these standard interfaces in more detail.

The third question relates to privileged access: Just because an application had the right to change one meta-space does not automatically give it the same rights on another meta-space. Thus, in cases where the new target meta-space is the most suitable of the existing meta-spaces, but if the local-effect customization efforts are insufficient, an application must clone the most suitable meta-space and customize it as required. It must be stressed, however, that the MetaOS vision believes that most applications will find a suitable, existing meta-space.

A different issue involves maintaining access rights to service objects. By default, applications keep the rights to the objects they own. For example, in the prototype discussed in the Chapter 6, when an application migrates, it is migrated along with the service objects it owns: its thread objects, its mail queue objects, and its name objects.

3.2.2.7 Removing Customizations

The life-time of customized code is another concern. In many cases, the customized service itself only lasts as long as the instance *I* that requested it. For example, if an application terminates, or leaves a meta-space by migrating to another meta-space, any changes affecting only it, such as its thread scheduling policy, are removed. This is accomplished by tying such local-effect changes to the application's context. In cases of meta-space upgrades, the changes must be permanent, i.e., they should exist as long

as the meta-space, or until another upgrade is introduced. This can be accomplished by tying such changes to the meta-space itself. Finally, there are customizations which are only supposed to be used by a select group of instances $I_i,...,I_j$, and are to be removed when all these instances have terminated or left the meta-space. Such customizations can be tied to the contexts of these instances, so that when the last one terminates or leaves, the customization is removed as well.

An instance *I*'s access rights are also removed automatically when *I* terminates or leaves a meta-space. Removing service object ACLs is straightforward, since these ACLs are tied to service objects, which in turn are tied to *I*'s context. Removing full authorization is also simple, since full authorization is tied to the user field in *I*'s context. Partial authorization is more difficult to remove, since it involves removal of *ID(I)* from individual ACLs. This removal is accomplished by using a *removeAuthorization* method, discussed in Chapter 6. The removal procedure can be simplified by letting the meta-space manager keep track of the ACLs to which *ID(I)* was added, or by letting the script which added partial rights for *I*, remove these rights when they are no longer needed.

3.2.3 Customizing Security

A typical application cannot arbitrarily change the underlying security policy or mechanism of a meta-space, thereby giving it new --- and unauthorized --- privileges; but it may add a meta-space to a running system which implements a completely different security policy and mechanism, and migrate to it. Replacing an existing security policy

or mechanism with an updated version can also be done, although this right is generally limited to an administrator.

There are two main reasons why a failure in one meta-level meta-space does not automatically spread to other meta-level meta-spaces. First, meta-spaces execute in their own address space. Second, meta-space methods which could have significant impact are secured against unauthorized accesses. Because of this, it is possible to have customizable security policies and mechanisms for different meta-level meta-spaces, and still have security.

3.2.3.1 Customizing Policy

It is possible to create new security policies in MetaOS that co-exist with existing security policies. For example, consider a policy based on mutual trust of selected applications and their users. This policy can be realized by creating a new meta-level meta-space *M'* which restricts admission to *M'* to these users and their selected applications, and which gives legitimate authorization to any user or application which is admitted to *M'*. Once admitted, no more checks for full or partial rights are required, since all admitted users and applications trust each other. This may be useful in a research environment where users trust each other, and do not like the strict default environment. At the same time, however, applications which remain with other meta-spaces will not be affected by this.

There may also be some cases where an existing policy is altered to reflect new conditions; for example, a company may just have changed their policy, and require two system administrators to independently approve changes to the system. This would

require changes to the policy, and is similar to re-keying locks, or installing additional locks. As such, these most radical types of changes may require changes to the applications. Implementing these changes would also require the replacement of existing meta-objects. For example, significant impact calls could be secured against a second ACL, representing the second administrator, meaning that whenever such a call is made, the meta-object involved would have to check if both system administrators gave the application instance the appropriate rights.

3.2.3.2 Customizing Mechanism

It is also possible to create new security mechanisms in MetaOS that co-exist with existing security mechanisms. For example, instead of relying on application identifiers and user identifiers, a meta-level meta-space could instead require the use of capabilities. Specifically, a new meta-space would have to be created which understands capabilities, rather than ACLs. This meta-space would issue capabilities for calls, based on a one-time password challenge. If the challenge is met, i.e., the password is correct, the meta-space would hand the requestor a capability. Subsequently, whenever a significant impact call was made, this capability would have to be presented. Unlike ACLs, this capability could also be passed freely from application to application. Because this scheme would be provided by a meta-level meta-space, other meta-level meta-spaces could keep the default ACL approach. Furthermore, any fault with this new scheme would be confined to that particular meta-level meta-space. Note that it would not be possible for an application *depending* on the default scheme to migrate to this new meta-

space, and continue to function as it had before. Determining compatibility was discussed in Section 3.2.2.6.

Similarly, assume a meta-object M has a defective implementation of an ACL. For example, even though the contents of the ACLs may be correct, their lookup may not always succeed, meaning access is not always given when it should. Since MetaOS allows meta-objects to be replaced dynamically, a new meta-object M', which fixes M's defect, can be used to replace M at run-time.

To ensure that the state of ACLs is maintained, MetaOS requires that meta-objects keep ACL information in a generic state object. Individual types of meta-objects can then subclass these generic objects to account for additional state, e.g., the namer would include the name table. When meta-objects are replaced, the state remains in the meta-space, and can be taken up by the replacement. This will not work if meta-objects cannot understand each other's state objects, so as part of the installation process, the installer has to make sure that the new meta-object is compatible with the existing meta-object. This is no different from upgrading existing systems: When one installs new software, one must ensure that one is using the correct version of the operating system. Similarly, when installing a new meta-object, one must ensure that the old and the new meta-objects are compatible. Version information can be obtained from the declarative interfaces, as discussed in Section 3.2.2.6.

Chapter 4

MetaOS Security Analysis

4.1 Master Meta-Space Security

To show that MetaOS is secure, it is crucial to demonstrate that the Master Meta-Space is secure. The following analysis will deal with illegal memory accesses, restricted system calls, and naming issues. The security of meta-level meta-spaces is considered in Section 4.2. Note that this analysis assumes that the model is correctly implemented, meaning that calls have been differentiated according to the policy of Section 3.2.1, and secured according to Section 3.2.2.

4.1.1 Illegal Memory Access

One of the first concerns is that the Master Meta-Space could be subject to illegal memory accesses by applications or meta-level meta-spaces, i.e., *invokers*. However, this concern is addressed by the fact that the Master Meta-Space resides in its own, hardware-protected address space, namely the kernel address space. The only access to that address space is by a trap into the Master Reflector.

4.1.2 Restricted System Calls

After every trap, the Master Reflector proceeds to analyze the call. If it understands the call, the Master Reflector calls the appropriate meta-object method. If it does not understand the call, the call is rejected, i.e., the invoker receives an error message. As mentioned in Section 3.2, there are calls which can significantly impact others, and calls which do not. According to the default policy, the former are restricted, whereas the latter can be freely invoked. Therefore, a concern is that a call is an unauthorized call to a restricted method. However, the MetaOS model demands that all such calls are first checked against an ACL before the call is allowed to proceed, and can thus prevent unauthorized access, subject to the constraints given in Section 4.1.3.

4.1.3 Naming

ACLs depend on secure naming, to prevent an invoker from falsely identifying itself. This raises several issues which must be addressed by the naming scheme. First of all, identifiers must be globally unique, to prevent an invoker *I* without proper privilege from accidentally having the same identifier as an invoker *I'* with proper privilege. MetaOS guarantees that the Master Namer supplies only globally-unique application (and meta-space) identifiers, based on the globally-unique machine identifier, and a machine-unique application (or meta-space) instance identifier. This way, even instances of the same application (or meta-space) will each receive their own, globally-unique identifier. The Master Meta-Space then adds this identifier to the context information which the Master Meta-Space maintains about the application (or meta-space), meaning that this

identifier will be associated with the application (or meta-space) for the rest of its run-time life-time.

Another concern is that the Namer will eventually run out of identifiers, forcing it to recycle them. Recycling of instance identifiers is surprisingly simple: Section 3.2.2.7 stated that a meta-space's ACLs are cleared of instance *I*'s identifier as soon as *I* leaves the meta-space or terminates. Furthermore, the MetaOS naming scheme prohibits its meta-spaces from assuming that instance identifiers will persist after the associated application or meta-space terminates, meaning that meta-spaces do not save the content of their ACLs. This means that the Master Namer only has to pick an identifier which is not used by a currently-running application or meta-space. Since the Master Namer maintains the identifiers for each application and meta-space instance, it only has to pick an identifier not currently in its identifier table.

Like application identifiers, user identifiers are globally unique, and are maintained by the Master Meta-Space. However, unlike application identifiers, meta-spaces may assume that user identifiers persist, even after a user has logged off. Therefore, if a meta-space adds a user identifier other than that of the super-user to its ACLs, it must give the Master Namer a one-time notice for each user. The Master Namer subsequently stores this in its database, where it is kept until the meta-space notifies the Master Namer that it no longer supports the user, or the meta-space is deleted.

When a user's system access privileges are removed, the Master Namer checks whether any meta-spaces have given access to the deleted user. If so, and if the meta-space is currently running, the Master Namer notifies the meta-space, which then deletes the user identifier from its ACLs. Note that usually, there is always at least one identifier

which remains in the ACL --- that of the super-user. For those meta-spaces which are not currently running, the Master Namer will notify them as soon as the first instance is started. It is then up to the first instance to update its policy to reflect the departed user. After that, the identifier can be re-used.

A problem with this approach is that a meta-space may never be started again because it was deleted, meaning that the Master Meta-Space ultimately has to monitor the file system. If a meta-space is removed, and if it registered a user identifier with the Master Namer, the registration is purged.

Another problem relates to attacks on the Master Namer. Since identifiers are kept in the Master Meta-Space, and are associated internally with an invoker, an invoker could try to alter an identifier by illegal memory access, but Section 4.1.1 discussed how this is prevented. The next concern is that an invoker could illegally try to call the Master Meta-Space's methods to change the naming policy, assign itself a different identifier, or add itself to the ACL of a method which gives out new privileges. However, Section 4.1.2 noted that such critical system calls are restricted by an ACL check. To pass this check, an invoker would have to spoof its identifier first; but this means that an invoker would first have to try to spoof its identifier before it could spoof its identifier, a contradiction. Hence this concern does not hold, either.

Another concern relates to assigning the wrong user privileges at login time. For this reason, MetaOS requires users to authenticate themselves before logging onto the system. This follows the standard login/password protocol, meaning that security, like the security of most operating systems, depends on making the login/password pair hard to guess. Like other authentication schemes, this is not designed to be 100% secure, but

is designed to make false authentication unlikely. Passwords, like identifiers, are protected in the Master Meta-Space.

4.2 Meta-Level Meta-Space Security

In addition to Master Meta-Space security, one must also consider the security of meta-level meta-spaces. Similar to the Master Meta-Space analysis, the following will discuss illegal memory accesses, restricted system calls, and naming issues.

4.2.1 Illegal Memory Access

One of the first concerns is that a meta-level meta-space could be subject to illegal memory accesses by applications or other meta-level meta-spaces. However, this concern is addressed by the fact that the code and data of a meta-level meta-space reside in their own, hardware-protected address space, inaccessible to any other application or meta-level meta-space. The only way to access this meta-space is by means of the Master Meta-Space. Thus, to access a meta-level meta-space *M*, the invoker has to send this request to the Master Meta-Space. The Master Meta-Space then schedules *M*, and places the request into *M*'s incoming message queue, along with a copy of the invoker's identifier. This queue will then be read by *M*'s reflector.

4.2.2 Restricted System Calls

M's reflector analyzes all incoming calls. If it understands the call, the reflector calls the appropriate meta-object method. If it does not understand the call, the call is rejected,

i.e., the invoker receives an error message. If the call came from an application which is supported by *M*, *M* places this message directly into that application's incoming message queue. Otherwise, *M* asks the Master Meta-Space to return the error message to the original invoker.

As is true for the Master Meta-Space, there are calls which can significantly impact other applications, and calls which are not. According to the default policy, the former are restricted, whereas the latter can be freely invoked. Therefore, a concern is that the incoming call is an unauthorized call to a restricted method. However, the MetaOS model demands that all such calls are first checked against an ACL before the call is allowed to proceed, and unauthorized access can thus be prevented, subject to the constraints listed in Section 4.2.3.

4.2.3 Naming

As mentioned in Section 4.1.3, ACLs depend on secure naming to prevent an invoker from falsely identifying itself. In MetaOS, all meta-level meta-spaces use the identifier provided by the Master Namer to identify the invoker. This identifier is kept internally, i.e., namers may provide different naming schemes to their applications, to support location transparency, but internally, these names are mapped to Master Namer identifiers. The security of Master Namer identifiers was already addressed in Section 4.1.3.

A concern is that the copies of Master Namer identifiers kept inside meta-level meta-spaces are subject to attack. For one, an invoker could try to alter an identifier by illegal memory access, but Section 4.2.1 showed how this is prevented. The next con-

cern is that an unauthorized invoker could try to call the meta-level meta-space's methods to change the naming policy, assign itself a different identifier, or add itself to an ACL of a method which gives out new privileges. However, Section 4.2.2 noted that all such critical system calls are restricted by ACL check. To pass this ACL check, the invoker would have to spoof its identifier first. However, Section 4.1.3 stated that Master identifiers, the identifiers also used by meta-level meta-spaces, cannot be spoofed.

4.3 Summary

This chapter showed how a meta-space can be secured against unauthorized access, by placing it into a separate address space, and by securing its significant impact methods. However, it must be stressed that an *authorized* application or meta-space has a position of trust, meaning that it must ensure that its changes will not break an existing meta-space. Thus, ultimately, there must be an authorized user who knows what changes are being made, and knows what the consequences will be. This is something that cannot be checked automatically. Should an authorized user fail to do this, the meta-space can crash. On the other hand, if this failure happens at the meta-level, other meta-level meta-spaces and their applications can continue to execute normally, as was outlined in Section 3.2.3.

Chapter 5

MetaOS Services

5.1 Overall Control Flow

The generic MetaOS model discussed in Chapter 3 is best illustrated by showing how this model can be used to implement actual operating system services, such as scheduling, inter-process communication, and naming. The overall flow of control that takes place when invoking these services turns out to be similar: If an application *A* makes a system call, the call is transferred to the Master Meta-Space. This call is synchronous, i.e., *A* does not regain control until the application's meta-space, *M*, has processed the request. After the Master Meta-Space receives the system call, the Master invokes *M*, providing it with the necessary information, such as the identifier of the application which made the system call, which call was made, etc.

One reason why the Master Meta-Space becomes involved is because *A* and *M* reside in different protection domains, meaning that going from one to the other involves manipulation of page tables. For the sake of security, this is handled by the Master Meta-Space. The Master Meta-Space furthermore becomes involved because it is in charge of scheduling the threads of the appropriate meta-level meta-space. For example, if application *A* wants to request a new scheduling policy, *A*'s meta-space *M* knows how to do this. However, in order for *M* to execute, it must be scheduled, which is handled by the Master Meta-Space. This approach would not have been necessary if *A* and *M*

resided in the same protection domain and shared the same thread of control. However, to ensure that MetaOS remains close to the original Apertos model, and to address security concerns, such as arbitrary memory accesses by *A* to *M*, this approach was not taken.

5.2 Scheduling

By default, each application and each meta-space has its own thread. Application threads are scheduled by meta-level meta-spaces, which maintain some context information for their applications, such as priorities. Other application context information, such as page tables and the program counter, is maintained by the Master Meta-Space. Meta-level meta-space threads are scheduled by the Master Meta-Space, which maintains all relevant context information for these meta-spaces. The Master Meta-Space is scheduled by external events: If an interrupt comes in, e.g., a timer interrupt or a user interrupt, it wakes up the Master Meta-Space thread. The thread then services the interrupt, and suspends itself again, pending arrival of a new interrupt.

5.2.1 Preemption

MetaOS scheduling is best described by illustrating how applications and meta-spaces are preempted. Preemption of an application is illustrated in Figures 5-1 (a) and (b): Part (a) describes this event in general terms of applications and meta-spaces, and Part (b) describes it in detailed terms of control flow among meta-objects. In these Figures, a timer interrupt is sent by an interrupt handler to the Master Reflector (1). Note that the timer is initialized and controlled by the Master Meta-Space. Realizing that a timer

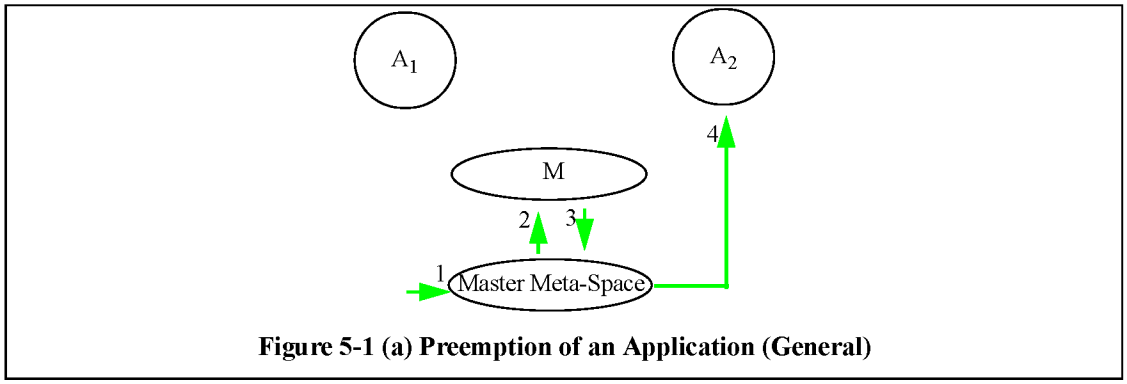


Figure 5-1 (a) Preemption of an Application (General)

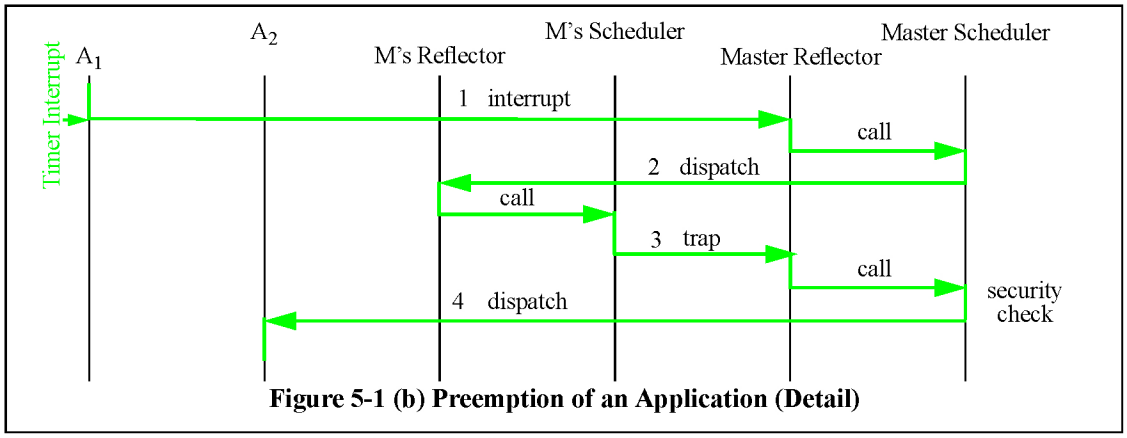


Figure 5-1 (b) Preemption of an Application (Detail)

interrupt has occurred, the Master Reflector notifies the Master Scheduler. The Master Scheduler, seeing that a timer interrupt has occurred, then determines whether or not to preempt the current meta-level meta-space, *M*. In particular, it determines if *M* still has time left in its time slice *S*, where *S* is a multiple of the timer's granularity. For example, if the timer granularity is set to 1 ms, *S* would be *n* ms, *n* being an integer > 0.

Assuming that *M* still has some time left in its time-slice, the Master Scheduler notifies *M*'s reflector that a timer interrupt has occurred (2). Realizing that this is an event for the *M*'s scheduler, *M*'s reflector forwards the event to that scheduler. The scheduler can then give CPU time to another application supported by *M*, such as A_2. However, context switching to A_2 cannot be done by the meta-level meta-space since it involves the manipulation of page tables. As mentioned earlier, only the Master Meta-

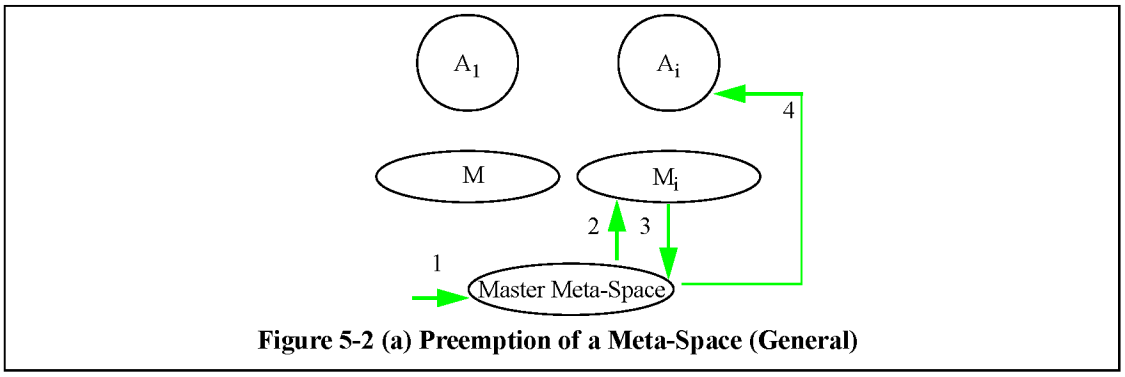


Figure 5-2 (a) Preemption of a Meta-Space (General)

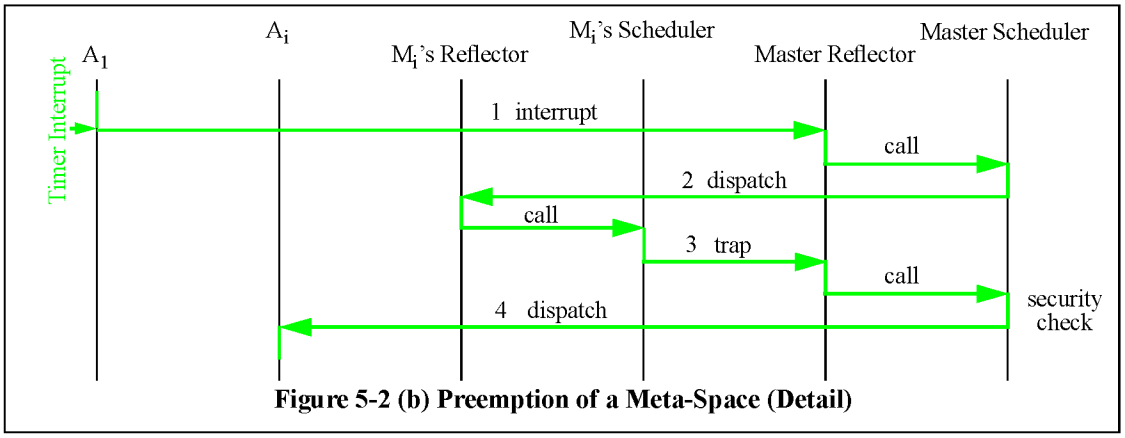


Figure 5-2 (b) Preemption of a Meta-Space (Detail)

Space can do this. Hence the meta-level meta-space notifies the Master Meta-Space that it would like to schedule A_2. This results in a trap to the Master Reflector (3), which then delegates the request to the Master Scheduler by calling its scheduling method. Since this method can have significant impact, the Master Scheduler checks to see if the meta-level meta-space is allowed to schedule A_2. If it is, the Master Scheduler dispatches A_2 (4).

Had the meta-space's time slice expired, the Master Scheduler would have saved the current meta-space's state, and notified another meta-space that it now had control of the CPU (2), as illustrated in Figures 5-2 (a) and (b). That meta-space's scheduler can then give CPU time to one of its applications. As was the case in Figure 5-1, this involves notifying the Master Scheduler (3). The Master Scheduler checks to see if the

meta-level meta-space is allowed to schedule the application, and if so, dispatches the designated application (4).

5.2.2 Customizing

Figures 5-3 (a) and (b) illustrate what happens when scheduling is customized. In this figure, *A* notices that the overall scheduling policy is inadequate, and asks *M* to change it. As before, this involves a trap into the Master Meta-Space (1), which notices that *A* is trying to contact its meta-space. The Master Reflector therefore calls the Master Mailer to place *A*'s request into *M*'s incoming message buffer, and then calls the Master Scheduler to dispatch *M* (2). *M*'s reflector, noticing that the call is a scheduler call, notifies the scheduler. Since changing the overall scheduling policy is of significant impact, the meta-level scheduler checks to see if *A* has the right to request this change. If it does, the scheduler changes the scheduling policy.

By guaranteeing time slice durations, the Master Scheduler enables meta-level meta-spaces to employ real-time schedulers. These real-time schedulers are then responsible for using the allotted time slice according to their particular policies, and usually demands the inclusion of QoS policies which require newly-launched applications to state their CPU demands, and which turn away applications whose CPU demands cannot be met.

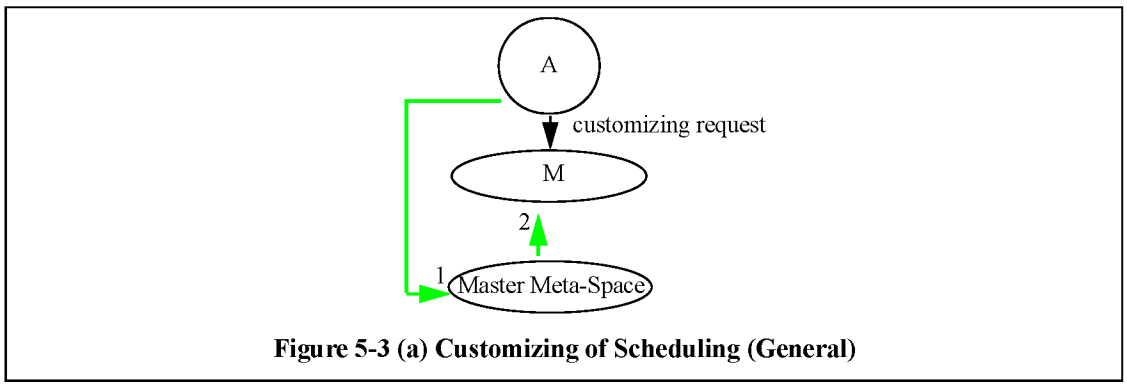


Figure 5-3 (a) Customizing of Scheduling (General)

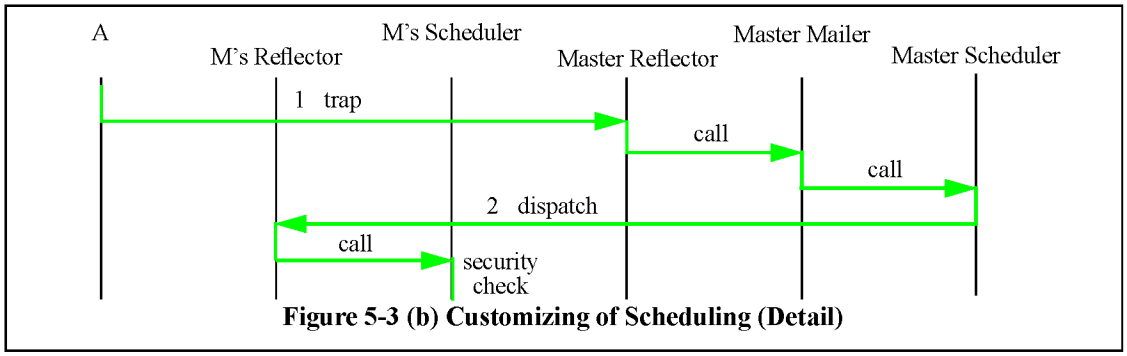


Figure 5-3 (b) Customizing of Scheduling (Detail)

5.3 Inter-Process Communication

By default, communication between applications running on the same meta-space is mainly handled by the applications' meta-level meta-space, and communication between meta-level meta-spaces is handled by the Master Meta-Space. Communication between applications running on different meta-space involves both applications' meta-spaces, as well as the Master Meta-Space.

5.3.1 Applications on the Same Meta-Space

Figures 5-4 (a) and (b) illustrate what happens when two applications, both residing on the same meta-space *M*, communicate in MetaOS. As in previous figures, (a) is general, and (b) is a more detailed depiction of the same event. From A_1's perspective, it directly sends a message to A_2. In reality, however, A_1 makes a system call that traps into the

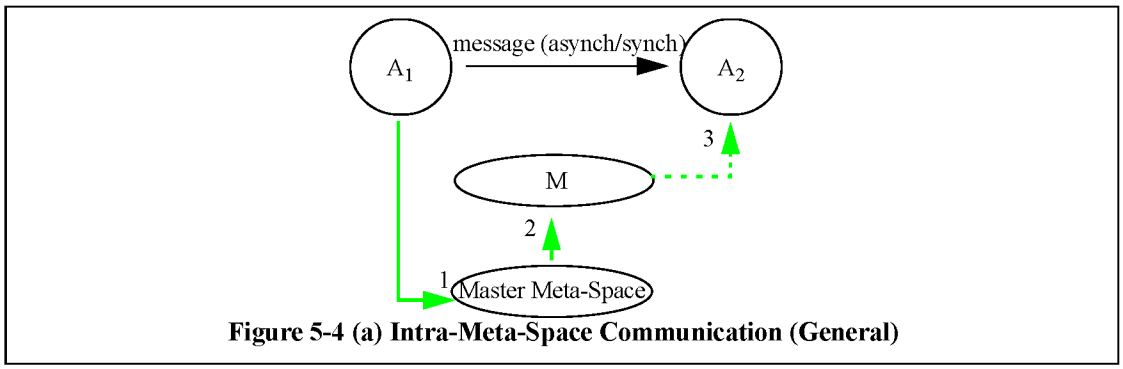


Figure 5-4 (a) Intra-Meta-Space Communication (General)

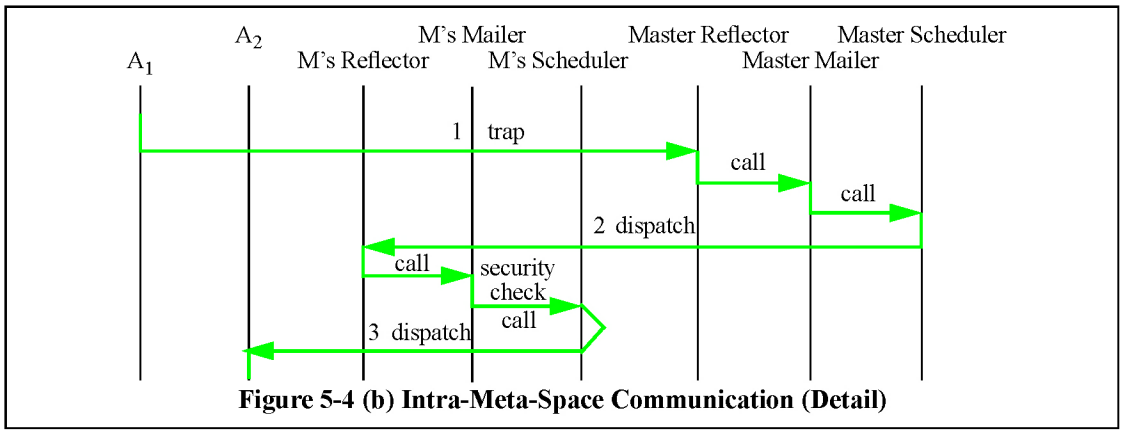


Figure 5-4 (b) Intra-Meta-Space Communication (Detail)

Master Meta-Space (1). This holds true whether or not messages are being sent synchronously or asynchronously. The Master Reflector, realizing that this request should be handled by *M*, calls the appropriate Master Mailer's method to place information about the system call into *M*'s incoming message queue. This queue is a priority queue maintained by the Master Mailer in the kernel address space. The contents are copied to the meta-level meta-space by request.

Subsequently, the Master Scheduler is called to schedule *M*. When *M* starts to execute (2), its reflector checks for incoming messages. Seeing that A_1's system call is a communication request, the reflector invokes *M*'s mailer to process the request. Depending on the meta-space's security policy, the mailer may check to see if A_1 is allowed to communicate with A_2. If it is, the mailer puts A_1's message into A_2's incom-

ing message queue. This message queue is a priority queue maintained by M. The contents are copied to the application by request.

Now the scheduler has to be invoked to schedule A_2. It is only at this point that there is a difference between synchronous and asynchronous communication. With synchronous communication, A_1 will not execute again until A_2 replies to the message. With asynchronous communication, there is no such constraint. In either case, because trapping into the reflector is always synchronous, A_1 definitely cannot execute until M has processed the request. Dispatching of A_1 and A_2 involves the help of the Master Scheduler, as was the case in Section 5.2.1, and is not illustrated for simplicity's sake.

The control flow discussed in this section may appear to be laborious, but it should be noted that it involves mostly function calls among meta-objects in the same address space. Specifically, after A_1 traps into the Master Meta-Space, communication among the Master Meta-Space's meta-objects involves only function calls. Then, to execute M, the first context switch occurs. Subsequent communication among M's meta-objects are again only function calls. The second context switch then occurs when M's next-scheduled application starts to execute.

5.3.2 Applications on Different Meta-Spaces

Communication between two applications on different meta-spaces is illustrated in Figures 5-5 (a) and (b). Here, if A_1 wants to send a message to A_i, A_1 must invoke its meta-space's inter-process communication method. By default, MetaOS inter-process communication does not hide the fact that the target is on a different meta-space, although

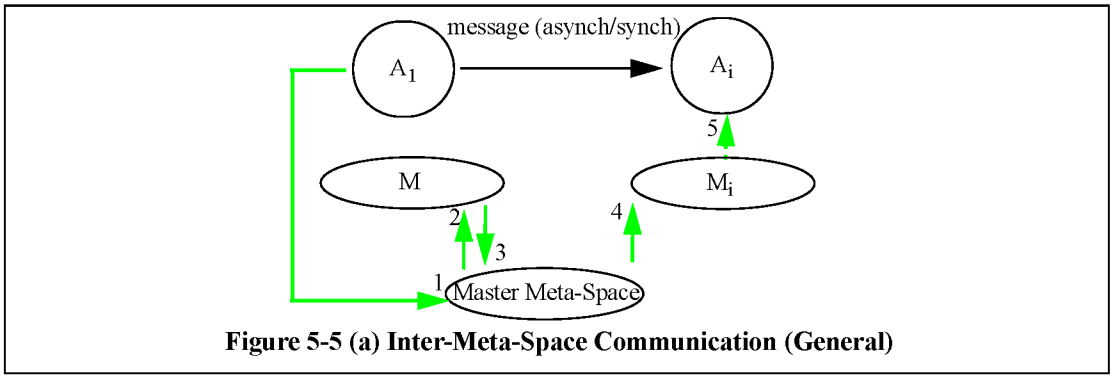


Figure 5-5 (a) Inter-Meta-Space Communication (General)

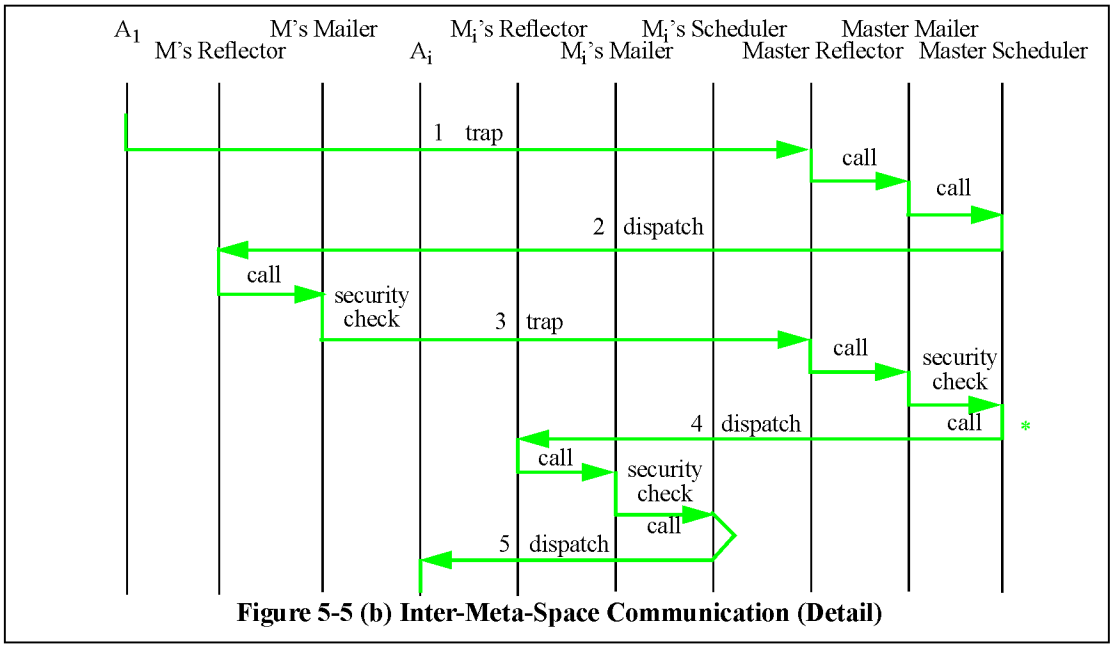


Figure 5-5 (b) Inter-Meta-Space Communication (Detail)

MetaOS does allow meta-space designers to design meta-spaces which make inter-process communication location-transparent.

As before, the system call results in a trap to the Master Meta-Space (1), causing the Master Meta-Space to deliver A_1's request to A_1's meta-space, M (2). Notification of M is important, because M may implement meta-space-specific policies, such as security policies which restrict the application to communicate only with applications on the same meta-space. Assuming that only certain applications are allowed to communicate with applications on other meta-spaces, M's mailer will perform a security check as soon as it realizes that the target resides on a different meta-space. Only if the application is

allowed to engage in inter-meta-space communication will the mailer ask the Master Meta-Space to send the message to the appropriate meta-space for further processing (3). M's mailer may also perform some marshalling and encryption.

Note that A_i's meta-space, M_i, does not always receive immediate control. Depending on the policy, control could be returned to M at the point indicated by the asterisk in (b). M_i (4) would then receive the message when its time-slice starts, assuming that the Master's security policy allows M and M_i to communicate. Depending on M_i's policy, M_i may also check to see if A_i is allowed to receive a message from outside of M_i, and only if this is the case, would A_i actually receive the message (5).

5.3.3 Meta-Spaces on the Master Meta-Space

Figures 5-5 (a) and (b) implicitly illustrate communication among meta-level meta-spaces. If M wants to communicate with M_i --- which it does in Figure 5-5, because it has to deliver A_1's message to M_i --- it must make a system call to the Master Meta-Space. This request will then be received by the Master Reflector, and be delegated to the Master Mailer. If the security policy permits it, the Master Mailer will then place the necessary information into M_i's incoming message queue. The Master Scheduler will also be involved, to make sure that M_i is scheduled for execution.

5.3.4 Customizing

Figures 5-6 (a) and (b) illustrate what happens when communication is customized. In this figure, *A* would like to initialize the way its messages are encrypted. As before, this involves a trap into the Master Meta-Space (1), which then dispatches *M* (2). *M*'s reflector, noticing that the call is a mailer call, notifies the meta-level mailer. Since changing the encryption policy is of local effect only, no security check is necessary at this point. However, had a significant impact call been made, the meta-level mailer would have checked to see if *A* has the right to request this change.

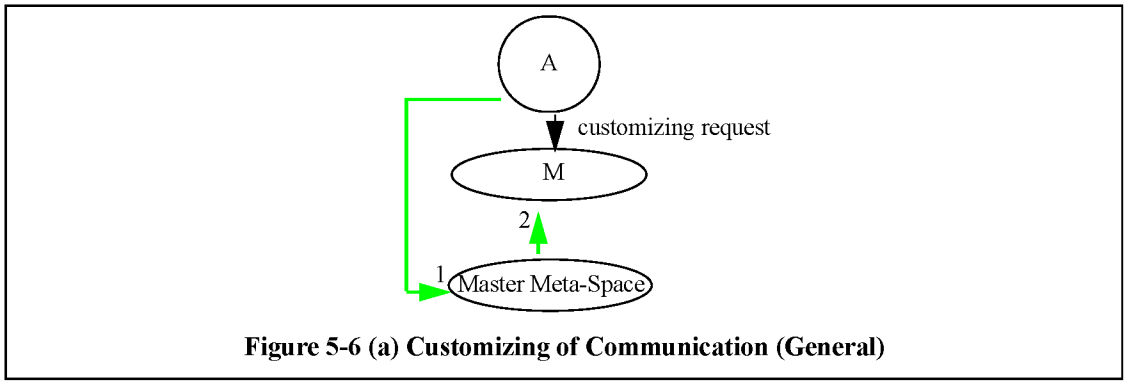


Figure 5-6 (a) Customizing of Communication (General)

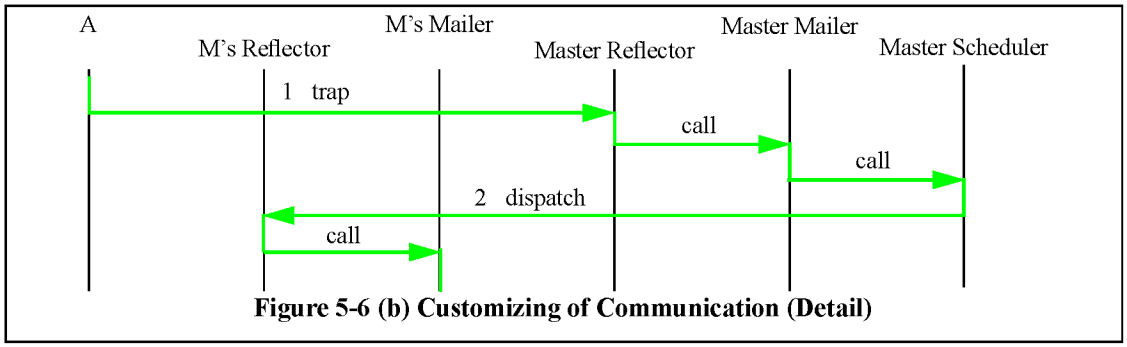


Figure 5-6 (b) Customizing of Communication (Detail)

5.4 Naming

Meta-level meta-space namers keep a list of the identifiers of the applications *they* support. Meta-level namers may also maintain a list of the applications' attributes, to allow lookup by attribute, rather than by identifier. For example, a client application may want to find the identifier of a server application with a particular version number. As discussed in Chapter 4, the Master Namer keeps track of the identifiers of *all* applications and meta-spaces running on the local machine. The Master Namer is furthermore in charge of assigning and maintaining globally-unique identifiers for applications and meta-spaces.

Typical identifier lookup is illustrated in Figures 5-7 (a) and (b). Here, an application *A* wants to retrieve the identifier of another application with a specific version

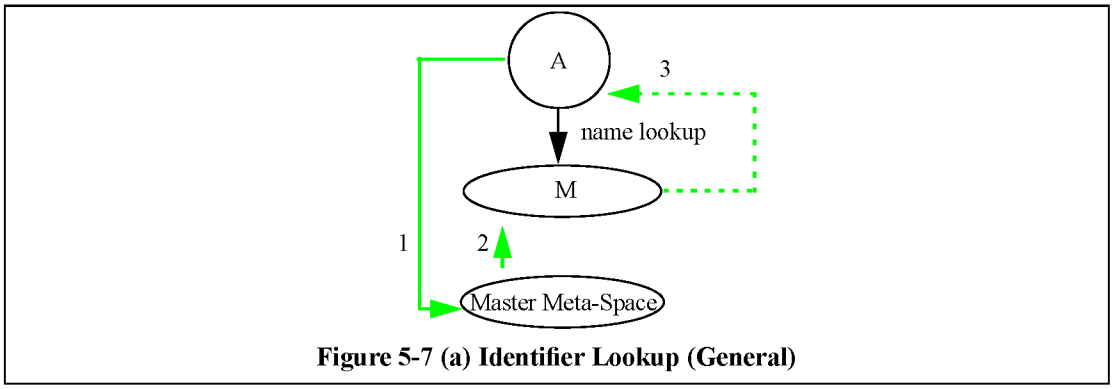


Figure 5-7 (a) Identifier Lookup (General)

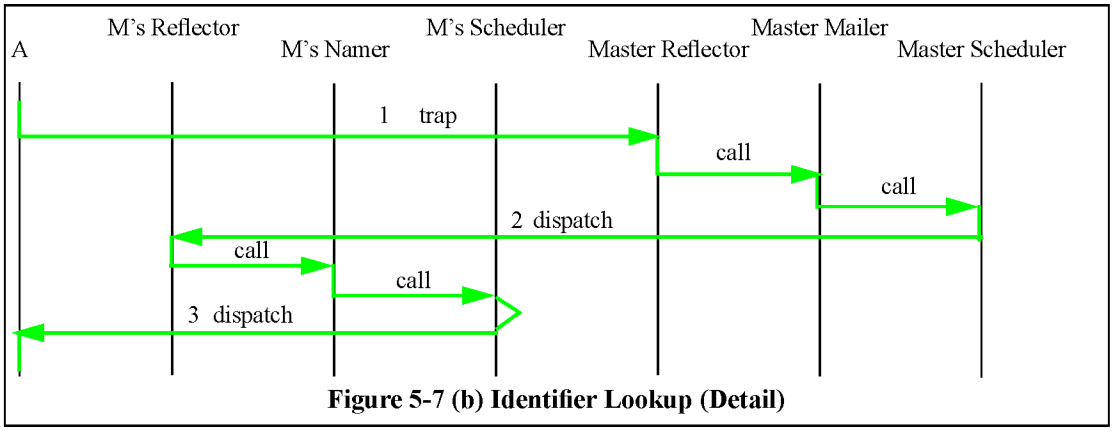


Figure 5-7 (b) Identifier Lookup (Detail)

number. To do this, *A* makes a system call (1), which is subsequently transferred to *A*'s meta-space, *M* (2). *M*'s reflector, seeing that an identifier lookup was requested, calls on the local namer to fulfill the request. By default, identifier lookup is not secured, so no security check takes place.

Figures 5-8 (a) and (b) illustrate what happens when naming is customized. In this figure, *A* would like the namer to log the life-time of applications on *M*. As before, *A* makes a system call (1), which is subsequently transferred to *A*'s meta-space, *M* (2). *M*'s reflector, noticing that the call is a call to the namer, notifies the meta-level namer. If the security policy considers tracking of statistics to be of no threat, no security check will be performed, and logging will commence.

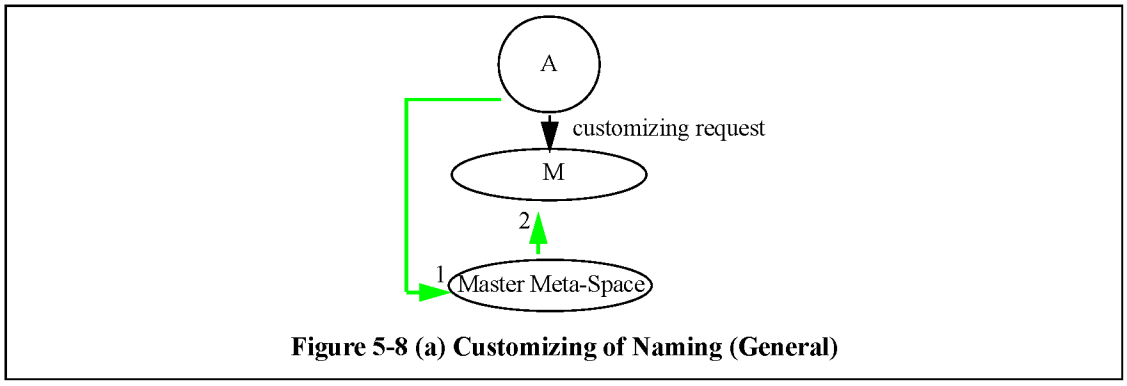


Figure 5-8 (a) Customizing of Naming (General)

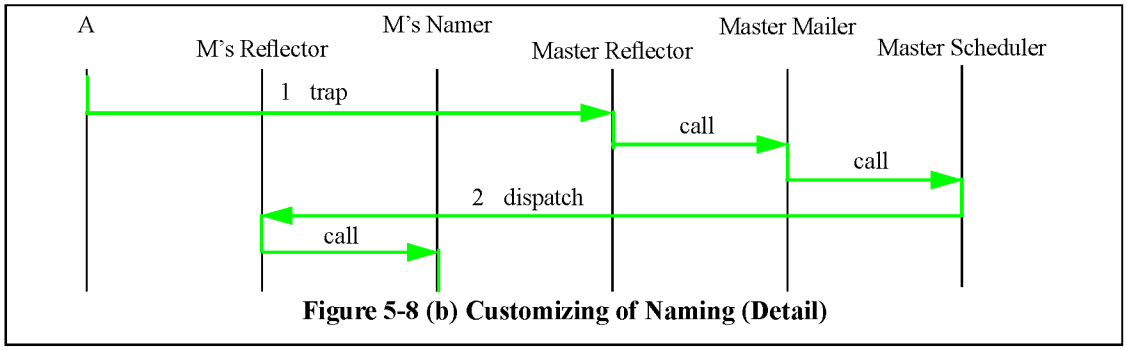


Figure 5-8 (b) Customizing of Naming (Detail)

Chapter 6

MetaOS Prototype

6.1 Building Blocks

Choosing a suitable language and hardware platform for the MetaOS prototype posed a challenge. On one hand, the choice of language should not encumber the performance of the prototype. On the other hand, to ease the implementation of MetaOS, the language should support abstract, type-safe interfaces. Since MetaOS heavily depends on interfaces for secure customizing, the latter requirement became the determining factor. As a result, Java was chosen as the prototype language; while performance is not its current *forté*, and while it forces system aspects such as context switches and address spaces to be simulated, it does provide abstract, type-safe interfaces. Java also offers useful prototyping features, such as automatic garbage collection, dynamic loading of classes, and exception handling. The use of Java furthermore obviated the selection of a hardware platform, due to Java's near-platform-independence. The remainder of this section discusses how the four building blocks of MetaOS --- meta-levels, meta-spaces, meta-objects, and meta-interfaces --- were implemented in Java. Subsequent sections will cover the implementation of MetaOS security and system services.

6.1.1 Meta-Levels

To facilitate interaction among the three hierarchy levels of MetaOS, any application or meta-level meta-space instance *I* contains a local *stub object* representing *I*'s meta-space. This stub object allows *I* to make system calls identical to the way any Java method is invoked. Internally, however, a call to a stub object's methods typically causes the stub to trap into the Master Meta-Space. This trap is simulated by the placement of a message into the Master Reflector's incoming message buffer. While this is not the purest Java solution --- *I* could have called the Master Reflector directly by means of Java method invocation --- it is closer to what happens in a bare-hardware-based system implementation.

Once the Master Reflector has received a message, it decides what should be done with it. In the case where *I* is an application attempting to communicate with its meta-space, the Master Meta-Space delegates the request to *I*'s meta-space. In the case *I* is a meta-level meta-space, its request is processed by the Master Meta-Space itself. By maintaining this hierarchical approach for system calls, the prototype enforces the three-level hierarchy.

6.1.2 Meta-Spaces

MetaOS meta-spaces are implemented by means of Java classes. In particular, each meta-space is an instance of a class, which in turn is derived from a generic meta-space class, *genericMSC*. The *genericMSC* class contains data structures for holding security-related information; data structures for holding references to the meta-objects which

make up the meta-space; and several methods that manipulate these structures. These methods, of which a partial list is given in Table 6-1, are all ultimately accessed through the reflector, and secured with ACLs.

Method	Interface	Effect
set authorization	declarative	authorizes a particular application or meta-space instance to make a particular call
remove authorization	declarative	removes authorization from a particular application or meta-space instance to make a particular call
check authorization	declarative	checks whether a particular application or meta-space instance is authorized to make a particular call
extend meta-space	imperative	extends the meta-space by adding another meta-object
contract meta-space	imperative	contracts the meta-space by removing a meta-object
set meta-space state	imperative	sets the state of a meta-space (state transfer is required for dynamic upgrades and cloning of meta-spaces)
get meta-space state	imperative	gets the state of a meta-space (state transfer is required for dynamic upgrades and cloning of meta-spaces)

Table 6-1: A List of Basic Meta-Space Methods

Set authorization, *remove authorization*, and *check authorization* form the basic security API, since they allow privileges to be added, removed, and checked. Section 6.2 will discuss their use in more detail. *Extend meta-space* and *contract meta-space* allow new meta-objects to be added to, and removed from the particular meta-space. *Set meta-space state* and *get meta-space state* support state transfer, which is part of dynamic upgrades and cloning. State transfer is important. For example, assume an old scheduler meta-object is about to be replaced by an updated scheduler meta-object. The old scheduler has to let the new scheduler know which processes are scheduled for execution, which ones are suspended, and so on; this is accomplished through state transfer.

The prototype comes with two meta-space classes derived from *genericMSC*: *masterMSC*, the Master Meta-Space class, and *simpleMSC*, a simple meta-space class designed to execute at the meta-level. Both automatically come with reflector meta-objects. Upon start-up, these reflectors load the other meta-objects of that meta-space, and fill in the ACLs in cooperation with the meta-objects. For example, the Master Reflector creates the Master Mailer, Master Namer, and Master Scheduler meta-objects, and places references to them into a meta-object table. Because of this table, calling meta-objects in the same meta-space involves a mere table lookup and a subsequent method invocation, such as *moTable.mailer.sendMsg(message, targetID)*. The Master Reflector then notifies each meta-object to give full access to the Master Meta-Space administrator.

6.1.3 Meta-Objects

All meta-objects are based on the generic meta-object class, called *genericMOC*, which implements the generic initialization method defined by the generic meta-object interface class, *genericMOI*, and hosts three internal member variables: A pointer to the log window object, to display error messages; a pointer to the meta-object table, to allow meta-objects to correspond with other meta-objects in the same meta-space; and a pointer to the current system call, including a full process descriptor of the invoker. Figure 6-1 shows a partial class hierarchy.

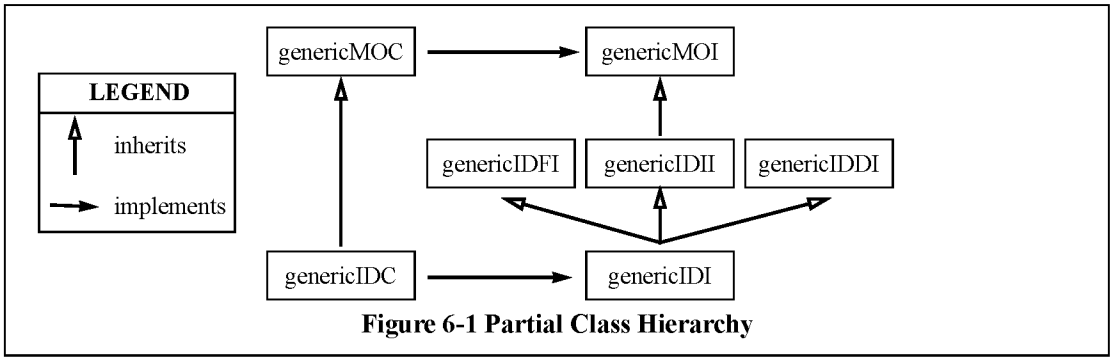


Figure 6-1 Partial Class Hierarchy

By subclassing *genericMOC*, the functionality of a meta-object can be specialized. For example, *genericIDC*, the generic namer class, is derived from *genericMOC*, and offers basic name services defined in the generic namer interface, *genericIDI*. *genericMsgC*, the generic mailer class, is also derived from *genericMOC*, but offers basic inter-process communication services defined in the generic mailer interface, *genericMsgI*.

Each meta-space further specializes these generic meta-objects. For example, the Master Meta-Space Namer specializes the generic namer by adding a unique naming policy. These specialized meta-objects do not directly subclass the generic meta-objects,

but refer to them by means of delegation [57]. For instance, *simpleIDC* and *masterIDC* are not directly subclassed from *genericIDC*, but from *genericMOC*. Whenever *simpleIDC* and *masterIDC* need to access the generic namer, they invoke it by means of the meta-object table. Since the meta-object table is modifiable, e.g., *genericIDC* can be replaced by an upgraded version of *genericIDC*, changing the underlying naming mechanism becomes easy; it also becomes scalable, because meta-space-specific meta-objects, like *simpleIDC* and *masterIDC*, will no longer have to be altered individually when *genericIDC* is modified.

6.1.4 Meta-Interfaces

All meta-objects implement functional, declarative, and imperative interfaces. These interfaces were implemented by means of Java interfaces. Because only one type of Java interface is available, the three types of interfaces were distinguished by the way they were named: All functional interfaces end with the letters *FI*, declarative interfaces end with the letters *DI*, and all imperative interfaces end with the letters *II*. Thus, *masterMsgFI* is the Master Mailer's functional interface, *masterMsgDI* is the Master Mailer's declarative interface, and *masterMsgII* is the Master Mailer's imperative interface.

6.2 Security

Each meta-space contains a private hash table. Each entry of this hash table represents a meta-object method and contains a queue which represents the ACL of that method.

Whenever a call with potentially significant impact reaches a meta-object, the meta-object retrieves its ACL queue from this table, and checks the invoker's identifier against the entries in the queue. If this succeeds, the call is allowed to go ahead. If this fails, it checks if the invoker's context has a user identifier associated with it, and if so, checks if the user's identifier is in the queue. If this succeeds, the call goes ahead. Otherwise, an error message is returned to the invoker. Manipulation of the hash table is accomplished by means of the previously-introduced *set authorization*, *remove authorization*, and *check authorization* given in Table 6-1.

The MetaOS model recommends that access to service objects be secured by means of ACLs. In case of the prototype, this recommendation was translated to the use of ACLs for securing context objects, since context objects consist of critical service objects like incoming message queues and threads associated with the context. As a result, all calls from outside a meta-space which try to manipulate a context object, be it by trying to read a message from the message queue or scheduling a thread, can only proceed if the invoker is in the target context object's ACL.

Address spaces are simulated by Java references. Whenever the model requires an entity E_1 to have access to the address space of another entity E_2, the prototype gives E_1 a reference to E_2. For example, a meta-level meta-space receives a reference to its applications, while applications do not receive a reference to their meta-level meta-space. However, this raises a problem when attempting to contact the Master Meta-Space. A reference is required to contact the Master, yet a reference to the Master could be a security loophole. Therefore, the MetaOS prototype requires that all application

and meta-level meta-space stubs employ a non-alterable class called *genericBOC*. Specifically, upon loading an application or meta-level meta-space instance *I*, the Master gives *I* access to an instance of *genericBOC*. *genericBOC* is furthermore given a reference to the Master Meta-Space. By using private variables and final methods, *genericBOC* ensures that applications and meta-level meta-spaces cannot access the Master Meta-Space reference given to *genericBOC*.

6.3 Services

The MetaOS prototype offers three operating system services: Scheduling, inter-process communication, and naming. The following discusses their implementation in more detail.

6.3.1 Scheduling

Java provides a default scheduling class; but this class only provides a priority-based scheduler, which cannot be customized dynamically. For this reason, the MetaOS prototype provides a dynamically-customizable scheduler running on top of Java. In particular, the Master Meta-Space launches a clock thread at Java's highest priority during boot-time, to simulate a timer chip. There are no other threads running at highest priority. Once started, the clock thread sends a *TICK* message to the Master Reflector every *t* milliseconds, where *t* is adjustable. Secondly, the Master Meta-Space thread that accepts, delegates, and processes requests runs at Java's second-highest priority. Finally, any other threads that are created and scheduled by the Master Scheduler, i.e., meta-level

threads, all execute at normal priority, and are suspended by being set to lowest priority. This means that the Master Meta-Space will generally get control of the CPU every *t* milliseconds, to make a scheduling decision. The interfaces of the generic scheduler, used both by the Master and Simple Schedulers, are shown in Table 6-2. Significant impact methods are italicized.

Following the MetaOS model, base-level threads are scheduled by meta-level meta-spaces. Specifically, base-level threads run at normal priority - 1, and are scheduled by the appropriate meta-level scheduler, running at normal priority. Meta-level schedulers are notified of timer interrupts by the Master Meta-Space, and can use these interrupts to schedule other base-level threads for execution.

The default MetaOS prototype scheduler provides a simple, round-robin-within-priority scheduling scheme. However, MetaOS can also support different scheduling schemes: Two of the MetaOS experiments involved the creation of a start-time fair-queueing (SFQ) scheduler [24,58] and a multi-level feedback (MLF) scheduler; both can be installed while MetaOS runs, and are described in the next chapter.

Table 6-2 lists *run-time* imperative interfaces, i.e., interfaces that authorized applications can call at run-time. There are also *compile-time* imperative interfaces, i.e., interfaces that applications cannot call at run-time, but that were explicitly designated to be customized by meta-object designers at compile time. *Compile-time* imperative interfaces contain methods that almost assuredly would cause an error, were they altered at run-time. For example, the generic scheduler interface provides a compile-time customizable method called *wake* which must be called to wake up a thread. The MetaOS prototype accomplishes this by using Java's thread priority functions; but Java's *resume*

Method	Interface	Effect
create thread	functional	creates a thread
destroy thread	functional	destroys a thread
schedule thread	functional	schedules a thread
unschedule thread	functional	unschedules a thread
next thread	functional	executes the next thread
preempt thread	functional	preempts the currently-running thread
current thread	functional	returns the ID of the currently-running thread
set priority	functional	sets a thread's priority
get priority	functional	gets a thread's priority
set queueing policy	declarative	sets the queueing policy of the scheduling queue
get queueing policy	declarative	gets the queueing policy of the scheduling queue
set interval policy	declarative	set the length of the time slice interval
get interval policy	declarative	gets the length of the time slice interval
set scheduler state	imperative	sets the state of the scheduler (for scheduler replacement)
get scheduler state	imperative	gets the state of the scheduler (for scheduler replacement)
clone scheduler	imperative	clones the scheduler

Table 6-2: A List of Basic Scheduler Methods

method could have been used as well. *wake*, however, should only be altered at compile time, since changing the way threads are implemented at run-time will most likely cause the MetaOS prototype to generate errors. Deciding which methods should go into the compile-time imperative interface is similar to deciding which methods should be over-

ridable, and which ones should be private: It has less to do with operating system design, and more with overall program design, and is therefore out of the scope of this work.

6.3.2 Inter-Process Communication

The main question surrounding the implementation of inter-process communication was whether the Master Meta-Space should be invoked directly by Java method invocation, or whether stubs should be used to trap into the kernel. As stated in Section 6.1.1, the latter approach was chosen, since it was closer to the design of a bare-hardware-based system. The run-time interfaces of the generic mailer, used both by the Master and Simple Mailers, are shown in Table 6-3. Significant impact methods are italicized.

An application or meta-level meta-space initiates inter-process communication by invoking the appropriate method of its stub object. This stub object then places the call and its parameters into the Master Meta-Space's incoming request message queue with help from *genericBOC*, as discussed in Section 6.2. The message queue is a priority queue, maintained by the Master Meta-Space, from which only the Master Meta-Space can remove messages. After the call is placed into the incoming request queue, it is processed in a fashion described in Section 5.3, i.e., the Master Reflector notices that a communication request was made, and notifies the Master Mailer by calling the *send message* method. If the invoker was a meta-level meta-space, the Master Mailer places the message into the target meta-space's incoming message queue. If the invoker was an application, the Master Mailer places the request into the invoking application's meta-space's incoming message queue. The Master Scheduler furthermore schedules the

Method	Interface	Effect
send message	functional	sends a messages
receive message	functional	receives a message (implemented by the stub; can only read the messages sent to the invoker)
reply to message	functional	replies to a message
set queueing policy	declarative	sets the message queuing policy (for the invoker only)
get queueing policy	declarative	gets the message queueing policy (for the invoker only)
set mailer state	imperative	sets the state of the mailer (for mailer replacement)
get mailer state	imperative	gets the state of the mailer (for mailer replacement)
clone mailer	imperative	clones the mailer

Table 6-3: A List of Basic Mailer Methods

meta-space, to allow it to process the request. Then, when the application's meta-space is executing, its reflector will pick up the message, and notify the meta-level mailer of the request. In case of intra-meta-space communication, the mailer will then place the message into the target application's incoming message queue. In case of inter-meta-space communication, the mailer will notify the Master Mailer, unless this type of communication is restricted by the meta-level meta-space's policy. Subsequently, the reply, if any, is placed into the invoking application's thread's reply queue.

In addition to the thread-specific reply queue, there is also an application-wide incoming message queue. This queue is used for messages that can be handled by any thread waiting on the incoming message queue. Typically, they are requests sent from another application or meta-space. Message queues enforce serialization, to deal with concurrency.

6.3.3 Naming

The namer is essentially made up of a hash table which maps machine-readable identifiers to contexts and to human-readable names. Contexts contain various data, including creator information. Human-readable names allow applications to look for generically-named services. Within meta-spaces, however, only the machine-readable identifiers are used. The run-time interfaces of the generic namer, used both by the Master and Simple Namers, are shown in Table 6-4. Significant impact methods are italicized. Note that *createID* and *destroyID* basically create a context, calling on other meta-objects to deal with other parts of the object's context, such as its threads.

The Master Namer generates application or meta-space identifiers based on the machine name and a counter. This counter is incremented by one, each time an application or meta-space is created. If the counter wraps around, the protocol described in Chapter 4 is used to recycle names, i.e., the Master Namer picks a counter value which is not found in the hash table. Similarly, user names are created based on machine name and a counter. As above, in case the counter wraps, user names are created using the protocol described in Chapter 4.

Method	Interface	Effect
create identifier	functional	creates an identifier
destroy identifier	functional	destroys an identifier
query identifier	functional	queries an identifier for creator information, etc.
set namer state	imperative	sets the state of the namer (for namer replacement)
get namer state	imperative	gets the state of the namer (for namer replacement)
clone namer	imperative	clones the namer

Table 6-4: A List of Basic Namer Methods

Chapter 7

Prototype Experiments

7.1 Making Major Changes

Many experiments were conducted to explore and demonstrate the capabilities of MetaOS. Among these experiments, six were especially crucial. The first of these six showed that a major change can be securely introduced into a meta-space.

An important service that can be offered by meta-spaces is inter-process multi-casting. It is possible to provide these services by an application running on top of a meta-space, but providing this service inside a meta-space is much more effective, since it reduces the number of context switches. However, installing such a service cannot be solely up to any application's discretion, because the application could try to introduce a Trojan horse, or it could lead to an overly-bulky meta-space. Thus, installation of such a service may only be done by an authorized application or meta-level meta-space.

Multicasting could have been added to the mailer, but this would have unneces-sarily complicated the mailer meta-object. Therefore, a new group multicast meta-object was created to provide methods which: Allow multicast groups to be created and destroyed; allow objects to join and leave a multicast group; and allow objects to send a message to a particular multicast group. These methods were implemented by manipu-lating a table-like structure, using a group's identifier as key. Thus, sending a multicast

message merely involves retrieving the group members using that particular key, and sending messages to each one of them with help from the mailer meta-object.

Run-time installation of the meta-object was straightforward. First, an installer application was created to install this customized code into the Simple Meta-Space. All this program does is to instantiate the multicast class, and then pass it to the Simple Meta-Space, using the meta-space's *extend meta-space* method. Because the Simple Meta-Space's security policy limits installation of new meta-objects to a particular administrator, only that administrator can run the installer and install the meta-object. Attempts by others were rejected by the Simple Meta-Space. This way, major changes to the meta-space are confined to the administrator with legitimate authorization, as required by the MetaOS model.

7.2 Making Minor Changes

Many applications will, at most, want to make minor changes, i.e., changes that impact only themselves. One such change is to turn on debugging of inter-process communication, which is useful to see what is sent to other applications, in what order it is sent, and to whom it is sent. Since debugging is selective, i.e., can only be turned on for the invoker, it is a local-effect declarative call, and therefore freely accessible to all.

Implementation of the debugging meta-object was simple. Since its functionality is essentially identical to that of the mailer meta-object, the debugger was created by subclassing an existing mailer meta-object. Next, each method which sends out a mes-

sage was overridden to allow the incoming message data to be made available outside of the meta-space.

Run-time installation was similar to that of the multicast meta-object: An installer was written that instantiates a debugging mailer, and passes it to the Simple Meta-Space. Again, only an authorized administrator can successfully accomplish the installation. However, once this is done, applications can freely turn on debugging as they see fit, as this only affects the invoking applications; other applications continue to execute unaffected.

7.3 Cloning

Another feature of the MetaOS model is cloning. For example, assume that an application requiring an overall multi-level feedback (MLF) scheduling has just started to execute on a default meta-space. By querying the declarative interfaces, it discovers that the underlying scheduling policy is round-robin, and that there are no other meta-spaces providing MLF scheduling. Since customizing the overall scheduling policy is a change with significant impact on other applications, a typical application cannot use the meta-interfaces of the meta-space to make this change. Furthermore, since this is such a drastic change, negotiation to introduce this new policy by means of the meta-space manager will fail. Thus, the application is required to clone the existing meta-space by sending a clone request to the meta-space. The meta-space then clones itself, and migrates the application to the clone, where the application has full control of the clone.

For this experiment, a Simple Meta-Space application was created which queries its meta-space's scheduling policy. Since this policy was not MLF, the application requested a clone of the Simple Meta-Space. The Simple Meta-Space subsequently created a clone and migrated the application to the clone, to which the invoking application received full access. Once there, the application instantiated a MLF scheduler meta-object, which was derived by overriding the scheduling policy of the generic scheduler. Since the application received full access rights to the clone, it was allowed to install the new scheduler.

7.4 Migrating

Before resorting to the radical step of cloning, an application can check if there is a meta-space available which already supplies the policies it requires. For example, consider an application which fares best under start-time fair-queueing (SFQ) scheduling. Using the Master Namer, it can look up the names of other publicly-available meta-spaces, and query these meta-spaces to see if the meta-spaces are compatible and provide the policies it requires. Then it can request its meta-space to migrate it to the appropriate meta-space, and once there, benefit from the SFQ policy.

For this experiment, two instances of the Simple Meta-Space were loaded. Then, an SFQ installer was run by an authorized administrator, to install an SFQ scheduler in one of the meta-spaces. Similar to the MLF scheduler, the SFQ scheduler was created by overriding the appropriate methods of the generic scheduler. Next, the test application was placed on the meta-space with the default round-robin scheduler. When it

started to execute, the application queried the Master Namer to see if there is another Simple Meta-Space. Since this was the case, it asked the other Simple Meta-Space what its scheduling policy was, and being SFQ, asked its meta-space to migrate it to the other Simple Meta-Space. Once there, the application then began to run its main program.

7.5 Customizing Security

To test the secure customizing claims, the author decided to dynamically introduce a meta-space to MetaOS which supports a simple capability model, rather than ACLs. For this experiment, typical applications running on the Capability Meta-Space initially received an empty capability. Only applications started by a fully-authorized user received a capability giving full access to the meta-space. Whenever an application wanted to make a change with significant impact, it had to pass the capability to the Capability Meta-Space. The Capability Meta-Space checked if the capability allowed the particular call to be made, and if so, proceeded with the call. Otherwise, an exception was generated.

To test capability transfer, the administrator started an application on the Capability Meta-Space. As expected, the Capability Meta-Space gave this application the capability to fully access the Capability Meta-Space. The application then transferred this capability to another, designated application which did not have such rights. After the capability transfer, however, the other application was able to make calls which require full authorization, as predicted.

7.6 Testing the Firewall

Another experiment involved testing of the meta-space firewalls. In particular, MetaOS stipulates that a crash in one meta-level meta-space should not affect any other meta-level meta-space. To test this, a scheduler meta-object was created which crashes after a few ticks. Next, two instances of the Simple Meta-Space were started, and applications were loaded onto each of them. Then, an authorized administrator installed the defective scheduler into one of the Simple Meta-Spaces. After a few ticks, the applications on the targeted meta-space froze, but the other meta-space and its applications continued to execute unaffected, as expected.

7.7 Summary

The above six experiments demonstrated key aspects of MetaOS's secure customizing: It was not possible for an unauthorized application to make the significant change of adding a new meta-object to the meta-space, but it was possible for an unauthorized application to customize itself in a way which only affected itself. The prototype also allowed an application to clone its meta-space, migrate to the clone, and customize it as it saw fit. It furthermore permitted an entirely new security policy and mechanism to be introduced dynamically, and confined the effects of a crash in one meta-space, allowing other meta-spaces and their applications to continue executing as before.

Chapter 8

Conclusions and Future Work

Over the years, many new computer applications have been developed. Time and again, they uncovered weaknesses in existing operating systems which had to be addressed by newer generations of system software. Multimedia computing is just the latest example. This on-going dilemma can be partially resolved by creating operating systems which can be dynamically customized throughout their life-times. This way, as new application requirements become known, appropriate system services can be devised and installed in existing systems. However, opening an operating system to application-initiated changes needs to be done in a secure manner. Furthermore, the manner in which this security is provided should be just as securely customizable as any other part of the system. MetaOS therefore supports secure run-time customizing, including secure customizing of the security model itself. This is accomplished by using meta-levels to localize customizable system services, meta-spaces to isolate run-time environments from each other, meta-objects to modularize meta-spaces, and meta-interfaces to securely customize run-time environments.

An interesting aspect of MetaOS is that it allows very fine-grained, remote system administration to take place: Rather than depending on one trusted administrator, it is possible to limit access on a per-meta-space basis, making it feasible to allow different vendors to administer their meta-spaces without giving access to unrelated parts of the

system. This can significantly aid in the spread of on-line technical assistance, as it allows users to open their system to vendors without giving the vendors access to any but the minimum required code and data. Similarly, it could help to turn today's internet service providers into full-fledged system service providers: Users could run their network computers at home or from the office, as is the case with today's desktop computers. However, should there be problems, the user could ask the service provider for assistance. The provider could then connect to the machine, and investigate the cause of the problem. Since the security of MetaOS is very fine-grained, the service provider can be restricted to a particular meta-space, and thus be prevented from accessing unrelated user data and code belonging to other meta-spaces. Appendix B talks about the progress made to date.

MetaOS's run-time performance is another issue which requires further work. One partial solution is to port MetaOS to a non-interpreted environment. However, many of Java's better features, such as abstract, type-safe interfaces, could be lost in the process. Another possibility is to adapt a bare-hardware-based Java VM, such as JavaOS [59], to MetaOS. While this would not be as fast as the first approach, it would benefit from many of Java's features.

Designing a securely-customizable system is not an easy task, partly because security is difficult to achieve, and partly because flexibility and security are basically opposite goals. Allowing the security model itself to be customized complicates this task even more. However, MetaOS demonstrates that it is possible. It is the hope of the author that this work will prove to be a valuable aid to those developing customizable operating systems.

Bibliography

[1] M. Stonebraker, "Operating System Support for Database Management," *Comm. the ACM*, Vol. 24, No. 7, 1981, pp. 412-418.

[2] P. Cao, E.W. Felten, and K. Li, "Implementation and Performance of Application-Controlled File Caching," *Proc. 1st Symp. Operating Systems Design and Implementation*, USENIX Assoc., Berkeley Calif., 1994, pp. 165-177.

[3] J. Nieh et al., "SVR4 UNIX Scheduler Unacceptable for Multimedia Applications," *Proc. 4th Workshop on Network and Operating System Support for Digital Audio and Video*, Doug Shepherd et al., eds., Springer Verlag, New York, 1994, pp. 41-53.

[4] P. Goyal, X. Guo, and H.M. Vin, "A Hierarchical CPU Scheduler for Multimedia Operating Systems," *Proc. 2nd Symp. Operating Systems and Design*, USENIX Assoc., Berkeley, Calif., 1996, pp. 107-121.

[5] J. Bruno et al., "Move-To-Rear List Scheduling: A New Scheduling Algorithm for Providing QoS Guarantees," *Proc. ACM Multimedia*, ACM Press, New York, 1997, pp. 63-73.

[6] B. Özden, R. Rastogi, and A. Silberschatz, "Buffer Replacement Algorithms for Multimedia Storage Systems," *Proc. IEEE Multimedia '96*, IEEE Press, Los Alamitos, Calif., 1996, pp. 172-180.

[7] B.N. Bershad, D.D. Redell, and J.R. Ellis, "Fast Mutual Exclusion for Uniprocessors," *Proc. 5th Int'l Conf Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, 1992, pp. 223-233.

[8] C. Pu et al., "Optimistic Incremental Specialization: Streamlining a Commercial Operating System," *Proc. 15th ACM Symp. Operating Systems Principles*, ACM Press, New York, 1995, pp. 314-324.

[9] T. von Eicken et al., "Active Messages: A Mechanism for Integrated Communication and Computation," *Proc. 19th Ann. Int'l Symp. Computer Architecture*, ACM Press, New York, 1992, pp. 256-266.

[10] P. Druschel, V.S. Pai, and W. Zwaenepoel, "Extensible Kernels are Leading OS Research Astray," *Proc. 6th Workshop on Hot Topics in Operating Systems*, IEEE Press, Los Alamitos, Calif., 1997, Pages Unknown.

[11] W.P. Stevens, G.J. Myers, and L.L. Constantine, "Structured Design," *IBM Systems J.*, Vol. 13, No. 2, 1974, pp. 115-139.

[12] Chiba, et al., *Foil for the Workshop on Open Implementations*, G. Kiczales, ed., http://www.parc.xerox.com/spl/projects/oi/workshop-94/foil/main.html (current May 18, 1998).

[13] K. Krueger et al., "Tools for the Development of Application-Specific Virtual Memory Management," *Proc. 8th Ann. Conf. Object-Oriented Programming Systems, Languages and Applications*, ACM Press, New York, 1993, Pages 48-64.

[14] R. Levin et al., "Policy/Mechanism Separation in HYDRA," *Proc. 5th ACM Symp. Operating Systems Principles*, ACM Press, New York, 1975, pp. 132-140.

[15] W.A. Wulf, R. Levin, and S.P. Harbison, *HYDRA/C.mmp: An Experimental Computer System*, McGraw-Hill, New York, 1981.

[16] P.B. Hansen, "The Nucleus of a Multiprogramming System," *Comm. the ACM*, Vol. 13, No. 4, 1970, pp. 238-250.

[17] R.J. Creasy, "Origin of the VM/370 Time-Sharing System," *IBM J. Research and Development*, Vol. 25, No. 5, 1981, pp. 483-490.

[18] T.F. La Porta and M. Schwartz, "Architectures, Features, and Implementations of High-Speed Protocols," *IEEE Network Magazine*, Vol. 5, No. 5, 1991, pp. 14-22.

[19] H. Schulzrinne, "Operating System Issues for Continuous Media," *Multimedia Systems*, Vol. 4, No. 5, 1996, Pages 269-280.

[20] T.E. Anderson, *The Case for Application-Specific Operating Systems*, Tech. Report CSD-93-738, Computer Science Dept., Univ. of California at Berkeley, Berkeley, Calif., 1993.

[21] M.I. Seltzer et al., "Issues in Extensible Operating Systems," submitted for publication.

[22] C. Small and M. Seltzer, "Structuring the Kernel as a Toolkit of Extensible, Reusable Components," *Proc. 4th Int'l Workshop on Object Orientation in Operating Systems*, IEEE Press, Los Alamitos, Calif., 1995, pp. 134-137.

[23] M. Horie et al., "Designing Meta-Interfaces for Object-Oriented Operating Systems," *Proc. 1997 IEEE Pacific Rim Conf. Comm., Computers, and Signal Processing*, IEEE Press, Los Alamitos, Calif., 1997, pp. 989-992.

[24] M. Horie et al., "Using Meta-Interfaces to Support Secure Dynamic System Reconfiguration," *Proc. 1998 IEEE Int'l Conf. Configurable Distributed Systems*, IEEE Press, Los Alamitos, Calif., 1998, pp. 164-171.

[25] J.G. Mitchell et al., "An Overview of the Spring System," *Proc. 39th IEEE Int'l Computer Conf.*, IEEE Press, Los Alamitos, Calif., 1994, pp. 122-131.

[26] G. Hamilton and P. Kougiouris, "The Spring Nucleus: A Microkernel for Objects," *Proc. USENIX Summer 1993 Technical Conf.*, USENIX Assoc., Berkeley, Calif., 1993, pp. 147-159.

[27] Y. Khalidi and M. Nelson, *The Spring Virtual Memory System*, Tech. Report SMLI-93-09, Sun Microsystems Lab, Palo Alto, Calif., 1993.

[28] R. Rashid et al., "Mach: A Foundation for Open Systems," *Proc. 2nd Workshop on Workstation Operating Systems*, IEEE Press, Los Alamitos, Calif., 1989, Pages Unknown.

[29] J.B. Dennis and E.C. van Horn, "Programming Semantics for Multiprogrammed Computations," *Comm. the ACM*, Vol. 9, No. 3, Mar. 1966, pp. 143-155.

[30] D.R. Cheriton and K.J. Duda, "A Caching Model of Operating System Kernel Functionality," *Proc. 1st Symp. Operating Systems Design and Implementation*, USENIX Assoc., Berkeley, Calif., 1994, pp. 179-193.

[31] D.R. Engler, M.F. Kaashoek, and J. O'Toole Jr., "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proc. 15th Symp. Operating Systems Principles*, ACM Press, New York, 1995, pp. 251-266.

[32] M.F. Kaashoek et al., "Application Performance and Flexibility of Exokernel Systems," *Proc. 16th Symp. Operating Systems Principles*, ACM Press, New York, 1997, pp. 52-65.

[33] B. Ford et al., "Microkernels Meet Recursive Virtual Machines," *Proc. 2nd Symp. Operating Systems Design and Implementation*, USENIX Assoc., Berkeley, Calif., 1996, pp. 137-151.

[34] D.C. Kulkarni, *Pi: A New Approach to Flexibility in System Software*, doctoral dissertation, Notre Dame Univ., Dept. Computer Science and Eng., Notre Dame, Ind., 1995.

[35] B.N. Bershad et al., "Extensibility, Safety and Performance in the SPIN Operating System," *Proc. 15th ACM Symp. Operating Systems Principles*, ACM Press, New York, 1995, pp. 267-284.

[36] P. Pardyak and B.N. Bershad, "Dynamic Binding for an Extensible System," *Proc. 2nd Symp. Operating Systems Design and Implementation*, USENIX Assoc., Berkeley, Calif., 1996, pp. 201-212.

[37] M.I. Seltzer et al., "Dealing With Disaster: Surviving Misbehaved Kernel Extensions," *Proc. 2nd Symp. Operating Systems Design and Implementation*, USENIX Assoc., Berkeley, Calif., 1996, pp. 213-227.

[38] R. Wahbe et al., "Efficient, Software-Based Fault Isolation," *Proc. 14th ACM Symp. Operating Systems Principles*, ACM Press, New York, 1993, pp. 203-216.

[39] Y. Yokote, "The Apertos Reflective Operating System: The Concept and Its Implementation," *Proc. 7th Ann. Conf. Object-Oriented Programming Systems, Languages and Applications*, ACM Press, New York, 1992, pp. 414-434.

[40] J. Itoh, R. Lea, and Y. Yokote, "Using Meta-Objects to Support Optimisation in the Apertos Operating System," *Proc. USENIX Conf. Object-Oriented Technologies and Systems*, USENIX Assoc., Berkeley, Calif., 1995, pp. 147-157.

[41] R. Lea, Y. Yokote, and J. Itoh, "Adaptive Operating System Design Using Reflection," *Proc. 5th Workshop on Hot Topics in Operating Systems*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1995, pp. 95-101.

[42] P. Maes, "Concepts and Experiments in Computational Reflection," *Proc. 2nd Ann. Conf. Object-Oriented Programming Systems, Languages and Applications*, ACM Press, New York, 1987, pp.147-155.

[43] P. Cao et al., "Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling," *ACM Trans. Computer Systems*, Vol. 14, No. 4, Nov. 1996, pp. 311-343.

[44] C. Lee, M.C. Chen, and R. Chang, "HiPEC: High Performance External Virtual Memory Caching," *Proc. 1st Symp. Operating Systems Design and Implementation*, USENIX Assoc., Berkeley, Calif., 1994, pp. 153-164.

[45] P.C.H. Lee, R. Chang, and M.C. Chen, "Hipec: A System for Application-Customized Virtual-Memory Caching Management," *Software --- Practice and Experience*, Vol. 27, No. 5, May 1997, pp. 547-571.

[46] G. Necula and P. Lee, "Safe Kernel Extensions without Run-Time Checking," *Proc. 2nd Symp. Operating Systems Design and Implementation*, USENIX Assoc., Berkeley, Calif., 1996, pp. 229-243.

[47] G. Necula, "Proof-Carrying Code," *Proc. 24th Ann. Symp. Principles of. Programming Languages*, ACM Press, New York, 1997, pp. 106-119.

[48] A. Turing, "On Computable Numbers with an Application to the Entscheidungsproblem," *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems, and Computable Functions*, M. Davis, ed., Raven Press, Hewlett, N.Y., 1965, pp. 116-154.

[49] B.C. Smith, *Reflection and Semantics in a Procedural Language*, Tech. Report 272, Laboratory of Computer Science, Massachusetts Inst. of Technology, Cambridge, Mass., 1982.

[50] S. Matsuoka, T. Watanabe, and A. Yonezawa, "Hybrid Group Reflective Architecture for Object-Oriented Concurrent Programming," *Proc. 1991 European Conf. Object-Oriented Programming*, P. America, ed., Springer Verlag, New York, 1991, pp. 231-250.

[51] M. Horie, E.G. Manning, and G.C. Shoja, *Exploring the Apertos Operating System*, Tech. Report DCS-238-IR, Dept. Computer Science, Univ. of Victoria, Victoria, B.C., 1995.

[52] J. Liedtke, "The Performance of μ-Kernel-Based Systems," *Proc. 16th ACM Symp. Operating System Principles*, ACM Press, New York, 1997, pp. 66-77.

[53] K. Murata et al., "Unification of Compile-Time and Run-Time Meta-Level Definitions," *Advances in Object-Oriented Metalevel Architectures and Reflection*, C. Zimmerman, ed., CRC Press Inc., Boca Raton, Fla., 1996, Pages Unknown.

[54] G. Kiczales, "Why are Black Boxes so Hard to Reuse? (Towards a New Model of Abstraction in the Engineering of Software)," http://www.parc.xerox.com/spl/projects/oi/towards-talk/default.html (current May 18, 1998).

[55] R. Rao, "Implementational Reflection in Silica," *Proc. 1991 European Conf. Object-Oriented Programming*, P. America, ed., Springer Verlag, New York, 1991, pp. 251-267.

[56] W.T. Strayer, B.J. Dempsey, and A.C. Weaver, *Xtp: The Xpress Transfer Protocol*, Addison-Wesley, Reading, Mass., 1992.

[57] K.J. Lieberman, "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems," *Proc. 1st Ann. Conf. Object-Oriented Programming Systems, Languages and Applications*, ACM Press, New York, 1986, pp. 214-223.

[58] P. Goyal, H.M. Vin, H. Cheng, "Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks," *Proc. ACM SIGCOMM 1996*, ACM Press, New York, 1996, pp. 157-168.

[59] P. Madany et al., *JavaOS: A Stand-Alone Java Environment*, http://java.sun.com/products/javaos/javaos.white.html (current May 18, 1998).

[60] *Computer Associates: Enterprise Management Strategy White Paper*, http://www.cai.com/products/unicent/whitepap.htm#link13 (current May 18, 1998).

[61] R. Hawes et al., *TME 10 Security Management*, http://www.tivoli.com/redbooks/html/sg242021/2021fm.htm (current May 18, 1998).

[62] *The "Zero Administration" Initiative for Windows*, http://www.microsoft.com/windows/platform/info/zawmb.htm (current May 18, 1998).

[63] J.G. Steiner, B.C. Neuman, and J.I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," *Proc. 1988 USENIX Winter Conf.*, USENIX Assoc. Berkeley, Calif., 1988, pp. 191-202.

Appendix A

Glossary

Configuration. A *configuration* is essentially the same as a *customization*, although some define a configuration as a one-time event which occurs at boot-time. Under the latter definition, a change to the original configuration after boot-time is called a *reconfiguration*.

Customization. An operating system which allows aspects of its implementation to be altered at one or all points between boot-time and shut-down-time is considered a *customizable* system. The part of code which replaced or augmented the default system implementation is called a *customization*. Note that systems which can be altered at compile-time are not considered customizable in this work since, generally speaking, all systems can be changed at compile time. Expanding the term to include compile-time changes would cause it to lose its significance for purposes of this work.

Dispatcher. In SPIN, this is the entity which decides what *event handlers* to call when a particular *event* occurs.

Event. In SPIN, this is equivalent to a kernel method.

Event Handler. In SPIN, this is the entity which implements the *event*.

Exokernel. A system kernel, like Aegis, which attempts to push all system services, such as scheduling and memory management, to the user-level.

Extension. An *extension* is essentially the same as a *customization*, although some distinguish between the two, saying that an extension only covers additions to a system. Under the latter definition, an extensible system would allow new code to be added, but would not permit existing code to be changed.

Framework. A customizable set of classes which jointly form a component, such as a scheduler or a mailer.

Graft. In VINO, this is the entity which implements the *customization*.

Mechanism. The means by which an action occurs. Separating *mechanism* from *policy* as much as possible helps a system to be customizable.

Meta. Stating that *A* is *meta* to *B* means that *A* is behind *B*, or *A* works in the background of *B*. More specifically, computational *reflection* states that behind every program is a model of computation. Hence, this model of computation is meta to the program itself. Objects within the model of computation are therefore called *meta-objects*, interfaces to objects within this model are called *meta-interfaces*, and groups of meta-objects are called *meta-spaces*.

Microkernel. A kernel, like Spring, which attempts to push many traditional system services, such as file systems, to the user-level. Taking this approach to its extreme results in an *exokernel*.

Module. An *object*.

Object. An entity which encapsulates data and methods which modify this data. The data and some of the methods are usually hidden. Methods that are not hidden are the only official means by which to interact with the object.

Object-Oriented. For the purposes of this work, object-oriented means that system entities are abstracted as objects, and that most objects are created by inheriting data or methods from other objects.

Policy. Determines in what manner a system service is provided (e.g., using LRU or MRU for paging). A *policy* is used in conjunction with a *mechanism*. Separating mechanism from policy as much as possible helps a system to be customizable.

Reconfiguration. See *configuration*.

Reflection. Every program has a model of computation. In a language, the model specifies, among other things, how the program looks up methods and variables. In an operating system, the model specifies how a program accesses memory, how it is being scheduled, and so on. Model and program are causally connected, i.e., a change in the model directly affects the way the program is executed, and a change in the program directly affects the model. Reflection occurs when a program observes or modifies its own model of computation

Reflector. A *meta-object* which delegates an incoming call to another meta-object.

Sandboxing. *Sandboxing* takes existing assembly code, and modifies it such that only legal memory addresses can be accessed.

Script. A sequence of commands which can be interpreted by the kernel. A script is generally used to implement an application-specific resource allocation policy.

Subcontract. In Spring, an entity which is in charge of transmitting data between clients and their servers.

Type Safety. Type safety ensures that any variable of type T can only reference instances of T or its subtypes. This prevents users from using clever casting schemes to directly access memory owned by T or its subtypes, thereby bypassing the interfaces of T or its subtypes.

Appendix B

Remote Administration

In principle, MetaOS supports fine-grained, secure remote administration. There already are some remote administration tools, such as Computer Associates' Unicenter TNG [60], Tivoli's TME [61], and Microsoft's ZAW [62], but these products generally assume the existence of a super-user who has complete access to a system. MetaOS, on the other hand, allows administrative access to be much more fine-grained, confining access to individual meta-spaces. For example, assume there is a meta-space vendor or supporter *V*, and a user *U*. Furthermore, assume *U* experiences some trouble installing or using *V*'s meta-space *M*. MetaOS permits *V*, with permission of *U*, to remotely log onto *U*'s system, and investigate and customize *M*. Thus, rather than relying on a novice user's explanations, or trying to ask a novice user to enact certain changes via telephone, trained maintenance personnel have the opportunity to directly analyze and solve the problem.

Remote administration opens a security loophole. Specifically, *V* could try to access meta-spaces other than *M*. However, MetaOS's fine-grained security allows meta-spaces to be independently secured against unauthorized accesses, meaning that giving *V* access to *M* will not automatically give *V* access to any other meta-space.

A second problem is that *M* could be a Trojan horse, and once the user grants it certain access rights to certain data, it could compromise this data. For example,

although promising to be a reliable transaction meta-space, M could secretly be scanning data for credit card numbers, and transmit them to an unauthorized third party. However, this is a risk inherent to running an unknown application, and not exclusive to remote administration. Furthermore, since MetaOS meta-spaces cannot automatically access each other's data, the damage that M can cause will be limited to the data which was explicitly given to it.

A third problem arises when unsecured networks, rather than dedicated lines, are used for remote administration. For example, a third party V' could pretend to be V, or V' could intercept commands of V, and analyze, modify, or replay them at a later time. These problems have been investigated in depth, and can be addressed by solutions such as the Kerberos protocol [63]. Integrating Kerberos into MetaOS is subject to future research.

Figures B-1 (a) and (b) illustrate the control flow generated by remote administration. Here, A_r is the remote administration tool run by V, and A is an application run by user U on top of V's meta-space, M. In the Figure, V has already logged onto the customer's machine, and now initiates a customizing command from A_r. Following MetaOS's general control flow, a trap is generated to the Master Reflector (1), which then invokes the Master Mailer and Scheduler. For simplicity's sake, this step has been abbreviated with an asterisk in (b). A_r's meta-space is subsequently contacted to deal with local policy, such as adding encryption (2). The vendor's Master Meta-Space is then commissioned to send the message to the user's machine (3). At the other end, the user's Master Reflector receives a network interrupt (4), and notifies the Master Mailer.

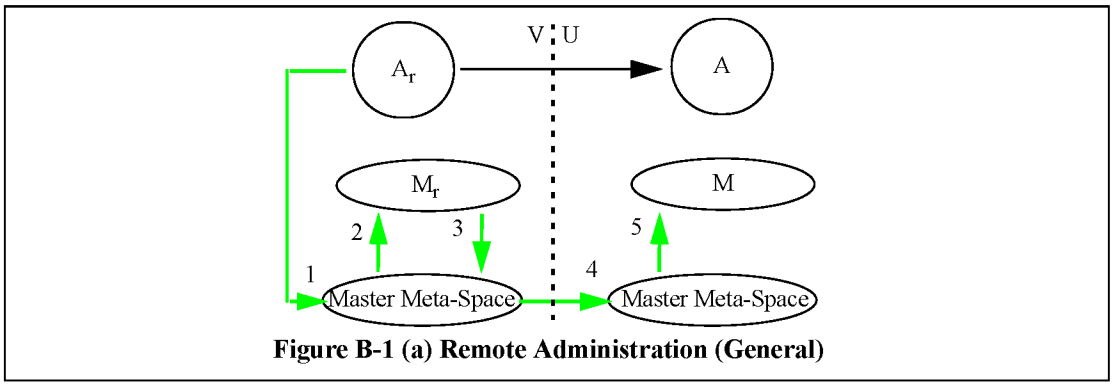


Figure B-1 (a) Remote Administration (General)

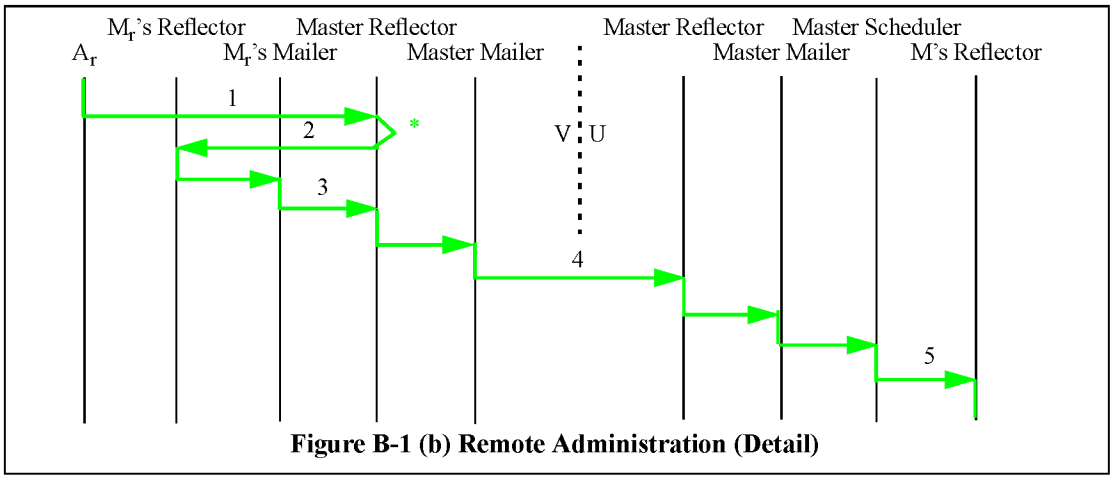


Figure B-1 (b) Remote Administration (Detail)

The Mailer detects that the message is meant for *M*, and places the message in *M*'s mail-box. Next, *M* is scheduled. When *M* receives the message (5), it checks if the message is authentic, correct, and authorized, and if so, executes the remote command.