SCS&E Report 9303
March, 1993

Capability-Based Protection in a Persistent Global Virtual Memory System

Jerry Vochteloo, Stephen Russell, Gernot Heiser

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
THE UNIVERSITY OF NEW SOUTH WALES

Abstract

A single address-space encompassing all virtual memory of a distributed computer system is an ideal environment for a persistent system. The issue of providing effective and efficient protection of objects in such an environment has, however, not been addressed satisfactorily. We propose a system which is based on password capabilities. A system-maintained data structure called the *capability tree* is used for long-term storage of capabilities, and reflects the hierarchical structure of object privacy. A second system data structure, the *active protection domain*, allows the system to find capabilities quickly when validating memory accesses. The proposal supports inheritance of protection domains, as well as temporary extension of protection domains to support privileged procedures. Untrusted programs can be confined to run in a restricted protection domain. The protection system performs efficiently on conventional architectures, and is simple enough that most programs do not need to be aware of its operation.

1 Introduction

Persistent systems (PS) offer the potential to greatly simplify application programming, as complex memory data structures no longer have to be translated into low-level linear arrays for permanent storage [1]. However, their practical use has so far been hampered by the problems associated with the storage of pointers on secondary memory, which lose their meaning if taken out of their address-space. As we have pointed out earlier [2], a clean and elegant solution to this dilemma is to provide an all-encompassing single address-space, which contains the virtual memory of all processes on all nodes in a distributed computing system. Such a global virtual memory system (GVMS) is automatically persistent as the validity of an object's address is decoupled from the existence of the object's creator process. The recent advent of 64-bit microprocessors (MIPS R4000 [3] and DEC Alpha [4]) makes such a single address-space operating system possible.

In the GVMS user processes only see a single, flat virtual memory. An individually allocated segment of virtual memory is called an *object*; objects are page-aligned and are the basic unit of protection. All data are referenced in a uniform manner by issuing an address, and addresses can be passed around freely. Sharing is therefore extremely simple and is not limited by any system-imposed restrictions other than protection against unauthorised access. Distribution is transparent and data can migrate freely between computing nodes; whenever a non-resident page is referenced by a process it is obtained from whichever node currently holds that page. The network is in this sense just an extension of the local paging disk. Processes can migrate just as easily as data: if the process control block (PCB), which contains the register set, is migrated to a different node, the process will, as soon as it starts running again, fault its working set across the network. More details of the system, including support for replication and fault tolerance, can be found in [5].

Potentially the biggest problem associated with a single address-space is security. Protection in traditional operating systems mostly depends on the fact that processes are running in different address-spaces: since it is impossible for a process to address an object outside its own address-space, explicit system intervention is required to make such objects accessible. The system has full control over such accesses and can reliably impose a protection model.

In a single address-space, however, every object is visible to each process, and no explicit system interaction is required to access an arbitrary object. Different protection mechanisms, which do not depend on address-space separation, must be employed in the GVMS. We propose a system based on password capabilities [6], which is largely transparent to the user, yet maintains a level of protection comparable to traditional approaches. In particular it gives users control over their protection domains and allows them to deal safely with untrusted programs. A set-uid-like service for temporary extension of protection domains is also provided.

The remainder of this paper focuses on the issue of protection in the GVMS. Section 2 presents the password capability scheme that forms the basis of the protection model. Section 3 describes the system-maintained data structure where users store capabilities, while Section 4 describes how the system performs access validation. Section 5 explains how protection domains can be inherited, restricted or expanded by user processes. Section 6 compares our proposal with other systems described in the literature, and Section 7 contains our conclusions.

2 Capabilities in a GVMS

2.1 Password Capabilities

Capabilities represent a location-independent object name, and are therefore ideally suited to a GVMS. There are three kinds of capabilities [7]: *Tagged* capabilities are distinguished from normal data by system-maintained hardware memory tags. Since this involves specialised hardware, they were not considered suitable for our system. *Partitioned* capabilities are kept in protected segments that cannot be manipulated by user programs. As we do not wish to impose any restrictions on the use of pointers by application programs, we have chosen the third alternative: *sparse* capabilities. These are simply long bit strings which are protected from forgery by the fact that only a very small number of all possible strings are valid capabilities.

We have opted for password capabilities, rather that other sparse capability schemes which require encryption [7], making creation and validation of capabilities expensive. Our capabilities consist of two parts: a 64-bit address and an (at least) 64-bit password. Therefore there are two kinds of pointers in our system: plain 64-bit addresses, and capabilities which contain an address as well as the corresponding password. Both kinds of pointers can be freely passed around by users, and can be stored in any user-level data structure. The advantage of allowing capabilities (as pointers) to reside in user-level data structures has been pointed out by Jones [8].

To make the system easy to use, the protection system aims to be as unintrusive as possible. In the normal case of a plain address being used for accessing memory, a capability must somehow be presented to the system so that it can validate the access. In particular, as long as a process is only accessing its "own" objects, it should not have to worry at all about protection. The system should somehow automatically recognise the user's *protection domain* (PD) which, in our system, is the set of all capabilities a user holds.

An explicit open operation would restrict the need for presenting capabilities to the first access only. Unlike traditional approaches, however, an open operation is not required in a GVMS, as there is no need to instruct the system to map files into the user's address-space. An *implicit* open operation can therefore be used. This requires, however, that the system can find the associated password when the address is first presented. The data structures to facilitate finding passwords are described in Sections 3 and 4.

2.2 Capability Types and Access Modes

Protection in our system is object-based; i.e., a capability always defines access rights for a whole object. The system view of objects is simply that of a contiguous, page-aligned segment of virtual memory. Any further structure on objects is the responsibility of higher software layers.

For this low-level protection, we propose four access modes: read (r), write (w), execute (x),[1] and destroy (d). Every capability permitting all the four basic access modes is called an *owner capability*. Whenever an object is created, i.e. virtual memory is allocated, the system returns an owner capability to the creator process. Its password is a random number generated

[1] There is a special form of execute mode, called *protection-domain-extension*, which is explained in Section 5.2.

by the system. Note that there does not exist a *unique* owner process or user for each object; *any* process which presents an owner capability to the system has permission to perform any operation on the object.

When an object is created, the system records its *base address*, *length*, and owner password in a global data structure called the *object table* (OT). The OT obviously contains sensitive data that must be protected from any access by user programs. The capability to read the OT is our system's equivalent to the root password in UNIX systems, and so must be protected.

2.3 Derived Capabilities

As well as owner capabilities, the system provides capabilities with more restricted access rights, such as read-only. A scheme is provided which allows users to derive less powerful capabilities as required. This method is similar to one proposed for Amoeba [9].

From the owner capability, C_{rwxd}, a new capability $C_{rwx} = f(C_{rwxd})$, where f is a well-known one-way function, can be derived which only gives permission to read, write and execute the object. That capability can be further restricted to $C_x = f_x(C_{rwx})$, which allows only execution, and $C_{rw} = f_{rw}(C_{rwx})$, which allows only reading and writing. f_{rw} and f_x are related one-way functions, e.g. f with a constant string XOR-ed with its argument: $f_{rw}(s) = f(s_{rw} \oplus s)$, $f_x(s) = f(s_x \oplus s)$. The former capability can be further restricted to $C_r = f(C_{rw})$, which only allows reading. The capability hierarchy is shown in Figure 1.

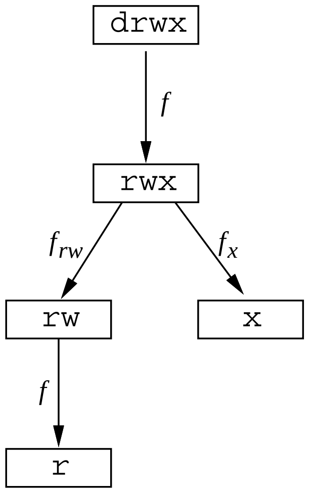

Figure 1: Hierarchy of derived capabilities

When an object is created, the system derives a full set of capabilities from the randomly generated owner password. All five passwords of these capabilities, together with their corresponding permission bits, are stored in the OT. Any process holding a valid capability to an object can then derive a weaker (more restricted) capability to the object by applying the well-known one-way function.

In addition to directly deriving less powerful capabilities from existing powerful ones, the owner (that is, any process holding an “owner” capability) can also create new capabilities for an object. To do this, the process provides a new password and the corresponding access rights to the system, which will record the new password in the OT, together with all of its derived passwords. Similarly, the owner can ask the system to remove certain passwords from the OT, thus selectively revoking access rights to the corresponding object. The owner can also obtain a complete list of valid capabilities from the system.

We restrict the addition of capabilities to owners, to ensure that they have full control over objects. If non-owners were allowed to add capabilities, an owner could not reliably revoke access to the object.

3 Capability Tree and Protection Domains

Data structures are needed which allow users to store and manipulate their capabilities, as well as allowing the system to quickly find capabilities after a protection fault. These data structures should also reflect the user’s intuitive view of the protection model.

Most existing operating systems support a hierarchical model of protection. In UNIX, for example, files have three sets of permissions, which govern access by the owner, the owner’s group, and everybody else. This hierarchy seems to map well to actual use, as a system typically has files which are needed by everyone (e.g. programs in /bin), those which are relevant to a particular group of users (e.g. project-related programs and documents), and those which are private.

3.1 The Capability Tree

To support a similar hierarchy of protection, we organise our capability store as a tree, the *capability tree* (Ctree), which is a single object shared by all kernels in the system. A node of this tree is called a *protection node* (Pnode) and is linked to a group of capabilities.

Capabilities for the most public objects are stored at the root node of the Ctree, while private capabilities are stored at the leaves. Group capabilities are stored in Pnodes at intermediate levels. When the system searches the tree, it will search from some Pnode all the way to the root, thus encountering the more private capabilities first. We call the set of capabilities encountered when traversing the Ctree from a particular Pnode to the root a *regular protection domain* (RPD). Any pointer to a Pnode defines an RPD. Note that RPDs are not strictly hierarchical, as the same capability can be stored at different Pnodes.

Each Pnode contains a pointer to a *capability list* (Clist). While the Ctree is a system data structure, the Clists are completely under user control. Every user holding a valid capability can modify a Clist by adding or removing capabilities, thus modifying the corresponding RPDs. Of course, normal users will only hold capabilities to their private Clists (belonging to Pnodes distant from the root), while only the system administrator will hold capabilities for the root Pnode’s Clist.

Each Pnode may also contain a pointer to a user-provided *protection fault handler* which provides an alternative search strategy. This is discussed further in Section 4.1

Figure 2 shows the structure of the capability tree.

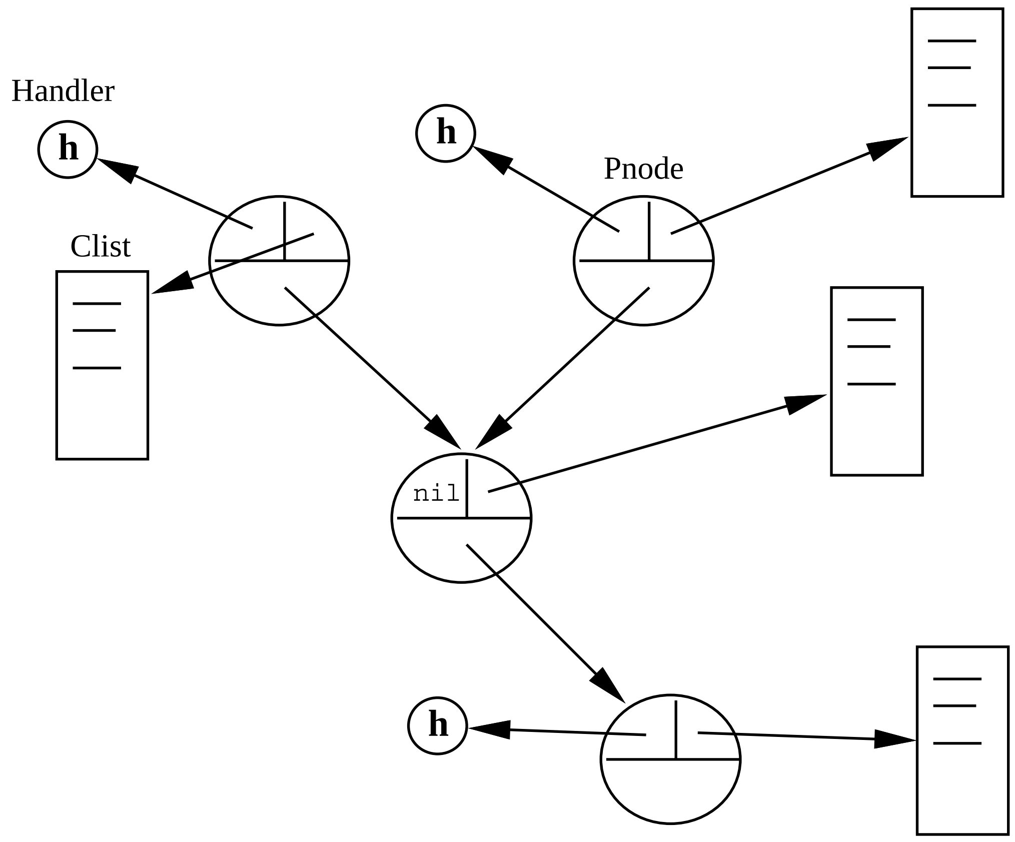

Figure 2: The capability tree

3.2 Operations on the Capability Tree

Each user of the system has an associated RPD, which is defined by a pointer stored in the system's user directory and which points to the appropriate Pnode. When a user is added or removed from the system, the system administrator adds or deletes a node in the Ctree. This requires that the system administrator holds the capability to execute maintenance routines which themselves hold the capability to the Pnodes.

Users are also allowed to add Pnodes to the Ctree. To do this, the user provides a capability for a new Clist, and a pointer to an existing Pnode which becomes the parent of the new Pnode. To perform this operation, the user must have read permission on the parent's Clist. Similarly, users may delete a Pnode if they hold a write capability to the node's Clist.

4 Active Protection Domains

While the Ctree reflects the hierarchy of protection that users might expect, it is not necessarily the most efficient or flexible way of maintaining protection domains. One problem is that processes cannot modify their protection domain, e.g. to restrict it before calling untrusted programs, without affecting the protection domains of other processes. Another problem is that it is difficult to determine how many pointers exist for a given Pnode, which makes it impossible to reclaim unreferenced Pnodes, and so the Ctree may grow indefinitely. This is particularly a problem with processes that terminate abnormally without removing any Pnodes have they added to the Ctree.

To overcome these problems, we introduce an *active protection domain* (APD), which is a data structure defining the protection domain in which a process is executing. APDs are similar to *local name spaces* in HYDRA [10], and *process resource lists* in CAP [11], and consist of an array of Clist and protection fault handler pointers held in the PCB. When a user logs into the system, the login process' APD is initialised from the user's RPD by traversing the Ctree, starting at the Pnode pointer found in the user directory, and copying all Clist and protection fault handler pointers into the APD. Subsequently, the user process is free to modify its APD by adding or removing Clists. Figure 3 shows the relation between the Ctree and an APD. Note that not all Clists need to appear in the Ctree.

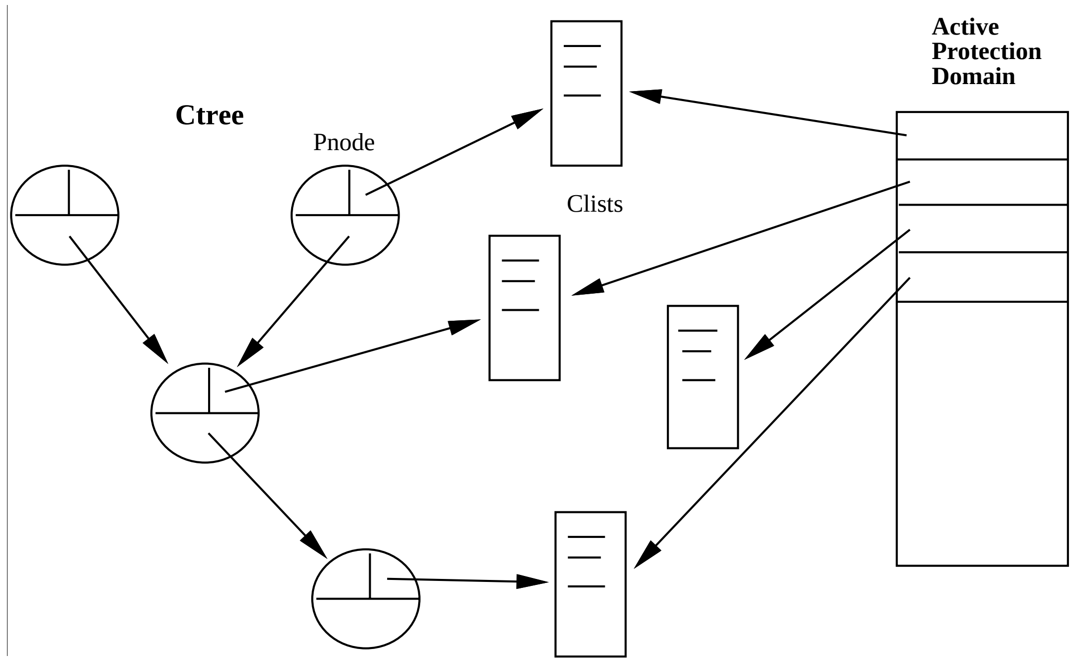

Figure 3: Capability tree and active protection domain

4.1 Access Validation

Since Clists are user data structures, the system cannot rely on them to contain valid capabilities. They must be validated by matching them against the passwords stored in the OT to determine their valid access modes.

When an object is accessed for the first time, a protection fault is generated, as the system has no information on the validity of the access. The system then consults the OT to obtain the base address of the object containing the faulting address, as well as the passwords and access modes belonging to that object. Next, the process' APD is searched for a matching capability granting at least the required access mode. If the validation is successful, the protection and translation tables described below are updated and the access is allowed to proceed.

To give users more freedom in the organisation of capabilities, we allow them to provide *protection fault handlers* (PFHs), which are upcalled by the kernel while searching the APD. A PFH is given the faulting address and the desired access mode and will return a capability to be validated, or a failure indication which causes the search to continue. This allows users to implement faster lookup schemes such as hashing.

For performance reasons, it is essential that the system can search a process' APD quickly when validating a memory access. We therefore use a fixed-size list in the PCB which can hold up to 16 Clist and handler pointers. This size is based on the experience of the CAP project [11], which also used pointers to capability segments, and which found that six were sufficient (even though there was space for 16). As usage patterns in our system are likely to be somewhat different, we allow for a larger list than what was found to be sufficient in CAP.

4.2 Caching Access Rights

It is of course impossible to perform this validation on every memory access, so it is essential that validation information is cached. The obvious way to do this is by using the page table. Once an access to an object has been validated, the page table entries will reflect the access right(s) found with the matching password.

This scheme does not work well with conventional hardware page tables. It requires each process to have a separate page table, even though all page table entries for shared pages contain the same translation information in the GVMS' single address-space, and differ only in their protection bits. This makes it difficult to maintain the consistency of the page table entries.

A further complication is that translation information cached in the translation lookaside buffer (TLB) may have to be invalidated on a process switch, as processes sharing a page may have different rights to it.[2] However, some of the translation information in the TLB may still be relevant to the new process. Hence the TLB entries will have to be invalidated even though their contents are correct except for a few bits, or the TLB needs to be tagged with the process ID.

A solution to the problem of multiple page tables is to use a software-loaded TLB. The system can now maintain a single translation table and separate per-process protection tables. On a TLB miss, the data from the translation table and the current protection table are merged

[2]In the common case of read-only or execute-only sharing of system objects, the access modes are likely to be the same.

to reload the TLB. On each process switch the system changes to a different protection table. This approach still requires invalidating some TLB entries, however.

An alternative solution is based on the idea of completely separating the hardware support mechanisms for translation and protection. The TLB then contains the usual translation information, but no access rights bits. In addition, there is a *protection lookaside buffer* (PLB) which caches process-specific validation data.[3] On each access, the TLB and the PLB are searched in parallel. A successful TLB search returns the physical address, an unsuccessful one generates a *translation fault*. The PLB search generates a *protection fault* if unsuccessful. The TLB is thus completely process-independent and does not need to be flushed at all. The PLB does not need to be flushed if it is tagged with a process ID. The PLB can also be smaller if it is object-based rather than page-based. Recent work suggests that such a device could be feasible [13].

5 Changing Protection Domains

5.1 Tailoring Protection Domains

Enlarging an APD by adding new capabilities is necessary to allow processes to create and share objects. Reducing an APD is essential in order to safely deal with untrusted programs: a user should be able to execute an untrusted program in a PD which contains only those capabilities the program needs to perform its duties. Both of these situations require services to initialise and modify APDs.

When a process is created, the parent provides a pointer to an initial APD data structure. This structure, for example, could be obtained from an RPD using a standard service routine, or could be specially constructed by the parent. A common case would be for the child to inherit the parent's APD.

APD modifications can be performed in one of two ways. A process may, provided it holds the appropriate capabilities, modify the actual Clists pointed to by the APD list in its PCB. Such a change will, of course, influence all processes whose APDs contains those Clists. Alternatively, system calls are provided to allow the process to modify the array of Clist and handler pointers in its PCB.

As an example of these mechanisms in operation, consider the case of executing an untrusted program. The parent process creates a buffer which will be used to pass data to and from the program. The parent then creates a new process whose APD only contains capabilities to access the buffer and to execute the untrusted program. The child process executes the untrusted program in this restricted environment, while the parent waits for the job to terminate.

5.2 Temporary Extension of Protection Domains

In UNIX systems it is possible to substitute temporarily one protection domain by another using the *set-uid* mechanism. This is extremely useful to let normal users perform special operations in a controlled manner via a privileged program. We propose a special kind of procedure, the *PDX* (protection domain extension) procedure, to perform a similar task in the GVMS.

[3] A similar approach has been proposed by Koldinger et al. [12]

Each time a user process attempts to call a PDX procedure, the system consults the object table in the usual manner to validate the execute capability for the procedure. The system then discovers that the object is a PDX procedure, which has an associated list of valid entry points and a single Clist pointer. If the call attempted by the user is consistent with the entry-point list, the system pushes the Clist pointer on the caller's APD and, prior to transferring control to the procedure, swaps the stacked return address with a dummy value. When the protected procedure returns, the dummy return address will cause a protection fault, which is caught by the kernel. The kernel restores the original APD and lets the user program resume execution from the original point of call.

The PDX mechanism is transparent to the caller, who does not need to know of the special status of the procedure. Restricting PDX access to controlled entry-points avoids security problems such as entering a procedure after its validation code. The ability to associate different sets of entry points with each PDX capability allows selective access control, similar to methods in object-oriented systems. Note that, unlike the UNIX set-uid facility, the PDX mechanism does allow access to the caller's environment. However, the caller can tailor a confined protection domain before calling the procedure if necessary.

6 Comparison

In this section we contrast our proposal to other approaches to protection in distributed virtual memory systems. Other GVM-like systems have recently been proposed, but these have either ignored protection [14, 15, 16], or have failed to provide sufficient details of its operation [17].

6.1 MONADS

The MONADS project has long recognised the value of a global address-space for the support of persistence. The system was designed to provide strong support for software engineering principles, such as modularisation and encapsulation. The MONADS protection model is a reflection of this fact. The system ensures that modules can only be accessed via well-defined entry points, and that the internal structure remains inaccessible otherwise. Besides modules (large-grain objects), MONADS also supports fine-grained objects within a module. The two kinds of objects are supported by different access and protection mechanisms, a reflection of differing usage patterns [18].

The main disadvantage of MONADS is that it is based on a specialised architecture, and so cannot easily make use of advances in processor design. Porting MONADS to a SPARC, for example, required the development of several items of customised hardware, which was a major investment in effort [19]. Our design is based on a conventional architecture, though it is most suited to machines with a software-loaded TLB.

Furthermore we believe that the address-space of a general-purpose workstation operating system should be flat (unlike MONADS use of structured addresses), to minimise the restrictions on the implementation of higher software layers. We also do not want to impose limitations on the storage of capabilities, whereas MONADS uses partitioned capabilities which are kept in user-inaccessible system areas.

6.2 Amoeba

Although not a distributed virtual memory system, Amoeba [9] uses sparse capabilities, consisting of the port number of the server responsible for the object, an object id, access rights, and a signature. The signature is computed by applying a one-way function to the access rights and a random number which is stored with the object. Capabilities are always presented explicitly as part of a typical client-server interaction. Less powerful capabilities can be derived with the help of one-way functions, but selective revocation of access rights is not possible, as only one "password" exists for each object.

Full validation of a capability (by encrypting the password and access rights and comparing this with the presented capability) is required each time the server receives a request for an operation on an object. This is acceptable in Amoeba's environment but is completely unfeasible in a GVMS. Furthermore, Amoeba has protected directory servers, which contain capabilities. In our system we have completely separated naming from protection by introducing separate data structures. Hence, directory servers do not require special privileges in our system.

6.3 Opal

The protection system in Opal [20] bears some similarity to our model. The Opal system uses a form of password capabilities called *protected pointers* to control access to objects. The protected pointers also contain *portal numbers* which are used in cross-domain procedure calls. As well, the Opal group is investigating hardware mechanisms to support the separation of translation and protection [12].

Opal also provides two methods of validating access to an object. The first is by explicitly presenting a capability to the `attach` system call. Attempts to access unattached objects cause a protection fault and the system then attempts to validate the access implicitly. It is not clear, however, what role protected pointers play in implicit validations, or whether an alternative protection method is involved.

Opal's cross-domain procedure calls serve a purpose similar to our PDX procedures. They are, however, different from normal procedure calls in that a special instruction is required and different parameter passing mechanisms are used. Therefore, the two kinds of procedures are not transparent to the caller.

7 Conclusions

A single address-space is an ideal environment in which to implement persistence: the meaning of object addresses is independent of the existence of the processes which created the objects, so objects in a global virtual memory are automatically persistent.

The single address-space, however, implies that completely different mechanisms must be used to provide protection in the system, unlike traditional systems which are based on the existence of a separate address-space for each process. We have proposed a novel protection system based on password capabilities which addresses this issue.

Our capability system fits well into the single address-space concept in that it gives users full freedom for sharing data, while at the same time effectively preventing unauthorised data access. A system-maintained persistent data structure, the *capability tree*, provides users with a convenient and safe store to keep their capabilities. The capability tree also defines a user's login protection domain. Rapid lookup of capabilities when validating a memory access is achieved using per-process *active protection domains*, which also allow flexible manipulation of the process' protection environment.

Protection is mostly transparent to users, and they do not require a detailed understanding of its operation. There is scope, however, for users who are aware of the underlying mechanism to tailor the security environment in which they execute. In particular, the system allows processes to inherit a protection domain, and to expand (when obtaining new capabilities) or reduce (when calling untrusted programs) their own protection domain. A transparent set-uid-like temporary modification of protection domains is supported. Adding new capabilities to objects is possible, as is selective revocation of access rights.

References

[1] M. Atkinson, P. Bailey, K. Chisholm, P. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26:360–5, 1983.

[2] S. Russell, A. Skea, K. Elphinstone, G. Heiser, K. Burston, I. Gorton, and G. Hellestrand. Distribution + persistence = global virtual memory. In L.-F. Cabrera and E. Jul, editors, *International Workshop on Object Orientation in Operating Systems*, volume 2, pages 96–99, Dourdan, France, 1992. IEEE.

[3] MIPS Computer Systems, Inc., Sunnyvale, CA. *MIPS R4000 Microprocessor User's Manual*, 1st edition, 1991.

[4] Digital Equipment Corp., Maynard, MA. *Alpha Architecture Handbook*, 1992.

[5] G. Heiser, K. Elphinstone, S. Russell, and G. R. Hellestrand. A distributed single address space system supporting persistence. School of Computer Science and Engineering Report 9302, University of NSW, Kensington, NSW, Australia, 2033, March 1993.

[6] M. Anderson, R. Pose, and C. Wallace. A password-capability system. *The Computer Journal*, 29(1):1–8, 1986.

[7] M. Anderson and C. Wallace. Some comments on the implementation of capabilities. *The Australian Computer Journal*, 30(3):122–33, 1988.

[8] A. Jones. Capability architecture revisited. *Operating Systems Review*, 14(3):33–5, 1980.

[9] S. J. Mullender and A. S. Tanenbaum. The design of a capability-based distributed operating system. *The Computer Journal*, 29:289–99, 1986.

[10] E. Cohen and D. Jefferson. Protection in the HYDRA operating system. In *ACM Symposium on OS Principles*, volume 5, pages 141–59, 1975.

[11] R. Needham and R. Walker. The Cambridge CAP computer and its protection system. In *ACM Symposium on OS Principles*, pages 1–10, 1977.

[12] E. J. Koldinger, J. S. Chase, and S. J. Eggers. Architectural support for single address space operating systems. In *Symposium on Architectural Support for Programming Languages and Operating Systems*, volume 5, pages 175–86, 1992.

[13] J. Kaiser and K. Czaja. ACOM: An access control monitor providing protection in persistent object-oriented systems. In *International Workshop on Persistent Object Systems*, volume 5, pages 359–73, Pisa, Italy, 1992. Morgan-Kauffman.

[14] D. Cohn, A. Benerji, P. Greenawalt, M. Casey, and D. Kulkarni. Workstation cooperation through a typed distributed shared memory abstraction. In *Workshop on Workstation Operating Systems* [21], pages 70–4.

[15] M. L. Scott and W. Garrett. Shared memory ought to be commonplace. In *Workshop on Workstation Operating Systems* [21], pages 86–90.

[16] A. Bartoli, S. J. Mullender, and M. van der Valk. Wide-address spaces—exploring the design space. *Operating Systems Review*, 27:11–17, January 1993.

[17] J. B. Carter, A. L. Cox, D. B. Johnson, and W. Zwaenepoel. Distributed operating systems based on a protected global virtual address space. In *Workshop on Workstation Operating Systems* [21], pages 75–9.

[18] J. Rosenberg. Architectural support for persistent object systems. In L.-F. Cabrera, V. Russo, and M. Shapiro, editors, *International Workshop on Object Orientation in Operating Systems*, volume 1, pages 48–60, Palo Alto, USA, 1991. IEEE.

[19] D. Koch and J. Rosenberg. A secure RISC-based architecture supporting data persistence. In J. Rosenberg and J. L. Keedy, editors, *International Workshop on Computer Architectures to Support Security and Persistence of Information*, pages 188–201, Bremen, Germany, 1990. Springer-Verlag.

[20] J. S. Chase, H. M. Levy, E. D. Lazowska, and M. Baker-Harvey. Lightweight shared objects in a 64-bit operating system. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1992.

[21] IEEE. *Workshop on Workstation Operating Systems*, volume 3, Key Biscayne, Florida, 1992.