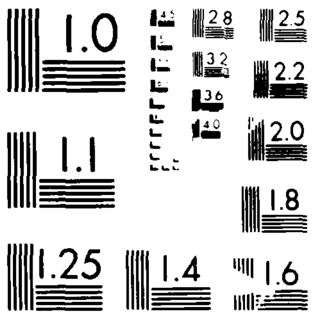
AD-A183 869 THE VARIETIES OF CAPABILITIES IN FLEX(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) I F CURRIE ET AL. APR 87 RSRE-MEMO-4042 DRIC-BR-102668 1/1

UNCLASSIFIED F/G 12/6 NL

END
DATE
FILMED
9 87


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

[illegible]

[illegible] No. [illegible]

ROYAL SIGNALS & RADAR ESTABLISHMENT

THE VARIETIES OF CAPABILITIES IN FLEX

Authors: I F Currie & J M Foster

[illegible] EXECUTIVE,

[illegible]

[illegible]

[illegible]

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 4042

TITLE: THE VARIETIES OF CAPABILITIES IN FLEX

AUTHORS: I.F. Currie and J.M. Foster

DATE: April 1987

SUMMARY

Capabilities in Flex are first class data objects which allow one to define and limit the right to access data or obey an action. Their use extends from mainstores to filestores and across networks of Flexes. This paper gives a general description of how Flex capabilities are implemented, controlled and used. They are classified into four varieties, mainstore, filestore, remote and universal. Each of these varieties has its own range and lifetime designed to combine consistency, integrity and utility with implementability.

1. Introduction

It is common in everyday life for the possession of an object to confer some rights or privileges on the holder. Examples of such objects abound: an airline ticket gives one the right to travel on a particular flight; a cash-point card gives one the right to use an automatic cash-dispensing machine; and, of course, a sufficiency of bank-notes gives the holder all that money can buy. These objects can be regarded as "capabilities" for the rights they confer. Often, the relation between capability and right is one to one; in other words, one has the right if and only if one posseses the capability. Clearly, a capability must be difficult to forge otherwise the value of the right which it represents is debased.

In the computing context, a capability confers the right to obey an action like reading or writing data or running a program. If possessing a capability is necessary to perform an action, we have a good basis for solving many problems associated with the security, privacy and integrity of computer systems as well as a solution to more mundane problems such as the detection and diagnosis of program errors. Just as with non-computer capabilities, the control of the creation and distribution of capabilities is crucial; there is little point in trying to enforce a discipline which is easy to circumvent either by accident or design.

A paradigm for computer capabilities is given by a simple example where the capability allows one to read and write to a contiguous area of store; it might be implementing a vector in a high level language. The data within this area would only be accessible by instructions which made use of the capability to this area. For example, using the notation given in figure 1, an instruction to load the fourth word (say) of the area into a register would have to include the capability cap and the displacement 4 in its operands , perhaps something like:

```
        loadreg 4,capword
```

where capword (perhaps a register) contains the actual capability cap. Of course, this instruction is only legal if the size of the area is greater than four words.

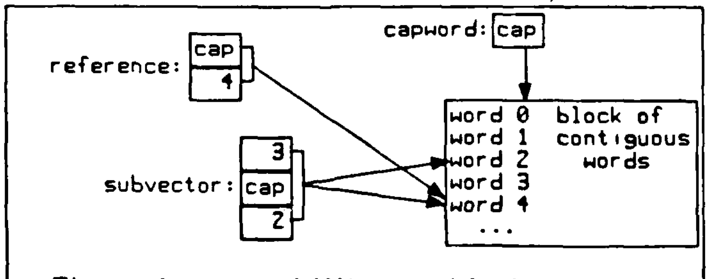


Figure 1 - a capability to a block of memory

In most capability architectures [1,2,3] the control and storage of capabilities to prevent misuse has usually been implemented by making capabilities special objects; only very privileged programs can create them and normal programs can only move them about in a circumscribed manner. In these architectures, capabilities exist in special registers or in special areas of memory; for example, capword in figure 1 would have to be a special purpose register or a word in a block of storage which contains nothing else but capabilities. This implies that one cannot easily hold capabilities and normal data in the same data structure.

Even at the lowest level of programming, one can be embarrassed by this separation of "scalar" data from capabilities. For example, the word-pair labelled reference in figure 1 could be usefully interpreted as a pointer to word 4 of the block whose capability is cap.The inability to store this object in contiguous words places a heavy burden on the programmer in implementing things like reference parameters to procedures. A similarly inconvenient object is the word-triple labelled subvector which might be a representation of a vector of three words starting from word 2 of the block given by cap. Any particular solution to this problem which gives special representations to references or vectors is inadequate since almost any juxtaposition of capabilities and scalars is required to create general data and program structures; these cannot all be anticipated in the initial design of the system.

This "apartheit" of scalars and capabilities is one of the reasons for the lack of success of the above capability machines. The difficulty of finding solutions to problems posed by the representation of arbitrary data structures caused programmers to flatten them out into a small number of blocks so that only a few capabilities had to be handled. This obviates most of the usefulness of capabilities by only giving very coarse-grained discrimination and protection. In addition , the system programmer finds that he cannot construct an object-oriented system because the components of an object like the reference and vector objects given above must be distributed across several blocks in store. In other words, these machines could only be used as rather inconvenient standard machines with few or none of the advantages of capabilities showing through.

Another drawback of these machines was their assumptions about how store was allocated. One assumption made was that compilers would check local objects so that capabilities were only needed for major objects. Thus the creation of new capabilities by allocating new storage would be done in large chunks and would be a fairly rare event, requiring a relatively expensive system call. However, this overlooked the possibility of the transmission of structured data between "programs" compiled independently. It also meant that compilers had to be correct, making the development of new compilers more difficult. To reap the benefits of a capability structure, one would like to use capabilities in as fine grained and dynamic fashion as possible. For example, I wish to implement lists as capabilities so that the cons function produces a new capability to a newly allocated block of two words containing the head and tail values; a long-winded system call to allocate the new block would be intolerable.

The Flex architecture [4, 5, 12] uses capabilities in a manner which does not suffer from these limitations. Capabilities in Flex are first class objects - they can be created by non-privileged programs and can be loaded and stored in just the same way as one loads or stores scalar objects without requiring specialised registers or storage areas. Instead of recognising capabilities by where they are stored, Flex capabilities are distinguishable data objects. For example, in Flex mainstore, each word has an ext a "tag" bit. Scalar words have their tag bits zero, and capabilities are words with set tag bits.

The tag bits are not involved in the normal arithmetic or logical operations in Flex; indeed these operations are only legal between scalars. Otherwise, the tag bits are copied consistently in the loading and storing operations in Flex. Thus, each word containing cap in figure 1 has the tag bit set and each word containing an integer has it cleared. When a word with a tag bit set is produced ab initio, the interpretation of this word is always such that it represents a new capability different from all others. In other words, if you possess a capability, you either created it yourself or else you were given it by somebody else via some other capability - you cannot forge capabilities.

In the implementations of Flex up to the present (March 1986), the capability rules in mainstore have been enforced by micro-coding a Flex instruction set on a micro-programmable machine, the most recent being the ICL Perq workstation [12]. In this instruction set, a new mainstore capability can created by obeying one of a set of unprivileged instructions to allocate the space and set it up depending on the type required. Capabilities other than mainstore ones exist in Flex; their access and creation rules are enforced by a mixture of software (both privileged and unprivileged) and firmware. These capabilities are represented in mainstore by one particular kind of mainstore capability; in addition they possess representations in other media like filestore or networks.

Capabilities as data objects form the basis of the Flex system and appear in many different guises and representations in mainstores, in filestores and across networks. Since one can handle them quite freely and create them to represent arbitrarily complicated objects, the Flex system is an "object oriented" system. It uses and creates objects directly without the need for intermediaries like names, directories or contexts.

Ideally one would wish that, once a unique capability has been created, it should exist as long as is it is required; in other words as long as it is referenced. Practically, this requirement cannot be met - unexpected machine errors over a network are likely to wreck any such scheme. This is not to say that one should abandon the ideal, but rather that one should approach it as closely as possible and ensure that if ever one does "lose" capabilities by machine error, one always has a consistent fall-back position. As a trivial example, a hardware error in a single processor system would cause the loss of those capabilities in the main store which are part of the currently running program. This has no consequential inconsistencies if these capabilities cannot exist outside the main store; however, if they could have been written to file store, say, we would have had references from file store to a non-existent mainstore, which could be disastrous.

In order to control the consistent existence of capabilities, the Flex system classifies them into four main groupings, each with different rules for where they can exist and how, if at all, they can be transmitted. These groups are:

1. Mainstore capabilities
2. Filestore capabilities
3. Remote capabilities
4. Universal capabilities

A mainstore capability exists only in **one** mainstore, cannot be transmitted elsewhere and implements things like arrays and procedures in running programs; it is obviously a rather temporary thing, disappearing when the machine is switched off. A filestore capability retains its meaning from session to session, can exist in **one** filestore or in the mainstore of any processor which can access that filestore directly, can be transmitted to and from this one filestore or between these mainstores and is used to implement things like files, modules, etc. A remote capability can exist in **any** mainstore, can be transmitted between mainstores and is a means of performing an action on some processor from another processor on a network. A universal capability can exist **anywhere** in the Flex world, in any Flex filestore, mainstore or memory and represents a Flex object which is common across all Flex systems, for example, some version of a commonly used compiler.

This classification of capabilities has been derived from experience in the construction and use of various Flex systems. These have included a system with several processors connected to a common file-store and one with disjoint file-stores connected across a network. The remaining sections of the paper highlight some of the important aspects of the varieties of capabilities and their uses in the Flex system.

2. Mainstore capabilities

A mainstore capability exists in only one mainstore and cannot be transmitted elsewhere. If its machine is switched off it will disappear.

All capabilities accessible to a running Flex program are represented by pointers to disjoint blocks of store. A pointer is simply a word with its tag bit set to distinguish it from scalar words. It contains the "address" of a block which contains the size of the block and type of the capability in its first word as shown in Figure 2. The quote symbols are used deliberately here since this notion of an address does not enter further into the Flex architecture. One can access the information in the block only if one has the capability; the actual physical address of the block can change and is both useless and irrelevant. These capabilities are unforgeable in the sense that one cannot create a word with the tag bit set which is the same as another except by copying that word. When one creates a new capability it is guaranteed to be different from all others.

One of the most basic kinds of mainstore capability allows one to read and write words into memory. There are several instructions in the Flex repertoire which allow one to create new capabilities of this memory type. For example, there is an instruction which returns a capability to read and write a new block of given size; another allows one to pack away a value consisting of some number of words into a new block and deliver a capability to read and write into that block. The read/write capability in figure 2 could have been created by one of these instructions, and the read-only capability could only have been created by another instruction from this read/write capability. Other instructions allow one to read or write words (any mixture of scalars and capabilities) in the block via the read/write capability while only allowing reading via the read-only capability.

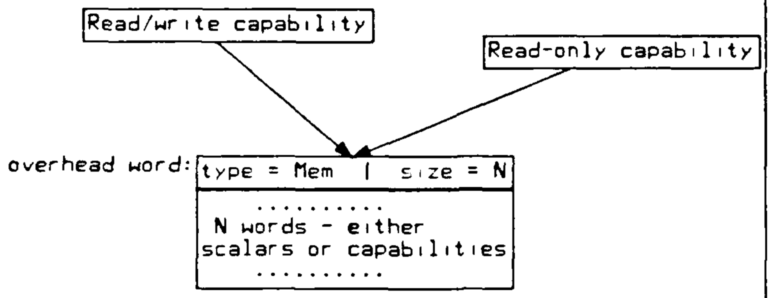


Figure 2 - memory capabilities

The access rules for memory blocks are just one example of the kind of rights and restrictions conferred by capabilities. The algorithms defining these rules are very simple - one can only read and write within the bounds of block defined by a memory capability. Clearly, one could imagine other algorithms defining other access rules, and Flex does have other kinds of capabilities with other fixed access rules. However, Flex also allows one to create capabilities where the algorithm is chosen by the programmer by using the most general kind of mainstore capability,the procedure.

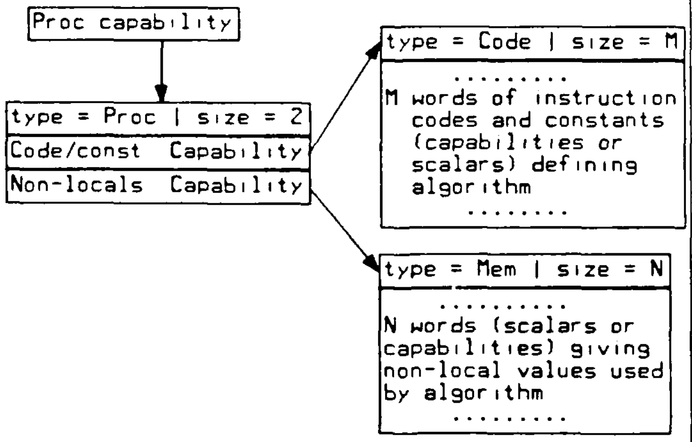
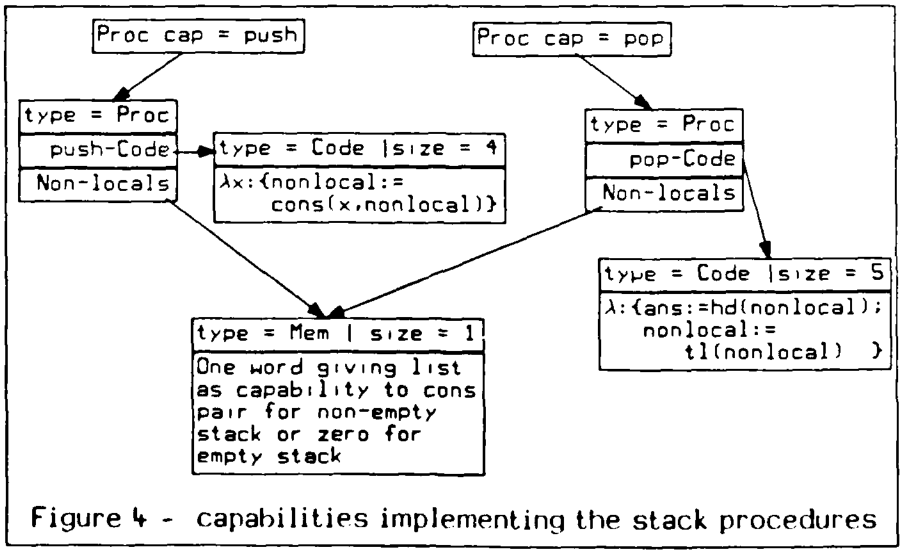


Figure 3 - a procedure capability

The procedures are just Landin's closures [6] , and using them, any arbitrary set of rights or restrictions can be implemented. Such a closure is created by an instruction which binds Flex code with values which form the non-locals of the procedure; both code and non-locals being themselves capabilities as in figure 3. On calling the procedure, the code in the code block can access the non-locals implicitly; for example, there is an instruction to load the nth word of the current non-local block without having to extract the capability explicitly. Two other areas are similarly accessible, namely the locals and the constants (in the code block) of the procedure. The locals block contains the local variables of the procedure and link information; it is produced by the call instruction either by generating it afresh or else by retrieving one which is finished with by a previous call. Note that the same code capability can be shared between many procedures; indeed code blocks are loaded in such a way so that there is never any more than one copy of a code block at any time in mainstore regardless of how many programs are using it.

The possession of a procedure capability does not allow one to read either its code or non-local values; one can only obey the code with these non-local values bound to it. This means that the code can control the kinds of access that are possible to the non-local values without the user of the procedure being aware of their representation or even of their existence. For example, it is trivial to construct a pair of procedures, push and pop, which implement the classical stack operations by sharing the same non-locals as in figure 4. Here the underlying data structure which contains the values in the stack is in fact a list but is completely hidden and is impossible to access in any way other than by calling the procedures. Procedures defining abstract data types or other kinds of packages are usually implemented in Flex sharing non-locals like this.


Proc cap = push
Proc cap = pop
type = Proc
push-Code
Non-locals
type = Code | size = 4
λx:{nonlocal:= cons(x,nonlocal)}
type = Proc
pop-Code
Non-locals
type = Code | size = 5
λ:{ans:=hd(nonlocal); nonlocal:= tl(nonlocal) }
type = Mem | size = 1
One word giving list as capability to cons pair for non-empty stack or zero for empty stack
Figure 4 - capabilities implementing the stack procedures

It is important to note that the creation of these procedures for abstract data types is essentially a dynamic process. For example, the natural way in Flex to construct the stack procedures in figure 4 would be to have another procedure make_stack, say, which when called would deliver a pair of procedures to implement a new stack, different from all others. Each pair would have different non-local blocks, but all of them would have the same code blocks given by the capabilities push-Code and pop-Code ; these capabilities would probably be found in the constants of the code block in the procedure make_stack which simply closes them with a newly generated non-locals block to give the two stack procedures. Once created, the push and pop procedure capabilities are completely independent of make_stack and can exist even when make_stack disappears.

An interesting kind of procedure capability is one which has type:

```
      (Key,Info) → (Key→Info)
```

and body:

```
      λ k,ι:{ λx:{if x=k then ι else FAIL fi} }
```

ie given a key and some information, it produces another procedure which will give back the information if and only if the parameter of its call is the same as the key. Since the key could be a capability and since capabilities are unforgeable, this gives a completely safe way of passing around sensitive information. Only those procedures which possess the key (probably in their non-locals) will be ableto get at the information. Thus, the information can be transferred safely between trusted procedures via an untrustworthy intermediary. This concept is used in representing the other varieties of capabilities (namely filestore, remote and universal) in mainstore, so that they can be held by untrustworthy program without fear of compromising their access rules.This use is so important that it has been particularised to form another type of capability called a "keyed" capability. This resembles a simple memory capability except that it can be locked so that its contents of the corresponding block are completely inaccessible. The only way to unlock the block is by knowing the contents of the first word of the block; thus the first word is the key and the remainder of the block is the information.

To reap the maximum benefit from its use, the lifetime of a mainstore capability is at least as long as it is required; i.e.,if one possesses a capability, it must be alive. In turn, this implies that a block of physical mainstore can only be reused if there is no capability which points to it. To discover this, it is necessary to do a general trace and garbage collection when the physical limits of the mainstore are reached. In the current implementations of Flex, this is done in the micro-code, as are all the other manipulations of physical addresses to produce capabilities. As mentioned before, a mainstore capability can only exist in one mainstore; one cannot transmit mainstore capabilities to other mainstores or filestores. This, together with the use of the tag bits to distinguish between scalars and capabilities, allows the use of a fast garbage collection algorithm which is linear in time in all of its variables.

The Flex instruction code which is interpreted by the micro-code clearly takes full advantage of the properties of the capabilities which it creates and manipulates. It has also been designed for ease of use by high level languages. There is no primitive level assembler for Flex; all of the programs written for and running in Flex have been compiled from some high level language. The instruction set allows one to produce compact code; for example, there are only 7 instructions (including the procedure-exit instruction) occupying 13 bytes in a straightforward, unoptimised translation of pop_Code given in figure 4. Leaving aside the procedure-call and -exit instructions, the obeyed code would involve about 12 memory accesses including the instruction fetches in the Perq implementation of Flex, each access taking 750ns for a 32 bit word [the Perq is actually a 16 bit word architecture, but 32 bit store access has no extra penalties on even word boundaries]. These accesses effectively define the time required to obey the instructions, since other actions required to be performed by these instructions (for example, checking the access rules) are mostly hidden behind the operand fetches. The code in push_Code is slightly smaller in size and also in time, provided that the cons operation does not provoke a garbage collection. The procedure-call instruction might also provoke a garbage collection if there was no workspace available for the locals of the procedure call. If workspace was available, then it takes 10 to 12 memory accesses to obey this instruction which deals with the link and sets up the new local areas; the procedure-exit instruction taking much the same time to do the inverse. In summary, the time taken by the pair of procedure calls given by the somewhat nugatory expression, push(pop), is about 65 store accesses, provided no garbage collection takes place.

The time taken by garbage collection obviously depends on the mix of blocks and capabilities in store at the time of garbage collection. Some blocks never have capabilities in them (for example, the block defining the raster display in the PerqFlex), while others could be filled with them. The PerqFlex garbage collector is a compacting one and all other Flex processing stops while it is active; of course, some interrupt proccessing and data transfers ,including keyboard interactions, continue at a lower level. The time taken by this garbage collector is given approximately by:

$(2*(L-F) + V + 5*C + 2*B + 4*A)$ memory accesses

where L is the total number of live words, V is the number of these which could be capabilities and C is the number which are capabilities. B is the total number of blocks before garbage collection and A the number of live blocks. F is the number of words which are not moved in the garbage collection, either because they are always required at the bottom of memory (eg the raster image) or simply because no free space is recovered below them. In a 2 Mbyte Perq, the actual time taken by one garbage collection averages out to about 1.3 ± 0.2 secs in typical use. Such a use might be running two long Algol68 compilations in parallel with normal text editing; in this case garbage collections occur about once every 45 secs, each collection recovering an average of 1.2 Mbyte of free store. These figures do not change appreciably when running three or even four compilations in parallel since the code is shared between the processes and this compiler (not originally targetted for Flex) is profligate in its use of temporary storage compared with the sizes of more permanent tables that it needs to maintain in mainstore across one compilation.

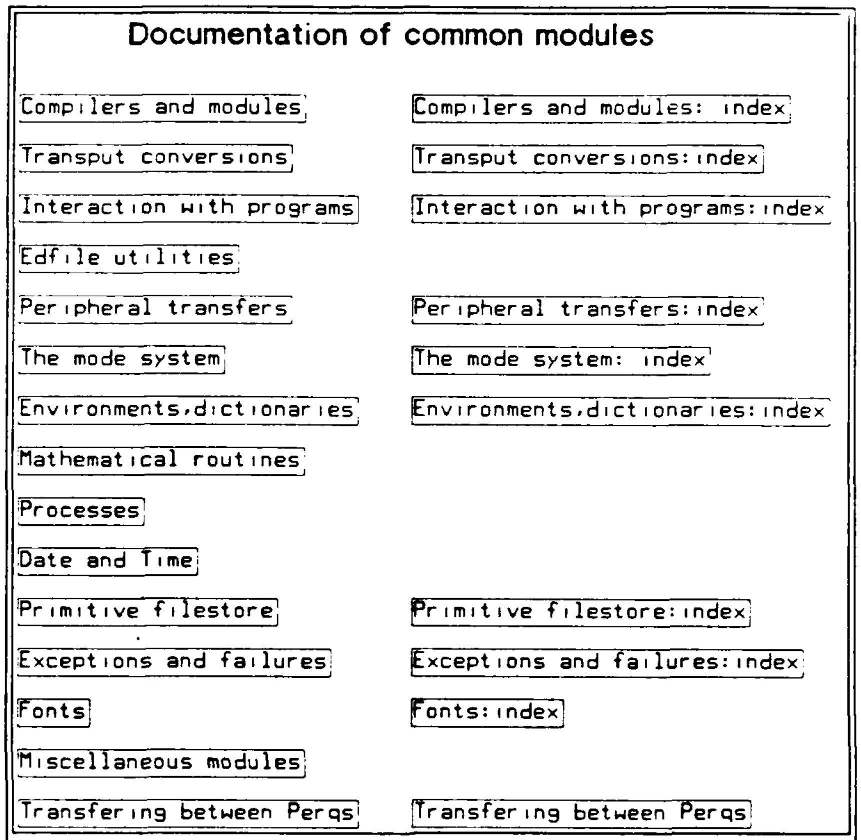
3. Filestore capabilities

A filestore capability can exist in only one filestore, or in the mainstore of any computer which directly accesses that filestore. It can be transmitted between mainstores and the filestore. It retains its meaning from session to session. The word "filestore" is used here for want of a better term. Its use carries no implication of the properties of existing filing systems, but simply defines memory which persists in some permanent form. Other terms such as "data-base" or "persistent heap" might equally well be used but would carry just as many unwanted connotations.

In analogy to mainstore capabilities, a filestore capability is a "pointer" to a block of data on a particular filestore. This data can include any scalars, any universal capabilities, and any other filestore capabilities belonging to the same filestore. Note that cross-filestore capabilities are not allowed; any filestore capability in a filestore "points" to another block in the same filestore. In mainstore, the minimum size of a block is one word (consisting of just the overhead word). On filestore the granularity is bigger depending on the implementation involved; in PerqFlex the minimum size of block is 32 bytes. Filestore capabilities have types similar to mainstore capabilities; the procedures which read the data corresponding to a filestore capability do so by producing a mainstore capability of the same type. In particular, filestore procedures can only be read to produce mainstore procedures. This means that private information can be safely hidden behind the interface of a filed procedure; for example, logging on to Flex is done simply by calling a filestore procedure which has things like passwords and dictionaries safely hidden in its non-locals.

As mentioned previously, a filestore capability is represented in mainstore by a keyed capability; its corresponding block contains information on how to retrieve the data. The key to this locked block is a characteristic of the filestore and the basic outputting instructions and procedures will not allow it to be transferred to an alien filestore. On the filestore itself, or in any transmission medium in networks, non-mainstore capabilities are distinguished from scalars in much the same way as mainstore capabilities, using extra tag bits or bytes; in fact, the total size of a PerqFlex filestore capability on filestore is 12 bytes which are recognisably different from non-capabilities. When inputting these capabilities into mainstore, care is taken that only one copy of the block corresponding to a particular capability occurs in each mainstore, that is, all the copies of a mainstore representation of a non-mainstore capability point to the same keyed block. The search to ensure this is fast and economic - it takes an average of 50 store cycles to establish that the capability is a new one and obviously less to find it if it is not. Filestore capabilities are not rare objects in the Flex mainstore; on average there are about 2000 alive at any one time, a large proportion arising from the fact that most mainstore code blocks have an equivalent on filestore. The uniqueness property is used mainly as an aid to short-circuit the traffic to and from filestore in the case of filestore capabilities; the implementation and use of remote and universal capabilities is more critically dependent on it.

A standard filestore capability is created by writing data of appropriate type to a filestore and receiving in return the filestore capability to read that data. Note that there is no notion here of writing to a particular place on filestore - it is a "write-once" operation. Since the data written away can include other capabilities, one can form non-circular tree structures (strictly speaking, acyclic graphs) of arbitrary complexity on filestore. This tends to mean that the system is quite economical about the amount of traffic to and from the filestore. For example, an editable file is implemented on Flex as an object of an abstract data type called an Edfile; procedures associated with this abstract type include an editor of type Edfile → Edfile and a lister of type Edfile → Void. The representation of each Edfile is a single filestore capability to a block which contains other values, including capabilities, as well as text. Figure 5 shows the screen representation given by the editor to an important file in the PerqFlex system. This particular example is rather short of plain text; most of it displays non-textual values. Each of the boxes, eg $\boxed{\text{Mathematical routines}}$, is the screen representation of a non-textual value in the file; the text in the box, namely "Mathematical routines", is just a convenient label or banner for the value. In fact, each of the non-textual values illustrated here happens to be a disc capability which is an Edfile containing further information, usually including yet more Edfiles and the banners for them are indeed unique in this file [see the treatment of this file as a universal capability]. The contents of any inner Edfile can be displayed by calling the editor recursively by pointing at it and pressing the appropriate key. A listing of the file shown in figure 5, including the expansion of its inner Edfiles covers 125 A4 pages, yet any part of it can be usefully reached by no more than five or six key-presses following the tree structure of the Edfiles, transferring no more than the equivalent of six pages from filestore.



Documentation of common modules

Compilers and modules	Compilers and modules: index
Transput conversions	Transput conversions:index
Interaction with programs	Interaction with programs:index
Edfile utilities	
Peripheral transfers	Peripheral transfers:index
The mode system	The mode system: index
Environments,dictionaries	Environments,dictionaries:index
Mathematical routines	
Processes	
Date and Time	
Primitive filestore	Primitive filestore:index
Exceptions and failures	Exceptions and failures:index
Fonts	Fonts:index
Miscellaneous modules	
Transfering between Perqs	Transfering between Perqs

Figure 5 - an example of an Edfile

By the same token, producing a new Edfile only involves writing away those inner Edfiles which actually change. When one exits from a call of the editor which actually makes some changes to the data rather than simply displaying it, the new data is written away to create and deliver a new Edfile; if it happened to be an inner call (on Mathematical routines, say) then this Edfile replaces the old value in the display using the same banner. The original file given by the parameter of the call has not been altered in any way; it is still available by the same method as was used to get it in the first place. Given the result of the outer call of an edit, committing the change is usually done by giving a name to the new Edfile in some dictionary; it could be the same name as used to get the parameter of the call to give a later version of the file.

The non-textual values in an Edfile are not necessarily other Edfiles; indeed the file illustrated in figure 5 is principally intended to contain values (and descriptions) of another abstract type called Module. For example, part of the internal file given by Mathematical routines is shown in figure 6; the boxes here are screen representations of these Module values. A Module is, in fact, several filestore capabilities which, when operated on by appropriate procedures, give the text (as an Edfile), interface specification and compiled code of some program; eg exp :Module gives the exponent routine as a procedure with a real parameter and real result. As it happens this routine was written in Algol68; however the Module is language independent and can be used by other languages. To include the interface entities of a module within a program, one usually puts the Module value itself into the text of the program, rather than its name; ie a program text with exp :Module in its "use-list" would be able to use the exponent routine in the normal manner.The advantage of hav g the module value rather than its name here is that the program text now effectively includes the texts of all of the modules which it uses, independently of context, so it can be examined once again in a tree-like fashion. Further, since the text and compiled code are bound closely together in a Module, there is never any confusion about the text of a compiled program, even at run time. There are approximately 500 Module values in the file shown in figure 5 (with a total of about 50000 lines of text) accessible for reuse by any programmer in any program. These range from the simple mathematical routines given in figure 6 to things which form part of the system like the compilers, editor and command interpreter. The creator of a Module value has the capability to amend it (by recompiling a corrected text, say). This is another committal operation, this time expressed as an operation on a value rather than by inserting a name in a dictionary.

Standard mathematical functions:

Module	Possible failures
arccos :Module	(8000,3) if ABS parameter > 1.0 Result in range [0.0, pi]
arccosh :Module	(8000,6) if parameter < 1.0
arccot :Module	None. Result in range [0.0, pi]
arccoth :Module	(8000,8) if ABS parameter <= 1.0
arcsin :Module	(8000,3) if ABS parameter > 1.0 Result in range [-pi/2, pi/2]
arcsinh :Module	None
arctan :Module	None. Result in range [-pi/2, pi/2]
arctanh :Module	(8000,7) if ABS parameter >= 1.0
arg :Module	None. Result in range (-pi, pi]
cos :Module	(8000,4) if ABS parameter > 2.0 ** 47
cosh :Module	Real overflow (0,16) if ABS parameter > 709.7 approx
cot :Module	(8000,5) if ABS parameter > 2.0 ** 47 Real overflow (0,16) if parameter is near multiple of pi
coth :Module	Real overflow (0,16) if ABS parameter < 1.12e-308 approx
exp :Module	Real overflow (0,16) if exp too big (ie if parameter > 709.0 approx)

Figure 6 - part of Mathematical routines

The various committal actions for remembering changes to things like files and modules clearly involves some way of overwriting filestore; there has to be somewhere where we can record the state of the filestore, at least when the machine is switched off. This is done in a Flex filestore by having a small number of root variables in the filestore which contain capabilities which allow one to reach all of the accessible filestore. A root variable can contain a single filestore capability; it can be read to give its contents and its contents can be altered by a single unitary operation. Otherwise, it can be used just like any other filestore capability so far as transferring it to and from filestore is concerned. The filestore is only considered to be different when a root changes and a filestore capability remains alive between sessions so long as it can be reached by some path in the tree starting from a root. Thus, a root variable would usually contain a pointer to some dictionary structures and set of modules for a given operating environment; each different log-in operation is likely to give access to a different root.

It is important that a filestore remains consistent within itself; in other words that it is never left in some state of incomplete updating. For example, let us consider how one updates a dictionary derived by a path through the tree structure on filestore starting from some root. One re-constructs a new dictionary and all the tree structure leading to it on filestore before updating this root. Since the process of updating the root is an unitary operation, we know that the filestore is either in its new state or, perhaps because of some failure on the way, in its old state with the old dictionary. One thing is certain, the dictionary is never part new and part old. Thus, provided that different roots contain independent information, that is, they never require to be updated together, the filestore never gets into an inconsistent state. Of course, this is just the most primitive level of consistency control; higher level controls for simultaneous updating and reading still require to be applied. However, any solution of the higher level problems requires that the lower level problem should be solved.

An extremely useful by-product of this method of organising filestore is that a complete history of consistent states of the filestore is potentially available. Since the only thing that is being overwritten is a single filestore capability in the root, one only has to arrange to remember the successive contents of the root. In the general purpose computing context, complete histories are seldom required and the Flex filestores are generally garbage collected and tidied periodically simply to save storage space. However, in the intervening periods, it is still a great boon to be able trivially to reset a file to the value it had a few days, hours or even minutes ago. On the other hand, in a project environment, the total history might be required for all sorts of reasons to control the project and this could be done in several ways with Flex filestore. Thus, it would be simplest if the total on-line mass storage was big enough to contain all the historical information; if not, one simply keeps off-line copies of the filestore before each garbage collection. In all of the current implementations of Flex the filestore garbage collection is done off-line. The time taken for this garbage collection is roughly proportional to the number of live capabilities in it which can lead to other capabilities; the effect of the other variables is swamped by the time taken to access the disc or discs on which they reside. A PerqFlex filestore on a single 35Mbyte Winchester disc containing about 25000 capabilities, including the standard system, takes 20 minutes to garbage collect, freeing about half the disc; it takes very heavy useage for this to be necessary more than once a week. This is a non-compacting garbage-collector. To compact it the live blocks are sent across a network to another filestore; this takes marginally longer than the non-compacting version.

4. Remote capabilities

Remote capabilities can exist in any mainstore and can be transmitted between mainstores. Mainstore and filestore capabilities allow access to data in local mainstores and filestores respectively; remote capabilities allow access to other mainstores and filestores across a network. Flex uses a remote procedure call [9,10] mechanism for its network; it differs from most other RPC networks in that the possible procedures do not have to be agreed between the machines from the start.

In the current implementations of Flex, the particular type of data held in a mainstore or filestore block associated with a capability is largely irrelevant to its access rules. For example, the access rules for a block containing integers is the same as for one containing floating point numbers; also the type of the parameters and results of a procedure do not effect the kind of checks that are done to ensure that the rules for procedures are obeyed. That is not to say that such checks are never done, but just that they can be done at a higher level of abstraction, for example, within compilers or command interpreters. Even if these checks are done wrongly (because of a bug in a compiler,say) then the integrity and security of the system is not compromised . Thus a small number of different types of capabilities (eg memory, procedure, keyed etc) suffices for mainstore and filestore capabilities. On the other hand, both remote and universal capabilities require to be described and implemented in a much more fine-grained manner using the kind of types found in strongly-typed programming languages. At a primitive level, one can see that a good type description is highly desirable for transfers between computers which use different representations of data (eg in changing floating point format). This use of types arises in the Courier protocol [11]. However, the Flex type structure is much more powerful and allows the transfer of capabilities for dynamically created objects, including procedures. The types of these procedures describe how their parameters and results are to be handled and also makes explicit the high-level protocol of the transactions involved in their calls.

Some of the notation for the high-level type structure in use in Flex has already been introduced. Aside from various primitive types like Real, Int, Char and Void, the "→" symbol indicates a procedure type separating its parameter and result type (eg sin has type Real→Real); structures or records are given by Cartesian products represented by parentheses and commas (eg a complex number might have type (Real,Real)); disjoint Cartesian sums giving unions or variants are represented by the prefix Union on a list of their possibilities. There are other constructors for concrete types which give representations of various other ways of structuring data in an orthogonal manner; the only one of these used here is Vec to describe a vector (eg a string of characters has type Vec Char). The representations given by abstract data types like Edfile and Module mentioned above are defined by the procedures which operate on them and chosen by the inventors of these abstract types. Flex types were originally conceived as part of the Flex command language [7] and were based on the type structure of ML [8].

A remote capability can be constructed to give a unique, token for a value of any type. This token can be transmitted anywhere in the network and always be decoded to give the original value in the node which created it.In practice, most remote capabilities are remote procedures, since the only generally available operation on remote capabilities is the remote procedure call. Access to any data in a remote machine can always be expressed by calling a procedure in that machine; no extra penalties are really involved since there is no question of a network directly "addressing" a mainstore in analogy to local access. For example, suppose that machine A wished to generate a stack in the mainstore of B to allow A to push and pop integers. Expressed in procedural terms this means that A must have a capability to call a make_stack procedure in B which can create two further capabilities, like those in figure 4, which it can give to A; these two new capabilities will themselves to allow calls of push and pop procedures for the new stack. In fact the make_stack procedure in B has type:

```
        Void → (Int → Void, Void → Int)
```

The first procedure of the result pair is the push for the stack and the second is its pop. In order to call this procedure in B, processor A must possess a remote capability which B associated with make_stack. This remote capability has type:

```
        Remote(Void → (Int → Void, Void → Int))
```

At the remote call, the two result procedures will be sent to A as further remote capabilities and these will be transformed in A into procedures of type:

```
        ((Int → Void), (Void → Int))
```

which themselves do remote calls for the push and pop operations.

The transformation of procedures to remote capabilities and back illustrated above is part of the mechanism of how the parameters and results of a remote call are treated. This mechanism is called "flattening"; it transforms structured data into a vector of bytes suitable for transmission across a network. On receipt of this vector, the inverse "unflattening" operation is carried out to reconstruct the data in the remote machine. Some flattening operations will involve the creation of new remote capabilities; these will be transmitted across the network as distinct tokens recognisably different from scalar data in the vector of bytes. This discussion is to a large extent independent of the particular lower level protocols required to send these vectors of bytes around the network. However, it will be seen that the security and integrity of the remote capability mechanism depends on that of the lower level protocols. If the protocols are insecure, logically or physically, then remote capabilities can be forged either by accident or design by sending a suitably constructed vector of bytes. Extra safeguards can be built in at the higher level, but these only can reduce the probability of forgery without actually making it impossible.

The action of calling a remote procedure, rem say in A, consists of flattening the parameter, par, of the call, sending the resulting vector of bytes together with the remote capability as a remote call to the originator of rem, B say, and then waiting for the result. The remote machine B will unflatten the parameter to reconstruct par and apply the local procedure associated with rem to it. The result of this local call will then be flattened and sent back to the waiting caller in A which unflattens it to give the result of the remote call. Thus A does something like this:

```
 unflatten_ans[rem](remote_call(rem,flatten_par[rem](par)))
```

where the answer to the call remote_call is evaluated in B as:

```
   flatten_ans[rem](associated_proc[rem](par))
```

The remote capability rem must have been invented in B by calling a procedure called new_remote. A call of new_remote with parameters consisting of a procedure capability and its type will create a new remote capability, different from al' others; this procedure will the one given by associated_proc above. The type will be used to define the various flattens and unflattens like flatten_ans and unflatten_ans used above. This is a simplification of what actually happens since Flex also has a system of trapping and analysing exceptions in local programming which is extended over the network to allow remote diagnosis of errors.

It is clear from the above that types must be treated as data to determine how one does the flattening and unflattening operations. This is provided for in Flex types by a new kind of value of type Moded; one can construct a value of type Moded from a combination of any other value and its type. This notion is used all over the Flex system; for example the type of the procedure which finds the meaning of a name in a user dictionary is:

```
          Vec Char → Moded
```

where the result includes both the value and type corresponding to the name given by the parameter. The type of the procedure new_remote given above is:

```
          Moded → Moded
```

where the type given in the answer Moded will always be that in the parameter Moded prefixed by Remote. This is similar to expressing the type of new_remote polymorphically as:

```
        ANYTYPE → Remote ANYTYPE
```

although the usual interpretation of polymorphism (see [8]) is that the procedure is independent of the type of its parameter rather than that it uses the type as data.

The representation of types in mainstore has gone though many metamorphoses; originally they were represented by a simple vector of integers. Now, they are represented by a natural graph structure of capabilities, each different type being represented by a unique keyed capability which gives the constructors and constituent types. This uniqueness is maintained in much the same way as for filestore capabilities and their representation allows efficient means of short-circuiting their translation to and from a filestore representation. The filestore representation of a type is just a pair consisting of a filestore capability and an integer; the integer just indexes one of the types coded in the block corresponding to the capability. This representation is not unique; different filestore representations can give the same type.

The flattened representation of a remote capability in the network must perforce be as some sequence of bytes; it is distinguishable from scalar data in the unflattening operation using the usual bit or byte tags. This sequence of bytes must identify the processor which created the capability in the first place and give a unique identification within that processor. The byte sequence will be used on input into a processor from the network to find (or create if it is not already there) the mainstore representation of the capability. In mainstore, a remote capability is represented as a capability to a keyed block containing, among other things, the identification information. As mentioned above, the inputting mechanism will ensure that there will only be at most one such block for each remote capability in each mainstore. Besides the identification information, the mainstore representation also contains an associated value defining the meaning of the capability. If the processor is the one which created the value in the first place by applying new_remote to some procedure, then the associated value will be this procedure. If the processor is some other one, then the associated value will be the type of the capability (which formed part of the flattened value sent to the processor). Thus, finding the correspondence between the capability and its meaning is a fast and economic process.

It is clear that it is fairly easy to find suitable flattened representations for values whose types are primitive or constructed from arrays or structures of other flattenable values. What might be less clear is how one can flatten any procedure values involved in the parameters or result of a remote call. To flatten a procedure value, one constructs a new remote capability by applying new_remote to the procedure being flattened. The flattened representation of this new capability in now the representation of the procedure. To unflatten this procedure representation one merely constructs a new procedure of the same type as the original which does a remote call on the capability as above. This is the way that the push and pop procedures in the remote call of make_stack above are sent from B to A. Sending procedures to and fro like this completely hides the remote capabilities and remote calls involved so that the network is quite transparent.

Such transparency is not always desirable; often one wishes to deal with the remote capabilities directly. For example, modifying the example above slightly, one could write a procedure in B which calls make_stack and applies new_remote to its resulting push and pop to give the result of the procedure; this procedure would have type:

```
        Void → (Remote(Int → Void), Remote(Void → Int))
```

A remote capability to this procedure in A would have type:

```
        Remote(Void → (Remote(Int → Void), Remote(Void → Int)))
```

A remote call of this would result in a pair of capabilities of type:

```
        (Remote(Int → Void), Remote(Void → Int))
```

which would have to be explicitly called remotely to give the stack operations. However, processor A could send one of them to processor C and the other to processor D, thus creating a channel of information from C to D via B which is totally independent of A.

Given that one possesses a remote procedure capability it is easy to see how others can be generated from its parameters or results. One way that has been chosen in Flex to start off the process is by means of the procedure first_function of type:

```
        ComputerId →(Vec Char → Moded)
```

which allows one to ask a remote node, identified by ComputerId, for a value associated with a name given by the Vec Char. Provided that the remote node allows it, this could give access to any of the facilities available in the remote node. For example, a possible example of the Moded value delivered might be a command line interpreter for the remote machine of type:

```
        ((Void → Vec Char), (Vec Char → Void)) → Void
```

whose first parameter is a procedure to give the command interpreter a line to interpret and the second is one to deal with the response to that command. Similarly, a serial file transfer might be a procedure of type:

```
        Vec Char →  (Void → Union(Vec Char,EndOfFile))
```

where the Vec Char parameter is some name to identify the file and the resulting procedure will give successive lines of the file on successive calls.

The lifetime of some remote capabilities can sometimes be deduced by their own actions; for example the capability involved in the serial transfer of a file become meaningless when the end of the file is reached. In these cases the originating processor can discard the capability, safe in the knowledge that any further calls on the capability are mistakes on the part of the remote processor. However, this is not sufficient to provide for the freeing of the resources associated with a remote capability; even in the case of serial transfer a failure in the remote processor might mean that the end of the file is never reached. For this reason, the principal method of freeing these resources once again depends on another kind of garbage collection. In general, a processor which creates a remote capability remembers it so long as another processor possesses it. Every time a remote capability is sent to another node in the network, then this fact is noted by the originating node, either because it sent it itself or else the sending node informed the originator. The originating node can then periodically enquire of these processors whether they still have it; if they do not, then the original processor can forget about it. The method of making this enquiry depends both on the uniqueness property of the mainstore representation of remote capabilities and the storage allocation for mainstores. The remote capability is sent to each remote processor. If there is no longer a copy of it there (ie it has been freed by a mainstore garbage collection), the inputting process will have to re-create the capability ; this fact can be recognised and sent to the enquirer. Both these enquiries and the primitive remote calls depend on some lower level of protocol to determine whether the remote processor is still active so that the communication can degrade gracefully when it is not.

5. Universal capabilities

A universal capability is one which can exist anywhere in the Flex world in mainstores, filestores or on networks. It will be used to represent a commonly used object like a compiler, editor, or common module or even a type. The data or program corresponding to a universal capability can exist in many copies distributed throughout the Flex world, usually in local filestores. This description would seem to imply that such an object would have to be constant through time, though we know that compilers, editors and such values are amended from time to time as errors are removed and improvements are made. We therefore choose to think of the value corresponding to a universal capability as an approximation to a Platonic ideal, and to say that these approximations are ordered in the sense that later ones are better than earlier ones. So every operation that can be done with an earlier value must be able to be done with a later one, with a result which is a better approximation than the result of the earlier operation. It is unfortunately impossible to check this property, we merely state that the mechanisms will work if it obtains. This same idea of approximation is applicable not only to objects like programs, where the idea is of a more accurate or less erroneous program or one which applies to more arguments, but also to such values as bank accounts, where tomorrow's statement contains the same information as today's, together with extra information and the only operations allowed are those which specify a date, rather than work in terms of "now".

One aim with universal capabilities is to provide a mechanism to allow better versions of program or data to be transmitted in a fairly passive way. By this is meant there is no need for the originator of the change to tell everybody about some change all at once; the change can be passed from processor to processor independently of the originator. All that is required is that a processor hears about a change from some other processor and it sets in train the actions to ask the other processor exactly what the change was.

The representation of a universal capability consists of an identification which is unique over the Flex world and a version number defining the approximation. Both of these parts are transmitted as the network or filestore representation of the capability. In addition, a processor which knows a meaning for the capability will associate this meaning (usually as some other kind of capability or capabilities) with its mainstore representation in much the same way as the meanings of remote capabilities are held. Clearly it must be possible to change this association if a later version of the capability is encountered, for example, as part of a remote transmission. In general, if a later version is found as part of a transmission, then the receiver will ask the sender to give the more up-to-date version. Thus, later versions of a capability will diffuse through a network so long as the various nodes of the network hear about them from any source.

The possession of a universal capability implies that one has the right to demand that another processor gives one its current meaning for the capability. Conversely, one has the obligation to provide a meaning that one possesses to anyone who demands it. Leaving aside the problem of how one introduces a universal capability in the first place, the updating from one version to a later one is achieved using remote capabilities. If a remote processor sends a universal capability with a later version than the local one, the local processor makes a remote call to get its version updated. The parameters of this remote call are simply the universal capability and a procedure to update its meaning in the local processor. For example, if the universal capability was a simple serial text file then the updating procedure could have type:

```
    Remote((Universal, Union(Vec Char,EndofFile) → Void) → Void)
```

where the `Union(Vec Char,EndofFile) → Void` parameter will be called by the remote processor with the successive lines of the new text file as parameter to recreate it in the local processor and update the universal.

This is an unrealistically simple example, since most important objects in Flex are much more highly structured than simple serial text files. For example, the documentation file shown in figure 5 is an important universal capability which is an `Edfile`. Its updater is much more complicated; it admits of the possibility of transferring other values besides lines of characters. Since the documentation file is structured so that each of its constituent `Edfiles` is small enough to be displayed in roughly a screenful, the protocol for its transfer can be expressed so that each `Edfile` is transferred in a single transaction in one block rather than serialising the file as in the previous example. The type of its updatercan be expressed as:

```
     Remote( (Universal, Block → Void) → Void)
 where
     Block = Union(PlainText,
                   Line, Page
                   InnerEdfile,
                   ..... {with about 15 other possibilities}
                   ....)
 and
     PlainText = Vec Char;
     Line = Page = Vec Block;
     InnerEdfile = (Vec Char, Date, Void → Block)
```

Each `Edfile` encountered as part of the file is transferred an an `InnerEdfile`. For example, $\boxed{\texttt{Mathematical routines}}$ is transferred as a triple consisting of the string `"Mathematical routines"`, the date and time at which it was created and a procedure which, if called, will deliver the `Block` corresponding to the contents of the inner `Edfile`.

If the local processor already has this Edfile created at this date in its copy of the documentation file, the procedure need never be called. Once again, only that part of the tree which has actually changed need be transferred. The other possibilities in the union given by Block include structures defining how to transfer Modules; this might involve transferring program text for compilation to amend the Module.

Just as the universal capability itself is an approximation to some constant, its updating procedure must also be a constant since it must be capable of being called remotely by some independent processor and hence the type and actions of its parameters must be known to both processors. There clearly has to be no possibility of disagreement in the updating process - once a universal capability has been created its updater must have the same properties of approximation to some ideal as the capability itself. In particular, the type of updater will never change.

In principle, one could introduce an arbitrary updater with each new universal capability. However, in order to transfer it to a new processor, the updater would have to be expressed in terms of existing universal capabilities, otherwise it could not be generally transferred. In practice, there are relatively few different kinds of values represented by universal capabilities, procedure modules, types and and various kinds of files like the documentation file above being the most important. Clearly all capabilities whose values are constructed in the same way can have the same updating procedure. Thus, one can start with one universal capability in every processor which allows one to introduce a new universal capability of one of these common kinds to a remote processor. In this way, most of the problem is solved provided that this initiating capability allows one to update its action by introducing new kinds of updaters.

Universal capabilities do not solve the problems involved with preventing simultaneous changes to the same object; somehow or other there has to be a controller for each capability to ensure that the versions are strictly ordered. Usually this controller is a human one, namely the originator (or inheritor) of the capability. However, on a Flex network, the controller can produce a new version from any of the most recent copies regardless of its physical location. Thus there need be no dependence on a single filestore, for example, to be the "master" copy, with all its attendant dangers of data loss.

6. Conclusion

The varieties of capabilities described here have been arrived at by the experience of implementing and using various Flex configurations using networked machines with both local and shared filestores. Once one accepts the notion of capabilities as first class data objects in the mainstore of a computer, then the extension to allow similar objects to exist in filestores and networks is inevitable. The particular classification given here is a consequence of the often conflicting aims of keeping capabilities as long as they are required while trying to preserve consistency in the sense that if one possesses a capability it should be meaningful. Storage limitations will dictate that, in general, this requires some kind of a trace of the capabilities being used with subsequent garbage collection. The classification given here of mainstore, filestore, remote and universal capabilities defines properties of their use and lifetimes so that this task remains manageable.

References

1. R.S.Fabry "Capability based addressing" CACM Vol. 19 pp403-412 (July 1974)

2. M.V.Wilkes "The Cambridge CAP computer and its operating system", North Holland, 1979

3. M.V.Wilkes "Hardware support for memory protection - capability implementations" Computer Architecture News Vol 10 No 2 pp107 - 116 , (March 1982)

4. J.M.Foster, I.F.Currie and P.W.Edwards "Flex: a working computer with an architecture based on procedure values" Proc. of international workshop on high-level architecture, pp 181 185, Fort Lauderdale, Florida (Dec 1982)

5. I.F.Currie, P.W.Edwards and J.M.Foster "Flex firmware" RSRE Report No. 81009, 1981.

6. P.J.Landin, "The mechanical evaluation of expressions" Computer Journal Vol. 6 No. 4 pp 308 - 320 (Jan 1964)

7. I.F.Currie and J.M.Foster "Curt: the command interpreter language for Flex" RSRE memorandum No. 3522, 1982

8. M.J.Gordon, A.J.Milner, C.P.Wadsworth "Edinburgh LCF" Springer-Verlag (Berlin 1979)

9. B.Liskov, R.Scheifler "Guardians and actions: linguistic support for robust, distributed programs" ACM TOPLAS Vol 5, No 3, pp 381 - 404 (July 1983)

10. S.K.Shrivastava and F. Panziari "The design of a reliable remote procedure call mechanism" IEEE Trans. Computing (July 1982)

11. Xerox Corporation "Courier: the remote procedure call protocol" Xerox Report XSIS 038112, (December 1981)

12. I.F.Currie, P.W.Edwards and J.M.Foster "PerqFlex firmware" RSRE Report No. 85015, 1985.

LMED

-8