

CHARGING IN A SECURE ENVIRONMENT

C.S. Wallace and R.D. Pose
Department of Computer Science
Monash University

ABSTRACT

The Monash Multiprocessor Architecture incorporates a monetary system at the lowest kernel level, integrated with a password capability scheme. Although the capability scheme is quite flexible, providing support for non-hierarchic security and access policies, abstract type management and information confinement, we show that it is possible for service providers to command use-based fees for service. The fee charging protocols must be designed with some care to avoid breaching required information confinement constraints when user and provider are mutually suspicious, but need not be very complicated.

1. INTRODUCTION

Most current multi-user operating systems do not support a satisfactory means for the providers of software and other services to charge for their services. The provider of, for example, a compiler may reasonably wish to be paid for use of this compiler. In most current systems, the only options for the provider are outright sale or some form of licence arrangement. In either case, the price charged must be calculated on the expected level of use and availability of the software rather than actual use, which is not normally readily discovered. Thus, a multi-user site hosting one or two users with an occasional need for the compiler may have to pay as much as a site having many frequent users. The problem is even more marked in large networked systems where cost factors force the network managers to purchase a licence for only one or two computers in the network, so that users elsewhere have to send jobs to a remote site for execution rather than having the necessary software available locally.

A sufficiently secure system can at least allow the provided software to make a tamper-proof record of its use for later billing. However, a suspicious client, perhaps wishing to use the compiler to compile a program of high commercial value, has then no guarantee that the billing records accumulated by the compiler do not capture sensitive information about the client's program. Further, it is not unreasonable for a client to object to the loss of anonymity consequent on the creation of a billing record which must at least serve to identify the client to the provider, and to indicate the scale and frequency of the client's activity. While it is no doubt possible to invent protocols for use-based charging without risk of unauthorised disclosure of clients' information and activity, and without risk of unauthorised copying or alteration of provided software, the protocols will almost certainly be rather elaborate. In this paper, we show how relatively simple but effective charging mechanisms can be built into a secure operating system. The system described is capability-based [1, 2]. Only the aspects of relevance to charging will be described here.

2. PASSWORD CAPABILITIES

2.1 Objects

Data, programs, processes and I/O resources are all treated as "objects" in this system. Broadly, different kinds of object are distinguished only on the basis of the different kinds of access permitted to them. Thus, the access right "send message to" can exist only for an object which is a process, and the access right "execute" would exist only for objects comprising executable code. The distinction is not absolute: in some circumstances read and write accesses may be permitted on an object normally treated as a process.

The distinction between processes and other objects is the one of most significance, since the system expects and enforces a certain format on process objects, whereas other objects are, at the lowest level, treated simply as a consecutively numbered set of words.

All objects can also act as stores for money.

2.2 Capabilities

A capability is a datum, knowledge of which permits a process or other agent to exercise one or more defined rights of access to a specific object.

There may be several capabilities for the same object. One of these, the *master*, is created when the object is created, and confers whatever rights the creator specifies. The others are known as derivatives. The set of capabilities for the object forms a rooted tree, the derivative tree, with the master at the root. Every capability save the master has a parent, namely the immediately adjacent capability in the direction of the root. An agent knowing a capability may call the system to create a derivative, which becomes a child of the original, and which may have any specified subset of the rights of the original.

The format of a capability is a 64-bit unique *object name* and a 64-bit randomly-chosen *password*, sometimes considered as comprising two 32-bit password fields P1 and P2. Such a capability is depicted in figure 1. The various capabilities for an object share the same name, but have different passwords. Note that a capability is not tagged or physically segregated from other data. It is simply a 128-bit datum which may be represented in any data structure, copied, printed, written on a piece of paper etc. The security of the capability system relies on the infeasibility of guessing a valid password.

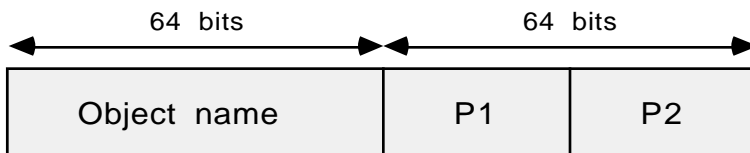


Figure 1: A Password Capability

The capabilities do not explicitly encode the size or exact location of the objects they name, nor the rights they confer. This and other information relating to a capability is held in a *catalogue*, which is a system data structure outside the world of objects, and for which there is no capability. The catalogue entry for a capability records its rights and its parent (if a derivative) or size, status, and location information (if a master).

When a new capability is created, its password is generated by a special hardware unit [3] guaranteed to produce an unpredictable sequence of words, and the password checked to ensure it is unique among all the capabilities for the same object.

2.3 Rights

Broadly, rights can be grouped into three classes: those applicable to processes, those applicable to passive data, and those applicable to objects of any kind.

Process rights: These include rights to send a message, to suspend or resume execution, and to *lock* (a security-oriented operation described later.)

Data object rights: These include read, write and execute rights, and the right to extend or contract the object size. Read, write and execute rights can be limited in extent, allowing access only to a consecutive subset of the words in the object, called a "window".

General rights: General rights for the most part relate to the capability conferring these rights rather than to the object itself. They include rights to derive another capability for the object of equal or lesser rights, the derivative becoming a child of the original; the right to destroy the capability, and implicitly all its derivatives; the right to rename the capability, in effect destroying it and all other capabilities for the object but replacing it with one of equal rights, which becomes the new master; the right to deposit money and the right to withdraw money. It is these last rights which will be of most concern in this paper.

Money rights: As has been mentioned, every object, besides its role as a process or body of data, is a store of money. The record of the amount held by the object resides in the catalogue entry of its master capability, in a *moneyword*. Derived capabilities also have money words, but these do not represent funds additional to those in the master money word. Rather, the money words of a derivative bounds the amount of the object's money which is accessible via the derivative. When a sum S is withdrawn from an object using a derived capability C , the money words of C , the master capability, and every capability in the derivation tree on the path from C to the master are decreased by S , and must remain non-negative. Each of these capabilities must, of course, have the "withdraw" right.

Conversely, when a sum S is deposited using a derived capability C , the money words of all capabilities from C to the master are increased by S , and each capability must have the "deposit" right. There is no requirement that the money word of a capability be less than that of its parent, nor that it have any relation to the money words of its derivatives, or their sum.

3. OBJECT RENTAL

Periodically, the system deducts rent from the money of each object in proportion to its size. If its money is thereby exhausted, the object is archived and eventually destroyed. It was for this purpose, of removing "garbage" from the system, that the money mechanism was originally introduced. The system has no concept of object ownership or user account numbers, so no user can be identified as responsible for the continued presence of an object. Further, the fact that valid capabilities may be recorded outside the computer system, and later legitimately presented for use, makes it impossible for the system to detect when an object has become inaccessible by any agent. The rent scheme ensures that objects will survive only so long as some agent is willing to pay its rent, and that any system memory occupied by objects is paid for.

4. SYSTEM CHARGES AND PROCESS CASH

A process object essentially comprises a processor state image, a mailbox for the receipt of messages, a set of capabilities defining the current context of the process (i.e. the code, stack, data and other objects visible in the process logical address space), a *lockword*, and a *cashword*. None of this information is normally directly readable or alterable by any process. The cashword represents a sum of money additional to the sum stored in the process object itself, and is immediately accessible to the process during execution, without requiring reference to the catalogue.

The process is charged for processor time, and for services invoked by system calls, with payment being deducted from the cashword. Money deposited by the process into objects (including itself) is taken from its cash, and withdrawals add to its cash. One may consider object moneywords as bank accounts, and process cash as spending money. A process which exhausts its cash is suspended, even if the process object contains money.

The inter-process message system, which basically allows a process to send a small message to another process and wake it up, also allows money to be sent. Money in a message is added to the cash of the receiving process immediately, and the size of the sum may be discovered by the process by a "receive message" call. Messages may be sent anonymously, or may include the name of the sending process. The receiving process may also discover the capability for itself used to send the message. Thus, if a process distributes various derived capabilities for itself to other agents, it may discover to which "hat" an incoming message (and perhaps payment) has been directed.

5. CREATION OF MONEY

Money is consumed by the system in two ways. Rent is collected from objects hence depleting their funds and processes are charged for their processor time and other system services. Unless more money is injected into the system the total money supply could eventually become exhausted.

There is a trusted system administrator who is able to create objects containing money. He would normally only do this in return for some form of legal currency. When a user first wants to use the system he pays the system administrator an amount of real money. The system administrator creates an object containing the equivalent amount of system funds at the current exchange rate and gives the user a capability with withdrawal rights for the object. The user can immediately rename the object in order to prevent anyone else having a capability through which these funds may be withdrawn.

The system administrator will also convert system funds into real money. Given a capability with withdrawal rights he will destroy the specified system funds and give the person an equivalent amount of real money. In this way surplus funds may be withdrawn from the system.

One of the aims of the system administrator is to have the computer system run at a profit or at the very least break even. The exchange rate between real money and system funds is thus set so that the rent charged for objects reflects the actual costs incurred in operating the hardware on which the objects are located and allowing for eventual replacement at the end of its service life. Similarly the charging for processor time and system services should reflect the actual costs of providing the services and may also include a profit component. It is expected that market forces will prevent the system administrator setting the exchange rate at unrealistic levels. If he charges too much people will not use his system and if he charges too little he will not recover his costs.

6. SIMPLE CHARGING

Many useful services may take the form of simply providing some code which the client may execute. The provider of such software may extract a fee for service very easily. The provider (or a process acting as her agent) installs the software as an object of executable code, and makes publicly available a derived capability for the object having only the execute right. A client process wishing to use the service may call the code by

using this capability, and execute the code in-process. The code extracts payment from the client by depositing in itself a sum which it calculates on the basis of the scale of service. The money of course is taken from the client user's process cash. This scheme is depicted in Figure 2.

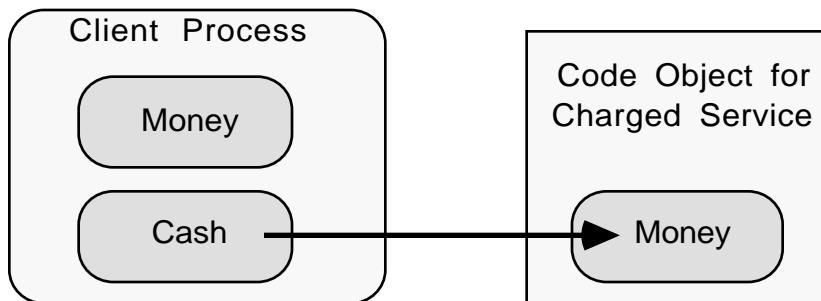


Figure 2: Simple Charging.

Funds accumulating in the code object may be withdrawn from time to time by the provider's agent, and by no-one else, since only the provider has a capability for the code giving withdrawal right.

This simple scheme places all the direct costs for processor time, temporary data storage etc. on the client, since it is the client's process cash which must be used to pay for these system services. The money deposited by the provided software in itself is wholly income for the provider save for rental cost incurred by the software object.

The above scheme is not limited to services provided by in-process execution. The provided code may create new objects, including new processes, and may itself call other service providers in the same way. Ultimately, all funds passed to other providers, or used in paying the system costs for new objects and processes, is drawn from the original client process cash. While simple and efficient, the scheme should be an adequate charging mechanism whenever the provider is not suspected of misusing client information, and where the entire fee for the service can be computed and captured while the client process is under the control of provided code.

7. TYPE-MANAGED SERVICES

Some services cannot be provided by a single call on server code. An example is a service for creating, updating and querying a database. In such case, the client will expect to make many calls on the service interspersed with execution of his own code. Further, one or more data objects may have to be made by the server, which must survive between calls on the server, whose rental is the client's responsibility, but whose internal structure should not be visible to the client. Such services are often called *abstract type managers*, and the data objects they create on behalf of a client *instances* of the type.

Our system provides a mechanism useful for such servers, allowing them to be provided as pure code objects just as in the simple case of section 5, without requiring the provider to assume responsibility for the rental or identification of instances.

When a type manager routine creates an object on behalf of a client process which has called it, the manager may *seal* the capability for the object by XOR-ing the P2 field of the capability with a key pattern K embedded in the manager code object. The sealed capability is returned to the client, who must pass it back to the manager whenever requesting a further action on the instance. The manager can of course recover the original instance capability by XOR-ing the P2 field of the sealed capability with K. The client, however, cannot access the instance. Figure 3 shows a process calling a type manager to create an instance of a type.

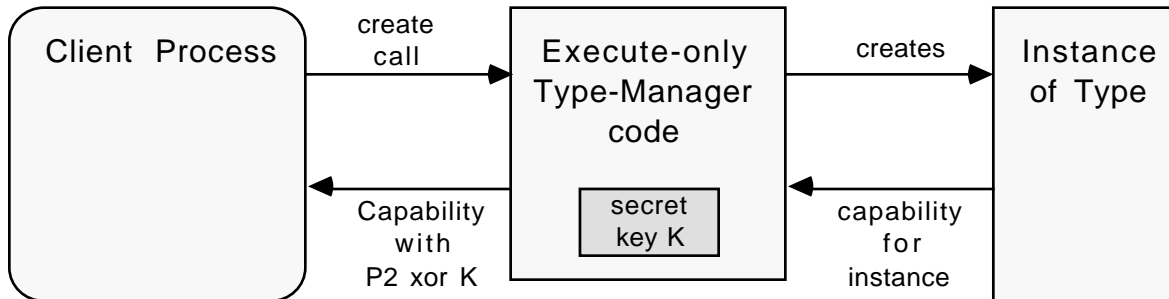


Figure 3: Creating a Typed Object.

Many systems can support type managers of this general sort. However, it is usually necessary for the client to call on the type manager for even the most routine management of the instance. In our case, it would seem that the client would have to call the type manager whenever the client wished to deposit funds in the instance, or to derive a child capability for the instance, or to copy the instance. In fact, our system allows the client to use the sealed capability directly for these purposes.

If a client presents a sealed capability to the system, its original unsealed version can be found in the catalogue by matching the object name and P1 fields alone. The system can then discover the seal key K as the XOR of the true and sealed P2 fields. The system will accept the sealed capability as sufficient authority for any action within the rights of the true capability, except actions which could reveal or alter the data in the instance. If the requested action is *derive*, *copy* or *rename*, involving the creation of a new capability, the system will seal the new capability with K before returning it to the client. Thus, a client may pay rent for, share use of or copy an instance without reference to the type manager, but any new capability created can be fully exercised only by the manager.

A type-managed service routine can charge for its use whenever called, and has available all instance information relevant to the fee calculation. However, it need carry no cost for the maintenance of the instances between calls, and need not even maintain itself any list or identification of extant instances.

8. VULNERABILITY OF PASSWORDS

The 64-bit password field of a capability provides quite good security against malicious forgery. It might be considered that, given that some objects may have many valid capabilities, even this password size would be vulnerable to a systematic search. However, the charging mechanism is used to make such an attack infeasible. Whenever an invalid password is presented to the system, a small fine is levied on the offending process. The fine, of order a cent, is small enough to be tolerable for occasional innocent mistakes by user processes, e.g. an attempt to access an object previously destroyed, but large enough to make the cost of a serious attack prohibitive.

It may seem that the willingness of the system to accept for some purposes capabilities with invalid P2 fields opens a much cheaper line of attack. It seems that an attacker could exhaustively try different P1 values, say for a *derive* operation, until one succeeds, then try all P2 values together with the successful P1 value, say for a *read* operation, until a fully valid capability is discovered. The number of fines to be paid would then be of order 2^{32} rather than 2^{64} , and perhaps tolerable if the stakes are high enough. However, the system defeats this line of attack by returning an apparently successful result on any operation where only a valid P1 is required, even if the presented P1 is invalid. For instance, a *derive* operation, when given an invalid P1, will happily return a derivative. The nature of the operations permitted using sealed capabilities is such that their success can be determined only by the later success or failure of an operation requiring a fully-valid capability. If this fails, the attacker cannot know whether the failure was due to an invalid P1, an invalid P2 or both, and is so left with a search space of 2^{64} .

Note that sealing is transitive. If client U uses manager M1 (key K1) which in turn uses manager M2 (Key K2), the instance capabilities returned to U for objects created by M2 will be sealed by both K1 and K2, and can be unsealed and used by M2 only if passed back through M1.

9. LOCKING

The above simple charging schemes are inadequate if the client suspects that the server may misuse the client's data. The capability system has a mechanism called *locking* which can prevent such disclosure. It is available both in-process (i.e. when the server is called as a procedure) and out-of-process (when the server is a separate process.) The former will be assumed here: there is little difference in the two cases.

A process may make a *protected procedure call* which differs from a normal procedure call in that

- (a) capabilities immediately available to the caller (i.e. objects which are visible to the caller because it has previously presented capabilities for them) may be hidden. The capabilities are placed in a stack in the process object which is not directly accessible to the process, and the objects will reappear only on return from the protected procedure (the server).
- (b) the caller specifies an arbitrary 64-bit *lock* L.
- (c) a set of capabilities may be passed to the procedure by the caller. However, the password (both P1 and P2 fields) of each is XOR-ed with L and only the modified form is visible to the called routine.
- (d) 64-bit *lockword* W in the process object is replaced by W XOR L.

During execution of the server procedure, all capabilities presented by the procedure to obtain access to an object are automatically XOR-ed with the new W before the catalogue is consulted. If W was originally zero, and the original capabilities passed in to the procedure were valid, the catalogue search will succeed. Further, if the server procedure creates any new capabilities, the form returned to the server by the system will have its password XOR-ed with W. The effect is that the locked procedure can freely access those objects passed to it by the caller, can freely create and use new objects, but cannot access any other preexisting object. Any object which the procedure creates may survive return from the procedure, and the procedure may return to the client (as an ordinary datum value) a capability for the object which the caller, knowing L, can unlock. However, even if the procedure, through some covert channel, can succeed in conveying to a third party a capability for some client or new object, the capability will not be valid, and cannot be unlocked without knowledge of L. Thus, disclosure by the server of any client data is infeasible. Figure 4 shows a process locking itself.

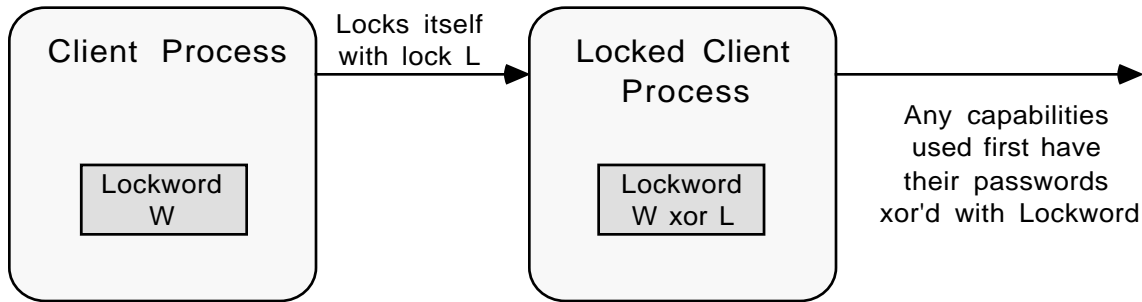


Figure 4: Locking a Process.

Note that locking is transitive. If a process with $W=0$ (i.e. unlocked) calls server $S1$ with lock $L1$, $S1$ will run under lockword $W1=L1$. If $S1$ calls server $S2$ with lock $L2$, $S2$ will run under lockword $W2=L1 \text{ XOR } L2$. If $S2$ creates a result object and returns to $S1$ its capability, the capability will be locked by $W2$. Knowing $L2$, $S1$ can remove the lock $L2$, giving a form locked by $L1$, and hence usable by $S1$. In fact, there is no real distinction between locked and unlocked states; a process cannot discover whether or not it is locked, let alone what its lockword might be.

When a locked process creates a new process, the new process normally inherits the lockword of its creator, and so has no more or less freedom than its creator. However, the creating process may if it wishes apply a further lock to the new process, in effect achieving an out-of-process form of the protected procedure call.

There is one exception to the automatic application of lockwords. A capability whose rights do not allow any change of state of the object or any capability for it is called a *non-alter* capability, and a reserved digit of the capability format is used to mark it as such. Non-alter capabilities are not locked on being passed in a protected procedure call, nor XOR-ed with the lockword on presentation, since they cannot be used to convey information to third parties. This exception allows a server procedure to exercise read-only or execute-only capabilities embedded in its code, for instance to call other procedures or to read tables of constants.

A cautionary note: No user of a locked service should pass to the locked server routine or process any capability which the user has received from a third party. If the user does so, it is possible that the third party has conspired with the service provider to place a copy of the capability somewhere where the locked server can read it using a non-alter capability. The server, by comparing the true copy with the locked version passed to it, could then discover the lock under which it is running, and hence disclose any information it liked. To prevent this possibility, a user should always rename or derive an equivalent of any capability received from third parties before passing it to a locked server.

10. CHARGING FOR LOCKED SERVICES

Charging for a locked service is rather more complex than for a trusted service. First, a locked routine cannot deposit fees in itself. To permit it to do so, while allowing the provider or her agent to withdraw the deposited funds, would open a covert channel through which the locked routine could, by the timing and/or value of its deposits, convey client information to the provider. Indeed, if the covert transmission is to be entirely blocked, the total fee recovered by the provider cannot be allowed to be a function of client information, and so must be fixed before the provider gains access to the client's data. Thus, the client must inform the provider's agent about the scale of service requested, and the agent must fix a price for the service, before the service commences.

Second, while it is necessary to lock the server procedure to prevent it from disclosing client data, it must still be possible for the procedure to alter the state of some object external to the procedure, but inaccessible to the client, so as to record the fact that service has been provided. Otherwise, once the client had paid for a service, there would be no memory outside the client's reach as to whether the service had been performed, and the client could re-use the service without further payment. We require that provision of the service alter some state information which is:

- (a) presettable by the provider, so as to fix the amount of service to be performed;
- (b) invisible to the provider, lest the alterations caused by the service procedure disclose information;
- (c) inaccessible to the client, lest the client reset the state to erase memory of service performed and hence steal additional service.

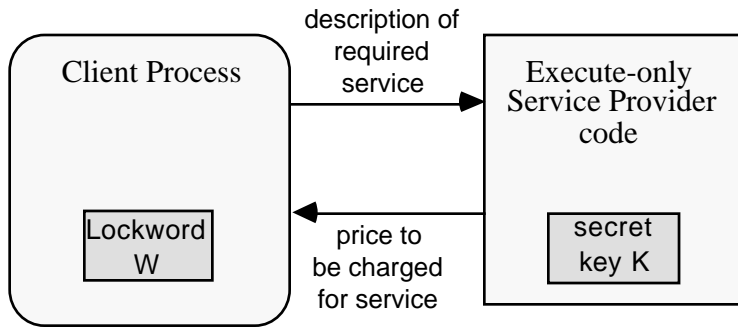
Our system allows these requirements to be met. Assume the service will be provided in-process by a locked procedure. The client describes the task to the provider. The provider quotes a price to the client. The client sends the provider the stated fee. Now the provider creates a data object containing a description of the amount (and perhaps nature) of the service to be done. The provider derives a capability C for this object, having only read, write, deposit and rename rights, then seals C with key K, and sends the sealed form to the client.

The client cannot read or alter the data object. However, he can deposit in it enough money to ensure its survival for the duration of the service (any excess will be lost). He then *renames* the capability, in effect destroying it and receiving a replacement with the same rights, a different password, and still sealed with K. The client cannot, of course, read or write using the new capability, but now the provider can no longer reach the object at all, since any capability for the object she may have retained is no longer valid. The client now calls the service procedure (the capability for which can be publicly known) via a protected procedure call with some lock L, and passing in a capability for his data, and the sealed, renamed capability C.

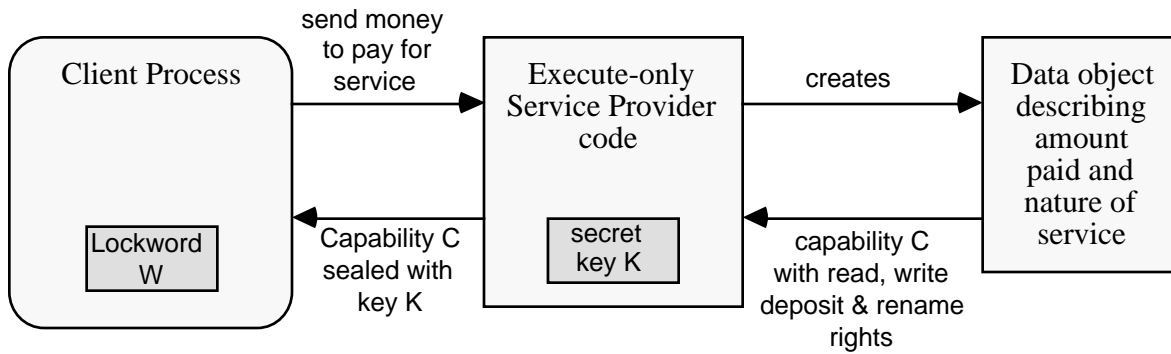
The service procedure has constant K in its code body, and so can unseal C. Note that the unsealed version will still be locked, but sealing and locking commute. The service procedure can now read and write the "amount of service" object, and decrement the amount as it proceeds. It will, of course, cease work if the amount decreases to zero.

This change of state in the "amount of service" record is invisible to the provider because of renaming, and inaccessible to the client because of sealing.

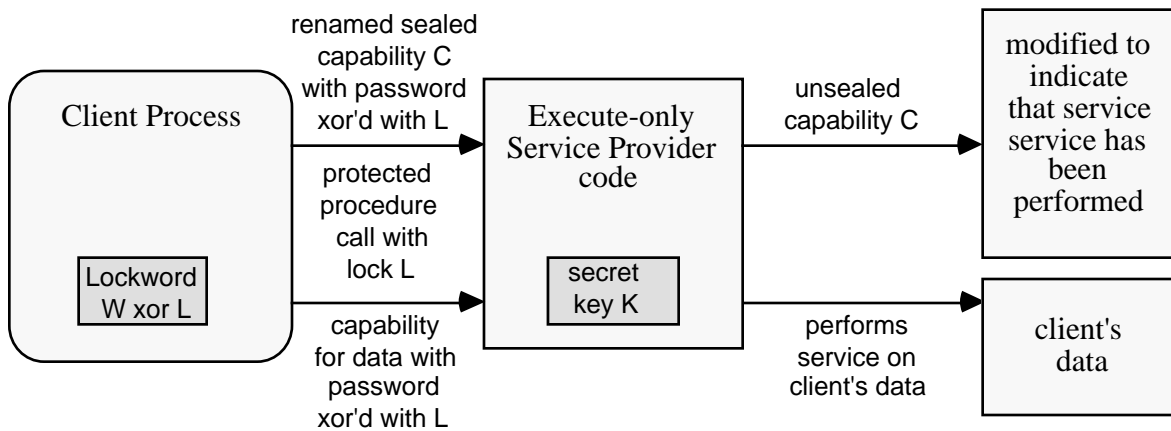
Note that if the service is a type manager, it can when first called copy the "amount of service" record into a type instance object, so it can accumulate the total amount of service provided over a succession of calls. The type instance is also inaccessible to the client (because sealed) and to the provider (through ignorance of its name). Such charging is depicted in Figure 5.



Step 1. Getting a quotation for the service.



Step 2. Paying for the service.



Step 3. Performing the service.

Figure 5: Charging for Locked Services.

The above scheme for charging for locked services is easily modified for out-of-process services. In this case, the provider when paid may construct a process to perform the service and give the client a capability for the process which does not allow the deposit or the sending of funds to the process. The initial cash placed in the process by the provider can be calculated to limit the amount of service provided by the process. It is still necessary for the client to rename and lock the server process, to block any information channel between server and provider. Note that, as the mailbox in which a process receives messages is finite, a process receiving messages can convey information to the process sending them by the timing of its reading and clearing the mailbox. By renaming the server process, we prevent the provider from sending it messages.

This is not the only scheme which may be devised for charging for locked services.

The service provider can make available a capability with execute-only rights for an object containing procedures which can provide quotations for service and which can charge for requested services. The charging procedure deposits the funds and records details of the type of service for which payment has been received and a randomly chosen password in an object whose capability is embedded in the execute-only code. A server process is created by the charging procedure. Its program code will perform the required service. A read-only capability for the object containing the record of payment and password and the password itself is placed in the server process. A capability for the server process with rights to deposit and withdraw funds, to lock the process and to send and receive messages but not to alter other aspects of the process is returned to the client. These procedures are run unlocked. The client should immediately rename the process to ensure that no-one else can make use of it as an information channel or just to steal services for which the client has paid. Note that when an existing process is locked, all loaded capabilities are revalidated to ensure that they are allowed to be used in the locked environment. This is required to prevent alter capabilities being used for information channels.

When the client wishes to have the service performed he may lock the server process and send it a message containing capabilities for objects defining the work to be performed. The process can verify the password and the service for which the client has paid by examining the object in which the charging record is stored using the read-only capability. It cannot however alter the charging record to indicate that the service has been performed since its capability only provides read access.

This scheme for charging for locked services uses a different mechanism for ensuring that the client does not use the service more than once without being charged for subsequent use. Instead of having the service adjust the "amount of service" data to reflect the actual use of the service, the use of the service is reflected in the changing of the program counter of the server process from its initial state. Because the client has no capability allowing for the program counter to be reset to the start of the service program, there is no way for the service to be used again without going through the charging procedure. The password allows for many charging records to be stored in the one object. If there is only one charging record in the object the password is not necessary.

11. CONCLUSION

The use of password capabilities makes a money system attractive if only as a means for garbage collection. However, the money system can then be used to allow rational used-based charging for services, even when those services are not trustworthy. The capability sealing and locking mechanisms, although introduced to facilitate type managers and information confinement, prove useful in implementing some versions of charging.

ACKNOWLEDGEMENT

We acknowledge the significant contribution of Mark Anderson in devising and implementing the Password Capability System and data confinement mechanism on which these charging mechanisms are based.

REFERENCES

1. Anderson, M., Pose R.D. and Wallace, C.S., "A Password-Capability System", *The Computer Journal*, Vol. 29, No. 1, 1986, pp. 1-8.
2. Anderson, M. and Wallace, C.S., "Some Comments on the Implementation of Capabilities", *The Australian Computer Journal*, Vol. 20, No. 3, 1988, pp. 122-130.
3. Wallace, C.S., "A Physically Random Generator", *Computer Systems Science and Engineering*, Vol. 5, No. 2, 1990, pp. 82-88.