Technical Report

UCAM-CL-TR-37
ISSN 1476-2986

Number 37

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Representation and authentication on computer networks

Christopher Gray Girling

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted April 1983 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Queens' College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

CONTENTS

SUMMARY

Controlling access to objects in a conventional operating system is a well understood problem for which solutions are currently in existence. Such solutions utilize mechanisms which accurately and trivially provide the identity of an accessing subject. In the context of a collection of computers communicating with each other over a network, provision of this mechanism is more complex. The design of such a mechanism and its implementation on the Cambridge Ring at Cambridge University is described.

The vehicle used to prove the identity of an object irrefutably is called a representation and the deduction of an object's identity is called authentication. Methods of authentication are given which show that the mechanism can cope with identification needs that arise in practice (even in a network where the function assigned to each computer is constantly changing). These generate representations for such important components of a computer network as people, services and addresses. The implementation of a representation system utilizing some of these methods is described, including the incorporation of its use into a real operating system. The place of representations within the communication protocols that must transport them is considered and some enhancements are proposed. In addition, some interesting variations and extensions of the system are explored.

Article 1

INTRODUCTION

This article explains the history of the research described in this thesis and the way in which the various sections can be read.

1.1 Development

In order to understand how the topics that appear in later articles are interrelated, it is useful to discuss the background into which they fit.

Originally, interest was centred on the provision of a Distributed Ring Operating System (DROS) for the Cambridge Ring, a local area network. The idea was to produce an environment in which computers (and other resources) were provided only when needed, in much the same way that a normal operating system controls and allocates the resources at its command.

A system was envisaged in which each user would log in to a small microprocessor on the network giving him access to a simple stub of a command interpreter. This stub would decode user input and initiate the allocation of the appropriate resources for each command decoded.

It soon became apparent that the existing system had insufficient facilities to provide what DROS would need. Apparatus for passing around access to a user's input (in this case provided by a terminal concentrator) was hypothesized and the nature of the small stub was investigated. The implementation of "dynamic services" seemed to pose the major hurdle. Dynamic services are those which are created on demand at some arbitrary computer in the network as distinct from static services which always exist at a particular place.

Since, by policy, all services should be addressed using a textual name, a mandatory feature of dynamic services was that the nameserver used to look their addresses up in should enable names to be inserted and deleted at frequent intervals. Such a nameserver was designed and proposed, but never implemented. It was pointed out that the addition and deletion of names to and from the nameserver was dangerous in that no access controls were

available that would prevent wanted services being deleted or Trojan Horse services from being inserted. Some way of identifying the people and services that use and update servers such as the nameserver seemed to be necessary.

In an environment in which the position of particular services could not be forecast DROS presented several problems. In particular, if any service potentially could be at any given address how could its actual identity be incontrovertibly asserted. From the opposite point of view, having been called into existence by a particular entity, how could a new service safely identify its creator in order to grant it access priority. Other parts of DROS obviously needed similar apparatus. As demonstrated in the nameserver example given above, any service may need to enforce some access control based upon the identity of its clients. When the clients were people, password mechanisms could easily be used for this authorization but how could this generalize to arbitrary clients (such as other services) and what could be done about a user's password management problems and the tedium of having to supply a password at each invocation of a service?

It was these problems that led to work in the area of object identification, representation and authentication. For every type of object the same method of identification and representation was used even though different methods of authentication were necessary. The "representations" that were produced seemed to slot quite neatly into the solutions of some of the problems that were presented by DROS and Methods of adopting them for these, apparently diverse, topics were developed.

Initially a method was devised involving a single authority able to create representations. This was soon found to have shortcomings in a practical system. It limited the use of representations to very trusted objects all of which had to be given the ability to create any representation whatsoever. A better system was found in which each representation was created by something that was made explicit in the representation's name - so that the representation could be evaluated in relation to its creator rather than being necessarily "absolutely correct" or "absolutely incorrect". The consequence of this change was that to a certain extent the topic of "representations" started to encroach into the area of "naming". The environment in which a name had been defined and the name of the authority that created it were both found to be fundamental parts of the specification (name) of an object and relevant to its representation.

This stimulated interest in other forms of names and the way in which they affected the quality and usefulness of representations using them. One conclusion from this, mostly theoretical, work was that the most useful names can have a very general format. In fact, a name can be composed of parts which can include concatenation, alternation and arbitrary repetition of their constituent parts. Such a name, however, presents severe problems in practice.

Existing work on authentication and object representation was found in the area of encryption. In general such work seemed to confuse the areas of communication and object representation; probably because of the ability to use encryption in both fields. A certain amount of effort was devoted to the division of these two areas and to the construction of a protocol which delineates the boundary between them. The view was taken that an object representation system is of a higher level than a private communication mechanism - the former being necessarily implemented on the latter. It was found that faulty communication could result despite absolutely private communication channels. The protocol devised to overcome this problem used object representations.

The mechanisms needed to provide object representations were implemented and the decisions that were taken in their design are documented. These mechanisms can be thought of as primary levels of a distributed operating system "kernel" which would be destined to support DROS. Higher levels which include resource management, the design of a "person server" (the stub described above) and various ancillary services were not implemented but the bases of their designs are described.

1.2 Thesis Structure

Although the original reason for the research was the generation of DROS its central theme is different being that of **object representation.** The thesis covers the way in which an example object representation system was designed and implemented; the way it could be (and is) used; how a distributed system could be based upon it; how it could be extended and improved.

A number of independent topics have been isolated from the overall area each forming a separate "article" (of the type that a newspaper might be composed of, rather than a constitution). It is this introduction alone

that attempts to show that the subjects of these articles are different aspects of a central theme. Each article is intended as far as possible to be free standing, although, sometimes, they may require another to have been read for background information. For this reason they are short and numerous.

Briefly the articles can be summarized as follows:

(1) Introduction

The history, structure and theme of the thesis.

(2) Base Network Structure

This describes the ideas and principal components of one model of a network. It is this model that will be used in later articles and so suitable nomenclature is introduced. The model is sufficiently general to apply to everyday networks and the telephone service is used as an example.

(3) Implementation Environment

This article outlines the Cambridge Ring on which a representation system has been implemented. It describes the actual versions of the components of the model presented in article (2) and the implications they have for a network user.

(4) Design of an Object Representation System

This is the first of two articles dealing with the development of an object representation system. It considers, informally, the meaning of "representation" and proposes three fundamental components: a naming system, a source of unique unguessable tokens, and a relation between names and tokens. It then builds up a list of properties that each of these components must possess. A global naming system is chosen together with a source of "almost" unforgeable tokens. A table is used for the implementation of the relation and its format is elaborated in the light of the control that must be exercised over it. The article develops an ad hoc notation suitable for talking about representations.

(5) Thesis Context

This contains a survey of some of the work done by others in the fields most closely allied to those in this discussion and the position of this thesis in relation to it is defined. The work reviewed encompasses access control mechanisms, models of protection and user authentication.

(6) Implementation of an Object Representation System

The second of the two articles on the development of an object representation system describes the actual implementation of the three main components of the system on the Cambridge Ring. It explains design decisions prompted by the nature of the network given in article (3) and the limitations arising from other decisions made about the implementation of the various components.

(7) The Source Of All Power

After considering the nature and "power" of various orders of Object Representation, as developed in article (4), this article concludes that at least one place from which all authority can be initially delegated is necessary. The most powerful name in a representation system is called the "SOAP name" and the feasibility of simultaneously acknowledging several such names is considered. The article mentions a service with an interface similar to that of the privilege manager, described in article (9), which utilizes the power associated with this position in a well controlled way.

(8) Example Servers

This hypothesizes some services that would make good use of object representations. They demonstrate how various problems that would be encountered in DROS could be overcome using them. In particular, the article discusses accounting, the construction of services (also discussed in article (13)), and the place of a human user in a fully distributed system.

(9) Privilege Management

This article describes a method of protecting entries to services using the representation system to create "privilege" objects. The way in which such objects are obtained and used is described and the design of the interface to the service is discussed in detail.

(10) Variants in AOT Design

This article gives the pros and cons of alternative ways of implementing representation systems. First it discusses the effect of different naming schemes. The systems corresponding to single-part and double-part names are evaluated and a correspondence between systems using double-part names and names with any fixed number of parts is found. Consideration of names with a variable number of name parts leads to an exploration of the most general form that a name could take. This is found to be a graph structure of such generality that one of the most useful properties of the names (that they have a canonical form) appears to be lost. It then considers the effect of restricting the period of validity of a representation to the period of validity of its creator and goes on to investigate other ways in which the use of representations could be restricted. Lastly,

a system which controls the number of simultaneous owners of a representation is described.

(11) Authentication

This article considers the meaning of "authentication" and its relationship to "representation". It develops a model of authentication as a mapping between one representation domain and another and isolates four broad categories of authentication methods. A distinction is drawn between trust and authenticity.

(12) User Authentication

This chooses the authentication method that best applies to people from those proposed in article (11). It then outlines the implementation and design of a service which authenticates users with particular reference to the choice of the functions that form its interface to the network. A similar service that authenticates computer operating systems is mentioned.

(13) Service Representation

Using the guide lines set down in article (11) this article considers the ways in which a service can be authenticated in order to provide it with a representation for itself. It explains the role of a booting service as a service authenticator and the reasons for using "authentication by creation". Some ways in which a service would use its representation are also suggested.

(14) Resource Management

After discussing what is meant by the term "resource" with respect to a service, this article explains the use of object representations for resources. Resource ownership and possession of the representation for a resource are linked so that resource allocation can be implemented by simply distributing representations. The article gives an example in which computers are shared amongst clients.

(15) Authenticated Communication

Even when perfectly private communication paths are provided there is a possibility that data communicated can be misappropriated by being transferred to the wrong party. Two broad ways of combating this problem are proposed both involving the authentication of addresses. The first authenticates just one end of a communication and the second authenticates both ends. Each method uses the identification that a service obtains when it is authenticated in the manner given in article (13). This article gives protocols used to implement these two methods and explains the concept of the "principal" of a communication. It also discusses the relationship between encryption and secure communication drawing a distinction

between methods of secure communication and methods of object representation.

(16) UID Sets on TRIPOS

This article documents the ways in which the TRIPOS operating system makes use of the facilities provided by the services that have been implemented on the Ring. A "fridge" for the maintenance of representations on the user's behalf (mentioned in article (8)) and some other necessary enhancements to TRIPOS are described. The article also mentions some commands for using the user authenticator, the AOT and the privilege manager (outlined in articles (6), (12), and (9) respectively).

(17) Summary and Conclusions

The most important aspects of the foregoing articles are consolidated and summarized in three sections corresponding to the categories listed below.

(G) Glossary

This explains certain key words, that are used in special ways throughout the thesis.

(R) References

A list of references to other work is given.

(A1) Appendix 1

This gives the layout of the parameters in an authenticated BSP open block.

Different sets of articles can be read to reveal various aspects of the work. These aspects are roughly threefold:

- A number of the articles deal with the theoretical and more abstract nature of naming, representations, and authentication. These include articles (2), (4), (5), (7), (10), and (11).
- Another set deals with the actual implementation of some of the theory in the form of servers on the Cambridge Ring. Among these are articles (3), (6), (7), (9), (12) and (16).
- A third group proposes the design of parts a system (DROS) based on the mechanisms implemented and mainly comprises articles (2), (7), (8), (9), (12), (13), (14) and (15).

Article 2

BASE NETWORK STRUCTURE

The articles that follow this one make some assumptions about the nature of the network discussed. This article describes the context that the word "network" implies and develops a certain amount of nomenclature (new terms being introduced in bold type).

2.1 Services

The entities that communicate on a network are called **services.** A service can be located on a network via its **address** which may or may not change with time. Services whose addresses do not change are called **static services** and those whose addresses do change periodically are called **dynamic services.** Services which exist solely to perform some function on behalf of their clients are called **servers.**

Services are **independent** and, with the exception of network communications, they use only information to which they have exclusive access (called local information).

Each service will typically provide several **entries.** Each entry to a service will cause it to perform some particular operation. In general, the local information in a service will be used to represent some abstraction, each operation on that abstraction being carried out by a separate entry.

2.2 Communication

Any service may communicate with any other service whose address is known. The protocols used, the mode of communication and the nature of the communications medium (electrical or otherwise) are irrelevant to this model. It is assumed that some reliable method of passing information from one service to another has already been found.

It is proposed to use this existing method of communication for confidential information. The method of communication least likely to divulge such data will be called the Best Inter Service Communication Method (**BISCM**). No mechanism for improving BISCM will be proposed, it is assumed that the level of privacy provided by BISCM has been accepted as sufficient.

2.3 Objects

The things that are sent across the network are referred to as **objects** here. An object can be passed from one service to another only if both sending and receiving service understand the format in which it is sent. A formal definition of which format to use for which type of object would constitute a parameter transfer protocol (**PTP**).

Services using local information to provide an abstraction may wish to pass representations of individual abstract objects to clients. Such a representation need only be a reference to the object; duplication of the local information considered by the service to represent the object is not necessary.

Passing such a reference from service to service, in effect, passes the abstract object from service to service. Possession of the reference can be regarded as possession of the abstract object to which it refers. To this extent such a reference is similar to a capability in capability operating systems. Such references are discussed in greater detail in succeeding articles.

2.4 Network System

On top of the network so far outlined a network operating system can be built from a collection of services. Such a system is responsible for creating new services dynamically and for providing services of general utility.

The services required before dynamic services* can be created, plus the

---

* Dynamic services are services which have a fixed address during their (typically quite short) lifetime but which do not necessarily come into existence at the same address each time they are created.

services providing general utilities are together referred to as **kernel** services. It will be shown how dynamic services may be created and used by describing the necessary kernel services.

2.5 System Aims

Distribution brings with it a certain number of advantageous features:

(1) Addition of extra resources - extensibility.

Given that the electrical connection of an extra resource to a network forms part of the network's existing technology the adequacy of the resource's interface to the rest of the system determines the difficulty of adding it to the network. To simplify this task these interfaces should be designed with their duplication in mind.

(2) Fail soft characteristics.

If resources are deliberately duplicated, in the manner recommended above, the system will show a "fail soft" characteristic so long as the duplicated resources are not mutually dependent. (That is, a failure of one component in the system will tend not to cause the whole system to fail). When distributing the functions of an operating system the best system will result from the use of the fewest interdependencies.

(3) Autonomy.

As an extension of the above principle each service should be as autonomous as possible. That is, in the extreme, it should be possible for a service to operate in complete isolation - no other service being dependent upon it and it being dependent upon no other service. In particular entire network systems (should there be cause for more than one of them to coexist) should display this property. A system in which each component machine is obliged to run a "kernel" of controlling code, for example, is not ideal in this respect because it does not leave each computer free to execute independently.

(4) Modularity.

A certain degree of modularity is enforced on a system that is built from services that only communicate via a network; however, modularity is not the exclusive property of a distributed system - it is merely more difficult to fabricate a system that is not modular if it is distributed.

(5) Protection.

Domains of protection will tend to be available on a network, even one which consists entirely of unprotected machines (when, because it cannot be actively accessed by any of the others, each machine constitutes a separate protection domain). Note that services do not necessarily require a protection domain of their own. A service was defined as something which accesses none other than its own local information (and information sent to it by other services). A protection domain is only necessary in the cases in which services cannot be trusted to conform to this rule.

In conclusion, the kind of system preferred is one in which (1 and 2) duplication of function is possible and perhaps common, (3 and 4) system function is not spread thinly over each processor but is provided only in a few system services, and (5) the available protection domains need only be used sparingly. Further considerations on the nature of distributed computing are reported in [Peterson79]. Relatively simple processors could be used and the system would be naturally modular in structure.

2.6 Conclusion

The environment described consisting of services, objects and some form of communication is typical of a very wide class of existing network systems. As a model it conforms, for example, to the telephone network in which the analogies are as follows:

service - A telephone subscriber.

server - A subscriber prepared to work for a caller, for example: "directory enquiries" or "dial-a-disk".

entry - One of the things that a subscriber might be able to do for a caller.

communication - The telephone network.

objects - Some objects can be transferred by 'phone (pictures for example) but, by and large, references to them must be sent, and there are only ad hoc methods for doing this. For example: giving the object's name and relying upon voice identification or a credit card number for security.

A similar analogy holds true for the postal system but, naturally, the application of the model to a network of computers is of primary interest. A solution to the problem of "objects" (above), for example, that applies to the model would provide solutions in any of these examples. The likely structure of such a system and some guide lines for its production have been given which relate not to a particular system but to any system constructed in the environment outlined.

Article 3

ENVIRONMENT IMPLEMENTATION

The representation system described in the following articles was implemented at Cambridge University Computer Laboratory on the Cambridge Ring (which will be subsequently referred to as the "Ring"). There follow some salient points:

3.1 Hardware

Details of the Ring's hardware design are speculated in [Wilkes75] and reported in [Wilkes79]. It is compared with various other local area networks in [Hopper78]. There are two points relevant to this dissertation. The first is that, using the standard hardware, the number of the station* from which data was sent is available and can be considered incorruptible. Thus, since there is no "broadcast" mode of operation, the recipient of data can be assured of its private reception.

The other major facet of the Ring's hardware is its very low error rate. The hardware itself will rarely corrupt data. There are no official measurements for this error rate but [Needham80] suggests that one bit in every 10^{10} or 10^{11} may be likely. This has an effect upon the development of protocols since, in general, quite large blocks of data can be sent with negligible risk of corruption. Error detection mechanisms are necessary in these protocols (because of the possibility of an error: however remote) but elaborate error correction mechanisms are not. Because it is necessary so seldom, it is possible to retransmit large amounts of data which are found to be corrupt with only a small loss of overall efficiency.

---

* "Station" is the name given to the hardware interface to the Ring. There is normally one station attached to each computer.

3.2 Protocols

There have been two main protocols that have been developed for the Ring. The first, called BSP (Byte Stream Protocol)* [Johnson79], provides a bi-directional byte stream with flow control, which has been used as a base for file transfer and virtual terminal protocols. The second, called SSP (Single Shot Protocol) [Gibbons80b], allows a very simple remote procedure call in which a block of request data (of limited size) is sent and a reply block received. Both of these protocols are based upon BBP (Basic Block Protocol) a description of which can be found in [Walker79].

SSP is used where possible because of its simplicity. BSP is used in cases in which a very low error rate, a continuous stream of data or flow control is necessary.

An interface to a service that uses SSP entries needs careful consideration. Each entry must be **idempotent** (which implies repeatability). This is because, although it is unlikely, either an SSP request or an SSP reply may be lost (through data corruption, for example). In the case in which the sender receives no reply, therefore, it will be impossible to know whether or not the requested operation has actually been done. If the operation is idempotent repeating it will cause the expected result in either case. Basically, operations must have post-conditions that are expressible in terms of known constants and the parameters given with a request. For example, "delete the last element of the list" is not idempotent because its post-condition involves a variable local to the service (essentially the "end of the list" variable) which is not supplied as a parameter. However the two entries "tell me what is at the end of the list" and "delete thing X from the list" are both idempotent, the first because its post condition is just the same as its pre-condition and the second because the post condition "X does not exist in the list" involves only constants (the list) and parameters given with the request (X). Note that what is achieved by the former entry, which is not idempotent, can

---

* The use of BSP to provide an "authenticated byte stream" is discussed in the article called "Authenticated Communication".

approximately be achieved by these last two#. A service's interface can very often be made idempotent in this way.

3.3 Addressing

A service's address, in general, has three parts as follows:

- station number

  An eight bit number identifying the station that the service uses to communicate with the Ring.

- port number

  A fourteen bit number identifying a port on which a service is listening. Ports are software created abstractions. Traffic arriving at a port on which no service within a machine is listening is disregarded.

- function code

  In order to distinguish the different reasons for which traffic may arrive at a particular port a 16 bit function code number is used. Both the port number and the function code are necessary in general before a unique service can be identified. This is due to the operation of a typical multi-processing operating system in which a single port may be isolated for the purpose of initial connection and the function code used to distinguish the service to be (created and) used.

These three address parts can be found by presenting the textual name of the service to a server called the nameserver.

The use of any service is expected to be preceded by such a name lookup. Thus it is not necessary to "publish" the addresses of services or to build their addresses into other services that use them. In order to relocate a service it is only necessary to change its entry in the nameserver.

This is only approximately true since the X deleted may not be at the end of the list for one of two reasons: firstly X may no longer be at the end of the list since additions may have been made; and, secondly if X is not unique the X deleted may not be the one originally found at the end of the list. In point of fact "delete the last element from the list" will require an interlock of some kind over the list for the duration of the update. The practical difference between such an interlock and the concept of a "connection" is negligible (in that a "connection" can be thought to exist in the time during which an interlock is extant).

The only service address that must be known on the Ring is that of the nameserver - all other addresses can be found through it.

3.4 Machines

The Ring has a few large and statically configured computers offering multi user services. These include a VAX running UNIX, a PDP-11 running a locally enhanced version of RSX 11M, a research machine called CAP running a capability based operating system and the University's large mainframe, an IBM 3081.

In addition there are a number of uncommitted minicomputers and microcomputers which are allocated upon request and which run a local operating system called TRIPOS*.

The remaining processors on the ring are mainly Z80 microprocessor systems of two basic sizes (for a description of their development see [Gibbons80a]). These machines will normally provide only one or two services and run in a completely dedicated mode. They possess an extensive booting and debugging environment described in [Ody79] and can be programmed in Algol68C. The services provided by these machines are static and, by and large, very reliable. A few Z80s are reserved for development work.

3.5 Existing Services

In addition to the services to be described in the following articles the Ring currently possesses a large number of both static and dynamic services, the former being continuously available and the latter being available only at times of demand. The dynamic services, which include connection to and file transfer to and from dynamically allocated machines, are currently run on an ad hoc basis. There is as yet no management apparatus generally available for them.

---

* Some work on TRIPOS is described in the article called "UID sets on TRIPOS". TRIPOS itself is fully documented in [Knight82].

The static services include the nameservers (above), a resource manager, two file servers, several terminal concentrators, and services for controlling I/O devices attached to the Ring, such as a Diablo terminal, a line printer, and a pointing machine used for wire wrapping.

A terminal concentrator [Ody80] gives its users the ability to "connect into" different services using a virtual terminal protocol called RATS (based on BSP). Several connections to each terminal can be handled concurrently. Recently facilities for making a connection to the terminal concentrator from the Ring have been added.

The resource manager manages a set of LSI4 minicomputers and MC68000 microcomputers jointly referred to as "the processor bank". A user can connect into the resource manager and, using a small command language, request a machine, conforming to a certain set of criteria, and a system to run on it. A RATS byte stream is then routed from the allocated machine (if one could be found and was successfully booted) and "reverse connected" into the terminal concentrator being used. The same machine provides a service which will perform the allocation and loading according to default criteria and a default system (the TRIPOS service).

The processor bank has two servers (one for each type of allocated machine) responsible for initially booting the machines and providing a hardware independent interface to the resource manager. Each service is called an ancilla (from the latin for "housemaid").

The fileservers [Dion81] are some of the few machines on the Ring with disc storage devices. They provide a modified SSP interface for the manipulation of files and directories - which are represented by 64 bit unique identifiers. Operating systems are free to use the interface to build any of a range of filing systems from a simple one-level directory scheme to a quite general scheme in which graph like directory structures are allowed. The service is used by both the CAP and the TRIPOS operating systems to provide filing systems. In addition CAP uses it for swapping. The simplicity of the SSP interface enables files to be used moderately easily even from the servers provided in Z80 microprocessors.

3.6 The Cambridge Distributed Computing System

The Cambridge Distributed Computing System is the name that has been given to the assembly of the above components. Some of the underlying philosophy regarding the precise way in which they have been put together to form a coherent whole can be found in [Needham78] and a first attempt at describing the consequences of its early development is preserved in [Wilkes80]. By far the most comprehensive description of the system is given in [Needham82].

Article 4

DESIGN OF AN OBJECT REPRESENTATION SYSTEM

This article addresses itself to the making of references to objects on a network. After some discussion during which the nature and meaning of an "object representation" are deduced, the component parts of a mechanism for creating and using them are isolated. The properties of each of these components are then considered separately and additional problems arising from their synthesis into an object representation system are finally resolved.

4.1 Representation Of Objects

Once some mechanism for representing objects is available, it can be used to provide obvious solutions to problems which can seem difficult, without such a mechanism. For example, in the design of resource management and allocation mechanisms, the designer may become bogged down with questions relating to the maintenance of a binding between an allocated resource and its owner. An obvious solution in this case is to give each resource some representation and simply give the named resource (no matter how abstract) to its new owner*.

One of the more fundamental problems about such a scheme, and one which is pertinent to the above example, is to ensure that a representation is authentic - that it really does represent what it is expected to, and is not devoid of meaning. (The resource allocation system above must ensure that only authentic representations of its resources are used, in order to prevent services from fabricating their own).

A certain number of words have just been used which ought to be discussed in more depth:

---

* The question of resource management and allocation is dealt with in more detail in "Resource Management".

"name"

When a name is used it means a particular object (if it means anything at all). This "meaning" is all that is necessary in a name. A name refers to only one object and, if ambiguity is to be avoided, an object should have only one name. For as long as an object exists, and even before it has come into existence and after it has gone out of existence, a name can be used to refer to that object. It is permanent, unique and it identifies.

"authentic"

What facet of a representation is it that may not be authentic? What is in doubt? Clearly a name is always authentic in the sense that it always refers to a particular object. Authenticity must be a property of something other than a name. There is something about the representation which must be valid.

In the above examples, if a resource is represented it is essential that the representation used is one that was made valid by the resource allocator. Trust has been bestowed on this representation and it is in this sense that it is "authentic". Being authentic means being trusted by something. Notice that being authentic is not an absolute quality - it has no meaning unless there is a trusting "authority". It is a relative quality: being authentic implies that there is something bestowing trust - if this is not true then being authentic has very little meaning (and even less value).

"representation"

A representation is something with an identity which can be seen to be authentic. As discussed above, this authenticity must be conferred by something in order to have meaning. A name, as already outlined, identifies the object to which it refers and so can be used for "identity" in the above sentence. Hence where an object is represented it must be possible to verify a statement such as "<something> says that this object is identified with <name>". It is this statement that must be valid in the paragraph about "authentic" and it is the ability to verify the statement that distinguishes a representation.

Having decided upon a "working definition" of what a representation actually is, a method of implementation can now be investigated. It is necessary to be able to pass something akin to the statement "this object is identified by <name> : <something> says so" from service to service and be able to have it verified.

It is convenient, at this point, to introduce some notation for the above statement, since in later articles it is to be used quite often. "<something>" refers to the name of the authority that conferred validity upon the representation. It is the identity of the representation's authority. This name will be referred to as the representation's **authentity** (a contraction of "authority identity"). The statement itself is a proposition involving a name (N) and an authentity (A): it will be represented using the notation

p(A\N)

The problem of finding something which can be passed from service to service and which can be used to verify p(A\N) can be partially solved by considering the properties of such a token.

- It must be unforgeable.

  It is essential that no other token can be counterfeited to verify p(A\N). Otherwise those who own a true representation for an object will be indistinguishable from those who have merely manufactured the same token. This implies a certain amount of privacy. It is necessary to ensure that those not entitled to a representation cannot steal one from a bona fide owner, either by copying or by guessing. The invalid copying problem does not apply to a service's local data, by the definition of "service" which stated that they access only their own local data and data communicated to them. During network communications, however, BISCM is relied upon.

- It is distinct from but related to p(A\N).

  p(A\N) is just a proposition. Any proposition can be made by anyone - regardless of its truth. Any service could claim that p(A\N) is true. It is not, of itself, sufficient as an implementation of a representation. Something else is necessary, something which implies p(A\N). Clearly the token is related to p(A\N) and although there must be some way to verify this relationship the actual token and p(A\N) are distinct.

- It is temporary.

  The representation of an object needs to exist on a network whilst the object is in use. For example, the representation of a user need only exist whilst that user is logged on*. If the user is not active on the network in any form, then it could be contended that there should

---

* User authentication and representation are discussed in the article called "User Authentication"

be no valid representation for the user which would imply that the user does, in fact, currently exist at all. There is a clear distinction here between the user's representation, which will exist (relatively speaking) only transiently, and the user's name, which is permanent.

The following picture begins to emerge. The token passed about the network to represent an object N under A's authority must be unique and unguessable. It is temporarily and verifiably related to the proposition "this is object N: A says so", p(A\N).

To formalize what has been discussed some abbreviations are introduced. If a token for an object's representation and p(A\N) are related then, mathematically speaking, there exists a relation that combines them: call it V (for verify). Thus a token t represents object A\N (that is, the object called N under A's authority) if and only if

$$t \; V \; p(A\backslash N)$$

(read "t validates the proposition that it represents object N under A's authority").

In conclusion, in order for objects to be represented on a network, the following are necessary.

- A naming system (for A and for N).
- Unforgeable tokens (for t).
- A relation (for V).

Each of these elements is now discussed in detail.

4.2 A Naming System

It is necessary to design "names" which will be used to identify objects both uniquely and permanently. Such a scheme is referred to as a **global naming scheme.** Since these names are permanently allocated, if they are to be unique, a service which generates new names (as is proposed) must never give any name that has been given previously. This has two implications:

i) There must be a very large name space, so that there will be enough names never to "run out".

ii) There must be some way of determining whether a new name has been generated before.

i) can be solved simply by using a lot of bits to represent a name. ii) is complicated by the potential existence of several different name producing services on the network either concurrently or at different times. It is necessary for each service to know about not only the names that it has generated, but also the names that all the other name producing services have produced (that ever were or are). Dividing the name space up into a fixed number of equal parts and assigning each part to a different name producing service resolves this problem. This can be implemented by reserving a field in names to identify the service that produced them. Thus each service is free to look after only its own name space.

A simple name management scheme for name generators uses a well ordered set from which names are chosen. It is then only necessary to remember the last name generated, generating its successor when the next name is requested. In this scheme the names that have already been produced will always be known (they will be all those between the first and the last generated).

4.3 Unique Unforgeable Tokens

Since there can be no constraints enforced on services or on communication*, the existence of tokens cannot be controlled. This is true in the sense that nothing can be done to stop such tokens naturally arising in services' local information and being subsequently transmitted around the network. It can be made arbitrarily unlikely that such a token arises naturally, even if it is being created by a dedicated and malicious service, by randomly choosing new tokens from a suitably large name space. If the choice is truly random then a service, no matter how methodical the algorithm, will have one chance in (however many names there are in the name space) of guessing a given token (at each attempt).

---

* This is necessary if the system aims expressed in "Base Network Structure" (the provision of autonomy in particular) are to be fulfilled.

Thus the size of the name space can be made large enough to ensure that each token is as secure as desired. For example, it can be chosen so that a correct guess is less likely than catastrophic failure of the network.

This method of generating unforgeability will also go part way to ensuring that each token is unique. Tokens are temporary and, at any given instant, there is likely to be only a small number of them relative to the size of the name space. The chances of choosing a new token at random which is identical to another valid token might be conveniently small. If this is not so it will be seen that it is possible to check newly generated tokens for uniqueness.

Note that "unforgeable" in the above sense applies only to those not already in possession of a token. It is impossible, and undesirable, to stop the possessor of a token fabricating another exactly like it. Similarly it is both impossible and undesirable to stop the possessor of a token passing it on, over the network, to whom it pleases. Such activities are "desirable" because they are the very properties that we wish to encourage in a token and the representation that it implements.

4.4 A Relation

A relation between the above tokens and (authentity, name) pairs must be implemented. This relation will be expected to change as the tokens, which are temporary in nature, come and go. In order to implement such a relation it is merely necessary to provide a service on the network which will verify or deny that a given token is related to a particular (authentity, name) pair.

Such a service could maintain a "relation", in the relational database sense of the word, with domains 'token', 'name' and 'authentity' - each tuple representing an association of one particular token to an (authentity, name) pair. Verifying that a token corresponds with a particular authentity and name simply consists of checking that the relevant tuple exists within the table representing the relation.

TOKEN	AUTHENTITY	NAME
:	:	:
t	A	N
:	:	:

A table for the relation "V".

Creation of an object representation is implemented by creating a new token and inserting it against the name and authentity of the created object in the table. Deletion of an object representation corresponds to deleting that tuple. Note that upon the deletion of a tuple each of the tokens on the network, wherever they are, cease to be of any use. Such deletion is similar to the revocation of capabilities*.

4.5 Control of the Relation

The need to create and delete tuples from the table brings with it some access control problems:

a) Who should be allowed to create tuples?

b) Under what circumstances should a tuple be deleted?

4.5.1 Tuple creation

If the identity of the creator is put into the authentity field it is not necessary to put any constraint upon who it is that may create tuples. However, if the identity of the creator is to include its authentity (and so on), multi-part names would have to be utilized (which would be both difficult and inconvenient#).

---

* In point of fact not many capability based operating systems actually implement revocation.

\# This topic is discussed more fully in the article "Variants in AOT Design".

If not everyone is allowed to create tuples there must be some criteria defining the set of things which are allowed to. The ability to create object representations (each marked with the same authentity) in some ways resembles the ability to create objects of a particular **type**, in a general sense. Using this comparison the type is given by the authentity of the generated object representation. The ability to create a particular kind of object can be imagined to be due to the possession of a type. It is this possession that distinguishes those allowed to create tuples. It would be possible to recognize such a possession if there were a representation for "type" objects (that is, if there were objects of type "type"). Such a representation can be created with the existing apparatus - all that is necessary is the definition of an authentity for "type" for use in the representations of objects which allow the creation of other objects (of a particular authentity). This authentity will be referred to as **'auth'**. Thus, in the nomenclature developed above it is possible to create the tuple for

$$x\ V\ p(A\backslash N)$$

if a token y which proves

$$y\ V\ p(AUTH\backslash A)$$

can be given.

Note that object representations for types (that is, with authentity AUTH) themselves form a type, members of which can be created if a token z which proves

$$z\ V\ p(AUTH\backslash AUTH)$$

can be given. Such a token obviously has a special part to play in a representation system and it is discussed in the article called "The Source Of All Power".

4.5.2 Tuple deletion

Practically speaking, one of the reasons that a tuple is deleted will stem from the table maintenance software itself. Since no constraints are put on the rest of the network, it will be impossible to guarantee that anything ever deletes the tuples created in the table. The maintenance code must, therefore, have a criterion for deciding when a given tuple has fallen into disuse, and delete it itself, in order to prevent the table becoming full (a practical rather than a logical problem). Such a criterion can be

provided by the association of a "timeout" with each tuple. That is, some number denoting the amount of time for which the relationship between its parts will hold.

There are two ways that such a timeout could be operated. The first is to give a tuple a timeout longer than the representation could possibly exist leaving the timeout to decay until the tuple is explicitly deleted (in some manner yet to be decided). The second is to give a tuple a timeout just longer than the time after which the existence of the tuple is required to be reasserted. The continued existence of the tuple is then reasserted by changing its timeout back to the value from which it has decayed (also in some manner yet to be decided)*. By providing a mechanism for changing a tuple's timeout either of the above methods of control can be employed to ensure that a tuple exists during a required period of time. This also provides a way of deleting tuples since the tuple will be deleted if its timeout is set to zero.

Returning to the original problem ("who should be allowed to delete tuples ?", mentioned at the beginning of this section), the problem has been restated as "who is allowed to stop a tuple from not being deleted ?". The question can now be simplified to "who should be allowed to change a tuple's timeout ?" - since by continually resetting it to a strictly positive value the existence of the tuple will be maintained and by setting it to zero the tuple will be deleted.

In order to decide who it is that is to be given the ability to delete a representation it is necessary to investigate some different possibilities.

An obvious suggestion is to give the creator of a representation the unique ability to delete it. That a representation with some authentity A could have been created by a representation holder can be verified by checking that a token t can be provided for which

$$t \; V \; p(AUTH \backslash A)$$

Such a mechanism would allow any creator of A authentity representations to delete any A authentity representation - whether or not it had created that representation initially. In some ways this solution side steps the

---

* The point of this being to make sure that the tuple dies fairly quickly if the thing using it dies.

issue, since each creator would have to decide which customers it should delete representations for. This is the same as the original question applied to the creator instead of to the table maintenance software.

Another solution would be to allow the possessor of an object's representation to be able to delete it. This would seem to be in general keeping with the capability nature of a token. However, this has undesirable consequences in practice. For example, consider the case in which a shared object is to be given to several customers. Each of the customers is given some representation of the object to prove that they are allowed to use it. Using this solution each of the customers could potentially delete the representation that all the others are using - which may be undesirable. This solution makes central control of the existence of representations impossible.

Since the ability to use an object is represented by a token that can be passed from one holder to another a distinction is necessary for the original owner of that token. The crux of the problem stems from the difference between the ability to use an object (as represented by a token) and the ownership of the ability to use an object. Something which proves the former should not necessarily prove the latter.

The problem would be solved by considering the tuples as abstract objects themselves - each with a name, authentity (e.g. the table maintainer) and a token. The token would be used to prove that a named tuple was owned and could therefore be deleted and so on. For example when the tuple which proves

$$t \; V \; p(A \backslash N)$$

is created, the tuple is named ('tuple' say) and is represented with a token d under the authentity of the tuple's creator ('table' say). That is,

$$d \; V \; (table \backslash tuple)$$

If the tuple for t (above) is to be deleted, its representation, d, must be presented. Thus, in a sense, d is the representation of a representation.

Investigation quickly reveals that this solution has some rather severe failings. For example, who is allowed to delete d? This question leads to a hopelessly recursive solution. A further disadvantage is the necessity to name each tuple created and to bind the name with the tuple to which it refers.

Since each tuple needs such a representation it is profitable to consider the representation of a representation as a case apart and not use the more general method outlined above. The recursiveness of the above solution disappears when viewed in this light. The ability to delete the ability to delete a representation (to take the first recursive example) is unlikely to be useful and is certainly not necessary. An authentity for this representation is likewise not necessary since it is always the same ('table' in the above example). Hence we could implement this solution by having two sets of tables - one for normal representations, each tuple with an object's name, authentity, and representing token, and another for representation representations, each tuple with the name of the representation (a tuple in the first table) and a token representing it. Such an implementation could clearly be optimized by associating the representation's representation token directly with the tuple to which it refers and omitting the names altogether. That is, instead of

"t V p(A\N)" - call it 'tuple'

d V p(table\tuple)

we have something, avoiding the intermediate name 'tuple', similar to

d V p(table\"t V p(A\N)")

Using the notation i(t,A\N) to represent the proposition 'the table maintenance authority says that this object is identified by the statement "t V p(A\N)"' we can write for the above

d V i(t,A\N)

Note that, since the 'table' authentity is trustworthy

i(t,A\N) => t V p(A\N)

so that

d V i(t,A\N) => t V p(A\N)

We now have a full solution to our question "under what circumstances should a tuple be deleted ?". It is:

- if the tuple's timeout value reaches zero

This timeout value is changed in these two ways:

1) it is decremented by 1 every second

2) a request is received to change the tuple for "t V p(A\N)"'s timeout is accompanied by a token, d, for which d V i(t,A\N)

Thus for the object A\N above there are two tokens, t and d. The token t is used to prove

$$t \; V \; p(A\backslash N)$$

so that the holder can use A\N, and d is used to prove

$$d \; V \; i(t,A\backslash N)$$

so that the holder can maintain (or delete) t.

When a new tuple is generated two new tokens will be fabricated - one for the representation and the other to maintain that representation. The authority which created the representation need not pass on the maintenance token if central control is necessary.

4.6 Relation Representation

Returning to the design of the relation, a table with tuples to represent things of the form

$$t \; V \; p(A\backslash N)$$

is necessary. With each such tuple is associated

1) A timeout, Δ

2) A token, d, which proves

$$d \; V \; i(t,A\backslash N)$$

used to maintain this tuple

hence such a tuple would be represented

$$\langle d, t, A, N, \Delta \rangle$$

and the table would consist of the set of tuples:

$$R = \{ \langle d, t, A, N, \Delta \rangle : \Delta > 0 \;\; \& \;\; d \; V \; i(t,A\backslash N) \}$$

the service for maintaining such a table will need entries for at least the following:

1) to prove t V p(A\N) for some t and A\N

2) to create a tuple <d, t, A, N, $\underline{\triangle}$> for some A\N and $\underline{\triangle}$ returning d and t

3) to change the timeout of a tuple <d, t, A, N, $\underline{\triangle}_1$> to $\underline{\triangle}_2$ for some d, t, and A\N

Article 5

THESIS CONTEXT

Now that an introduction to the topic of this dissertation has been given this article discusses existing work in the areas related most closely to it, and considers its position with respect to the literature.

5.1 Introduction

The most closely related area is information protection. In [Jones78b] Jones positions protection mechanisms as a sub-goal to the controlled dissemination of information as defined by a set of **security policies.** She takes the view that the former merely represents the means whereby the latter can be enforced. Security policies can be enforced in at least two ways. Firstly, they can be enforced by employing particular protection mechanisms, in which case the potential information thief is simply unable to infringe a given policy. Secondly, they can be enforced using detection and legislation, in which case the potential information thief is not necessarily unable to infringe a given policy but will be "punished" if the fact that he has done so is detected.

A compact tutorial on protection can be found in [Saltzer75] which includes an extensive reference section. Denning and Denning in [Denning79] further subdivide protection mechanisms into four orthogonal areas which they call access control, flow control, inference control and cryptographic control. Access controls prevent the use of entries at access interfaces at which some arbitrary entry criterion is not correctly asserted. Flow control prevents the copying or derivation of information from an information set of greater confidentiality to one of lower confidentiality. Inference controls prevent the deduction of information that is otherwise explicitly denied. Cryptographic controls prevent information which cannot be protected in any other way from being meaningful. It is the first topic (access control) with which this dissertation is chiefly concerned. Flow control, inference control and cryptographic control are not to be of primary interest.

There are a good number of access control models at large, many of which have led to implementations (with varying degrees of success). Most of these implementations have been for an environment in which there is a single large processor, or possibly several tightly coupled ones all of the same type. However, the falling cost of computing elements and the resulting distribution of computers has promoted the formation of computer networks and, as argued in [Kahn72], the resulting level of resource availability is bound to give rise to distributed systems capable of controlling these resources. These distributed operating systems will be faced with exactly the same kind of access control problems as their mainframe counterparts. But at the same time, the importance of adequate information control in the computing elements that make up a distributed operating system will diminish, since functions for which sharing is necessary (the primary cause that makes information control mechanisms necessary) will become less common within individual computers. For example, there will be a tendency for single user systems to be used using either a personal file server or one available from the network. Information control within the single user system will in either event become less necessary.

One relevant difference between conventional operating systems and distributed ones is the forced differentiation of names and hardware addresses. When originally proposed in [Dennis66] capabilities were unforgeable object names protected by an operating system and used for access control. Fabry, however, later proposed an efficient way in which they could be used for addressing in [Fabry74]. In consequence the distinction between an object's name and its hardware location, in some capability operating systems, has become very slight. It would be extremely inadvisable for a distributed operating system, in which provision must be made for several different kinds of computing element, to attempt to name its resources in terms of the different addressing structures local to each component machine. In particular, a distributed operating system can have no direct control over accesses that any particular processor makes to its own addressing domain. Thus the highly efficient mechanisms developed in operating systems to cope with memory access control can not be realistically considered in a distributed operating system. Access controls are likely to have to be applied at rates determined by the network's communication protocols, rather than at hardware memory access speeds. The implication here is that access controls could possibly be provided by software in the distributed operating system, not necessarily by hardware or firmware.

5.2 Models of Security

As mentioned above there are several existing models for security mechanisms, one or two of which are described here.

One of the most influential models in use is derived from [Lampson69] in which accessing and accessed objects were identified as **subjects** and **objects** respectively. The term **domain** was defined as the set of objects accessible by an active entity at a particular instant. This model was presented in [Lampson71] and the concept of an **access matrix** was explained. In such a matrix each column bears the name of some object and each row the name of a subject. At the intersection of a particular row and column there is a set of access attributes which specify the types of access that the subject has to the object. The model is not suitable for literal implementation because the size of the matrix would be inhibiting - even though the actual amount of information being stored may not be too large to handle (most subject/object intersections will contain the null set of access attributes). Two main classes of implementation of this model were already well known: **access control lists** and **capability lists.**

An access control list is kept with each object. It denotes which subjects are allowed what access to that object. When a correctly identified subject requests access to the object its name and the requested type of access are checked off in the object's access control list and the access is disallowed if the check fails. This corresponds to keeping each column of the access matrix with the object to which it refers (naturally subjects with default access to the object need not be included in the list).

Capability lists were first investigated in [Dennis66] in which primitives for their manipulation were also proposed. A capability list is essentially the reverse of an access control list. A list of objects and the allowed access to them is kept with each subject. The resulting list then becomes a tangible implementation of a domain. An object itself may not have any active part and thus be unable to check the existence of a capability from this list on each access. Since it would clearly be incorrect for the check to be done by the subject it is necessary for operating systems to provide direct support for capability lists if they are to be used. At each access of a subject to an object an operating system is responsible for ensuring that the capability used from the capability list is correct and contains the appropriate access attribute. A mechanism whereby capabilities could efficiently be revoked (invalidated) was first

described in [Redell74]. Revocation is discussed under "Protected Object References" below.

In their exposition of the access matrix model Graham and Denning [Graham72] underline the necessity for correct identification of the subject in any implementation. If a subject's identity could be forged in either the access control list implementation or the capability list implementation security would be seriously undermined (the subject's identity is needed in a capability list implementation to verify ownership of the capability list used). Fortunately, within a single machine, it is not difficult to provide the names of active subjects with unquestionable authority. In many capability list implementations the identification of a process, which either actually is or is acting on behalf of a subject, is provided by the hardware or firmware. In a distributed operating system, as stated above, overall control of memory accesses within any particular computing element can by no means be counted upon to ensure the unforgeability of a subject's identity. Even if it were possible it is doubtful that exercising such control could be made very efficient. In this environment the generalization that all subjects will be represented as or by a process becomes useless because each component computer in the network is liable to support this concept in different ways - some may not support it at all (hence the definition of a service, given in "Base Network Structure").

A widely respected model of computing is provided by the abstract object model, in which each abstraction is represented by a body of code providing a fixed interface to its clients. Internal details of the implementation of the abstraction are actively hidden. A description of this model and the way in which it naturally facilitates the style of naming and protection that tends to emerge from the use of capability lists can be found in [Jones78a]. Such a model of computing is likely to be realized in a distributed operating system because of the enforced hiding of implementation details provided by the existence of programs in physically different kinds of computing elements and the necessary provision of an interface where a computer meets the rest of the network. The adoption of this model may well invalidate the assumption made earlier when discussing objects in a capability list implementation. Each object may well have an active part associated with it (in order to present the relevant abstract object interface) so that the provision of an intermediary to validate capabilities would become unnecessary - it could be done at the object's interface. It is interesting to note that this approach avoids many of the

issues dealing with the question of what to do when a conventional operating system evaluates an incorrect capability, by delegating the decision to the object (which is, after all, a more interested party). The possibility of objects performing all access checks begins to blur the distinction between the access control list approach and the capability list approach. The difference is that the access control list approach dictates that the object receives an unforgeable identity of the subject accessing it which it must validate and the capability list approach dictates that the object receives a capability to perform a particular function that it must validate (this could be considered to be an unforgeable identity of the operation to be performed). The representation system discussed in "Design of a Representation System" provides names that can be validated in general so that access controls using either the access control list or the capability list approach are possible.

Access control models can be divided into two parts denoted as **discretionary** and **non-discretionary** respectively. The former correspond to the access matrix model in which the access matrix is itself considered as an object so that it is possible for subjects to alter their (and other subject's) access to objects (in controlled ways). The latter corresponds to the case when the access matrix is not amongst the set of objects used. In point of fact an entire system cannot be precisely non-discretionary because some operation on an access matrix is implied when a new object is created. The best known example of a non-discretionary system is the military system in which each object is given an access category (top secret, secret, confidential or unclassified, for example) and each subject is given a clearance level which must exceed or equal the access category of the object being accessed. In addition access is further restricted by a need-to-know criterion in which only certain subjects may access certain objects - such as is easily modelled by an access matrix. The initial creation of the need-to-know lists and the initial access category of an object are not easily automated or formalized since, in reality, they are based on value judgment about both the subject and object. An attempt, however, was made to implement a data security system conforming to military standards in the ADEPT-50 operating system [Weissman69]. Karger argues in [Karger77] that formalisms can be made of non-discretionary systems that cannot be made of discretionary ones and he investigates the application of non-discretionary systems to the decentralized computing environment in general.

A good summary of current activities in the general field of information protection in a distributed environment can be found in [Davies81]. A short summary of the take-grant system developed by A. K. Jones can be found there. This system consists of a graph of nodes representing either subjects or objects connected by directed arcs labelled by the access attributes that one end has to the other. Access attributes include take, grant, create, and remove rights, through which subjects may manipulate access rights to others. Subjects are allowed to exercise their access rights directly whereas objects act as passive holders of access rights which can be manipulated by subjects. The model is a specific example of a discretionary access model which could be implemented using an access control matrix. Its benefit is in the choice of access matrix access primitives. Jones has shown that the potential access of one node to another can be decided in linear time if these primitives are used. This is not true for more complex models (such as those used in other discretionary access control implementations) whereas non-discretionary access control mechanisms have been advocated in the past because such decisions are trivial.

The number of primitives provided for the manipulation of the access matrix itself has been subject to consideration by others in this general area. In particular work has been carried out in the provision of verifiable security kernels on the grounds that the security of a large operating system is not easily verifiable while it may be possible to certify a small core of code dealing with the provision of security primitives. A small set of primitives is therefore desirable. This line of investigation has resulted in the development of several mathematical models for security. Except for the fact that only a few simple and mathematically specifiable operations are used in the representation system developed (which only forms a basis for an access control system) this work is largely tangential to this dissertation.

5.3 Authentication

Authentication, the act of proving a subject's identity, is used at several levels in a conventional operating system. It is most manifest to a user when he logs on and has to quote a password or offer some other credential. However, as mentioned above, authentication also happens at a much lower level within a computer - namely whenever a subject accesses an object in an invocation of the access control mechanism. In modern capability systems this may happen on every access to a memory location.

Authentication in this case could be provided by the presence of the object's name in a special hardware register. Thus, things both within an operating system (such as processes) and outside it (such as users) undergo authentication. In a distributed operating system there is yet another level corresponding to things within the distributed operating system but outside individual component operating systems. That is, there is a requirement for things that have been authenticated to the distributed operating system to be subsequently re-authenticated to component computers so that, for example, a user can log on only once to the distributed operating system which authenticates him to each component machine.

Because distributed operating systems are not yet common and the latter requirement is not, therefore, widely needed there is very little in the literature about it. The authentication internal to an operating system is rather too trivial to receive any specific attention hence most work is focused on the authentication of objects external to an operating system. When put into the context of a network this work, by and large, merely extends these methods.

Using passwords for authentication in a network has several disadvantages. In a remote job processing application, for example, changing passwords quoted in stored job images can be difficult if the job image is not to be invalidated. Naturally, any difficulty in the routine replacement of passwords prejudices the security they provide. The problem of a single user having to use several passwords is well known. If they are all different he is likely to write them down somewhere - which is highly insecure - and if they are all the same he is liable to be extremely reluctant to change any of them as often as is necessary.

One solution to these problems is proposed [JNT80] by the Job Transfer Protocol working group of the Data Communications Protocols Unit set up by the Department of Industry in which an audit trace is built from the name of each computing element that has passed on an indication that a correct password has been checked. The first name in the audit trace is the name of the element that made the check. Elements performing intermediate communication are merely required to add the name of their immediate sender to the list of names already in the trace. If the recipient trusts the authorizing computer and each of the others in the list it can elect to take this as sufficient proof of the initiator's identity, otherwise there is a provision for the use of passwords directly. The scheme relies on the ability of one element to authenticate another through its address. It is

assumed that the address of the sender can be correctly obtained and that this address is sufficient to identify the immediate sender (which may not be true if dynamic services are in use).

The transport service proposed for use in the above is, in fact, the one prepared by study group three of the Post Office PSS User Forum [Yellowbook80] which has no provision for authentication built into it. (Authentication is listed under "Points for further study"). The necessity for authentication at this level has since become apparent and the resulting modification to the protocol is described in "Authenticated Communication".

Karger [Karger77] describes a mechanism which he calls "proxy login" whereby a user of a particular computing element has the ability to let specified users of other computers have access under his name. On his authorization the computer will allow another computer to log on by authenticating itself and giving the name of its user for whom a session is to be initiated. Assuming the user's name is one that has previously been authorized, access is then allowed in the authorizing user's name. The intention here is that a user should allow access to his resources to himself as identified on other computers. Thus a user, if he has set everything up correctly, could log on to a computer only once and subsequently access many others. A similar scheme was independently conceived and implemented on the CAP computer at Cambridge University by Johnson whereby users could elect to be able to be authorized by the network user identification system described in this thesis instead of by its own password mechanism [Johnson80b].

As has often been pointed out, all internal authentication and security measures are of little use if the procedure used for external authentication of a user is fallible. The popular method of authenticating people by password is, in general, rather inadequate. Popek and Kline propose that far more secure methods are necessary such as a mechanism based upon fingerprints [Popek79]. Amongst the problems encountered have been the security of the password file and the difficulty that people seem to have inventing passwords with sufficient randomness. A short password can easily be guessed and a long one is likely to be written down, compromising the security that it would have had. In addition there is a tendency for passwords to be fairly predictable. Morris and Thompson give a good account of the many ways in which a password system can fail in [Morris79]. R. M. Needham is first credited with the idea of enciphering a passwords file

and matching similarly enciphered passwords against it in [Wilkes68]. Such an enciphering algorithm must be non-invertible. Purdy [Purdy74] gives a description of the features desirable in such a function and proposes one based on polynomials over a prime modulus. [Weiss74] proposes a function which is simply too difficult to analyse, let alone invert. The ADEPT-50 system [Weissman69] allows up to 64 twelve character passwords to be kept for each user - they are used serially.

5.4 Protected Object References

Representation systems to facilitate authentication between a network and its component computing elements have received little explicit attention. However, some of the topic has been covered under different names.

The best known representation systems use encryption. Because encryption is all but indispensable when wishing to provide privacy over vulnerable communication lines it seems to many to be the best mechanism available for authentication (this is discussed in "Authenticated Communication"). In general these mechanisms involve the encryption and subsequent transmission of some data which the recipient is either implicitly or explicitly told (for example, the time of day or the name of the sender). The receiver then decrypts the message and compares it with the original data. Identity implies that the sender knew the key necessary for its encryption and deductions are made from there. In the private-key authentication protocols given in [Needham78], for example, the data is a simple function of some known information and the key is one obtained in secret only by the participants in a communication. As explained by Diffie and Hellman in [Diffie76], such encrypted items of data can be used as "digital signatures". A protocol for the use of such a signature is again given in [Needham78] in which the encrypted data includes the signatries name, and the key is uniquely known by a central authentication server which will, at any time, verify that the signature is merely the encrypted name of the signatory. A similar mechanism for creating unforgeable and safe names was proposed for use in network file servers in [Needham79]. Details of the use of both public key and private key encryption systems in a network are given by Popek and Kline in [Popek79].

One possible advantage of Diffie and Hellman's digital signature scheme is that the representations produced are permanent and not revocable. They can, therefore, be saved and used to prove identity at a much later date. [Saltzer78] however, shows that permanency brings its own problems. The unique, secret information used to generate signatures is not liable to be compromised over short periods of time, but since it is required to be secret indefinitely if name forgery is to be avoided, its eventual discovery is inevitable. When and if this information is lost it is essential that all further signatures using that secret information should be revoked. A central authority holding a list of secret keys and their last date of validity for each signatory could be used for this. Signatures in this scheme would have to include an authentic value for the time at which they were used so that uses of the stolen key can be distinguished from uses of a valid one. Popek and Kline note some other solutions to this problem in [Popek79] in which **notaries** are used to record the use of non-revoked signatures so that they may be verified at a later date. Whatever the mechanism used, it is obvious that any "secret" information involved in the formulation of an object representation must be revocable.

Dion used secret 64 bit numbers to represent the files in his file server [Dion81]. Two types of number were used, the names of which greatly influenced the nomenclature used in this dissertation. The first was used as the permanent (but secret) name of a file, using which the file could always be accessed for as long as it existed. This was called a PUID (for Permanent Unique IDentifier). The second became transiently available whilst a file was open and, in consequence, was called a TUID (the "T" is for Temporary). The TUID required frequent use in order to prevent the closure of the file to which it referred and the TUIDs subsequent disappearance.

Donnelly and Fletcher examine four methods of producing network "capabilities" and the problems associated with each in [Donnelly80] . The first they consider is "password protected" capabilities in which the capability for a resource is simply a password invented at random by the resource holder to refer to it. The password is then passed around the network in the same way as a capability. The second method involves access lists in which reliably known addresses are used to identify subjects and, in conjunction with the list, authorize access. A third method relieves the resource maintainer of the tedium of keeping an access list by encrypting the resource's name together with the address of the bona fide client and using the result as a capability. These last two methods would need messages to the resource holder in order to pass the capability to a new

address but this necessity is avoided in the fourth solution. This employs public key encryption (with commutative encryption and decryption functions). In this last suggestion a password protected capability (the encrypted name of the resource) undergoes a series of encryptions and decryptions in such a way that is usable only by its current holder at any given time. Although revocation of such capabilities was not discussed, the relevant explicit facilities could be added to their first two solutions. There is no suggestion that capability passing or revocation could be centralized to provide a unified network service.

A second paper from the Lawrence Livermore Laboratory [Nesset81] builds upon Fletcher and Donnelly's work to present a method of securely passing capabilities which he calls "protected identifiers" using private rather than public encryption (which, as technology currently stands, can be implemented far more efficiently). Resources are referred to by a name, **ID**, and represented by a protected identifier **Pij(ID)** which can only be used by **j** at service **i**. In order to pass the protected identifier **j** can use the function **Tijk** which transforms **Pij(ID)** into **Pik(ID)**. These functions are discussed and devised using private key encryption. Such capabilities cannot be usefully stolen as long as the network provides a reliable method of determining the identity of a sender (otherwise it might be possible for **k** to steal **Pij(ID)** and use it pretending to be **j**). Unforgeability is achieved by using **Pij(ID+)** rather than **Pij(ID)**, where **ID+** contains two parts - the name of the resource that it is to represent, and an unguessable set of bits to which it is related by some function that only the originator can perform.

The work described in this thesis is reported in [Girling82] and several references to local documentation can be found in [Girling81] which describes the use of the mechanism developed.

Article 6

IMPLEMENTATION OF AN OBJECT REPRESENTATION SYSTEM

This article delineates the implementation of the design outlined in "Design of an Object Representation System". The implementation of each of the major items of a system of representation is detailed in the sequence:

- a naming system
- unique, unforgeable tokens
- a relation between the unforgeable tokens and propositions about the names of the objects so represented

This system is briefly explained in [Girling82].

6.1 A Naming System

In the article "Design of an Object Representation System" it was shown that a naming system could be supported by an arbitrary number of name generating servers* providing names with the following characteristics:

- They have many bits so that the same name never need be generated twice during the system's lifetime.
- They have a field reserved for identifying the generating server.
- They are generated in order from a well ordered set.

It was decided to use a 64-bit number (a convenient size) to represent a name, the least significant 48 bits being allocated by the generating server. Of the remaining 16 bits only the least significant 8 are used to identify the name generating server. The most significant 8 bits are always ones - providing a small amount of static information which can be checked so that these names can be distinguished from tokens (the design of which is to follow).

---

* This is in keeping with the general aim of "fail soft characteristics" given in "Base Network Structure".

Thus for the particular implementation on the Ring the following restraints have already been imposed:

- There will be no more than 256 different name producing servers on the network (during the network's life time).
- Each name producing server may produce no more than 2^{48} (about 281 billion) different names.

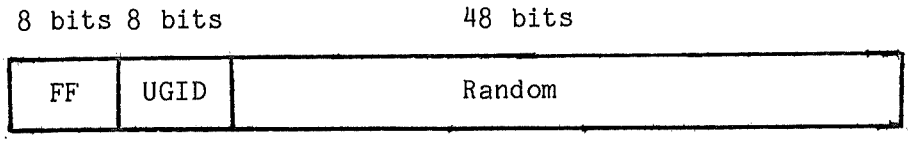
Despite the design for names recommended in the previous article it was decided, on this experimental system, not to generate names from a well ordered set but to pick them randomly from a series of 128-bit random numbers. (The series actually chosen is not quite pseudo-random, and is not, in fact, well ordered). The reasons for this are as follows:

- If the first name is a fixed value and each successive name is the next in a well known ordering (such as increasing integers) it would be some time before a name which could not easily be compacted into just a few bits arose*. This may tempt users into misrepresenting names in their programs and, perhaps, avoid some difficulty inherent in the use of large names. (Such problems would obviously be of interest).
- Name generating servers need not keep state in reliable storage whereas, if a strict sequence is to be produced, the last name generated must always be known in order to generate the next. Should this last name be lost a safe way to continue operation would be to abandon the current name producing identifier and choose an unallocated one. Since there are only a small number of these identifiers it is desirable that they should be able to last longer than the mean time between failures of the name producing servers (that is, that they can be reinstated after a crash). Name generating servers would have to be able to find the last name generated when restarting. This problem is completely avoided by using a random series since it needs no "state" information to operate. (Naturally, this is not the only solution to the problem - but it may be the easiest).
- Practically speaking, a random number generator is necessary, in any case, for the generation of tokens - as will be seen in the next section.

---

* Experience shows that names are allocated only very slowly under normal circumstances. Recent rates have been much less than even 100 names per month. However, this figure would be much larger if applications using nonce identifiers were developed.

A name produced in this way is called a permanent unique identifier (abbreviated to **PUID**) and has the following structure:



FF = all ones
UGID = 8-bit identification of UID generator

A single server whose address is given by the name server when sent the string "NEWPUID", which, upon request, will send a user a PUID in the above format.

Note that an SSP interface can be provided here because the loss of name generated and not received is insignificant - there being an adequate supply of names not to be concerned about the fraction that are lost.

6.2 Unique unforgeable Tokens

Relatively few constraints have been put upon the design of these tokens in the previous article: indeed the only characteristics determined have been:

- They are to be generated at random.
- They must have many bits so that they cannot be easily guessed during their lifetime.

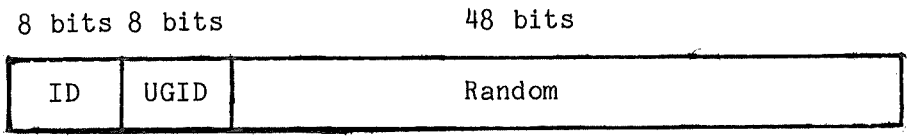
It was decided to use 64-bit numbers to represent these tokens, both for convenience and for the sake of uniformity (with PUIDs). Indeed, as described above, a PUID provides both of the characteristics necessary. Hence the NEWPUID server uses the code whose primary purpose is to generate new tokens. This functional dependency is purely for convenience and is not a logical necessity.

These tokens are generated only by servers maintaining relation tables. If there are several such tables (as is desirable in order to to fulfil the system aims outlined in the article on "Base Network Structure") it will be necessary to find which table contains any particular token (for the purpose of verification for example). Accordingly the most significant 8 bits of the PUID are changed from the ones present in a PUID to the identity of the particular table which should hold that token. Thus for a particular implementation the following restraints have been imposed:

- There will be no more than 254 representation tables at any given time. (Of the 256 values available from the 8-bit identification field in a token two are reserved: one (all ones) to denote a PUID; and all bits zero, which is used to denote a null token*).

- The chances of guessing a token so generated can be no better than 1 in 2^{56} (about 72 thousand billion). This assumes that there are a full complement of 256 NEWPUID servers chosen at random when a token is generated and that the representation table is already known. In the current implementation there is only one NEWPUID server so that the chances can be no better than 2^{48}. The chances of generating two identical tokens with overlapping periods of existence is somewhat lower than this and depends upon the first token's period of existence and the rate of token generation during that period. In the worst case a very rough calculation shows that this chance would drop as far as 1 in 2^{18} (about one in 260 thousand) - however, duplication is explicitly checked for, so this case never arises. This check can be performed because, by definition, a relation table server always holds all the tokens that it has generated that are valid.

A token produced in this way is called a temporary unique identifier (abbreviated to TUID) and has the following structure:



ID = identification of table in which TUID resides
UGID = 8-bit identification of UID generator

---

* The definition of a null token has been found to be a convenience in cases where an "impossible" name or token is desired.

TUIDs are generated only from representation tables' servers.

6.3 Active Object Table Service

The relation, forming the last part of an object representation system, is implemented in the form of an active object table (abbreviated to AOT). A service which maintains such a table is called an AOT server.

The active object table conceptually has tuples in the form:

$$\langle d, t, A, N, \triangle \rangle$$

as described in the notation given in "Design Of An Object Representation System". Both d and t are tokens, t is simply called the TUID. d is called the TPUID of the tuple because it can be thought of as the temporary (because it lasts only as long as the tuple) permanent name for the tuple. A is the tuple's authentity and N is its PUID. $\triangle$ is the tuple's time out value in seconds. The maximum value of $\triangle$ is 2^{24} (representing about six months) although a tuple can be maintained in existence for longer periods by refreshing the timeout.

6.3.1 Entries

As discussed in the previous article at least the following entries will be necessary:

- VERIFY

  To VERIFY that a representation is valid (that is, to prove t V p(A\N)).

- GETTUID

  To create a new representation given the tuple <d,t,A,N,$\triangle$> for a given $\triangle$ and N (given tokens x,y: x V i(y,AUTH\A) to prove that new representations may be created) returning d and t.

- REFRESH

  To change the timeout, $\triangle$ of a representation given by the tuple <d,t,A,N,$\triangle$> given d,t,A and N such that d V i(t,A\N).

In addition the following two entries are provided:

- IDENTIFY

  To VERIFY that a given representation is owned (that is, to prove d V i(t,A\N) for a given d,t,A and N).

- ENHANCE

  To create a new representation given by a tuple <d,t,A,N,△> using the same TUID as another representation and a given △ and N (if tokens exist x,y: x V i(y,AUTH\A)).

The above entries are basically of three types:

1) VERIFY and IDENTIFY are used to check things: VERIFY checks that a representation holds (t V p(A\N)) and IDENTIFY checks that a representation is owned (d V i(t,A\N)).

2) GETTUID and ENHANCE are used to create new tuples. In order to create a tuple it must first be proved that the representation of an authentity is owned. That is to say, x and y must be provided for which

$$x \; V \; i(y, \; AUTH\backslash A)$$

where A is the authentity for the new tuple. New tokens t and d are created using the GETTUID entry but the TUID t is specified in advance using the ENHANCE entry. The GETTUID entry can be used to create a representation under a first authentity, ENHANCE can then be used to make the same TUID a representation of other objects (or perhaps the same object under different authentities - a similar process to the signing of a contract by all those concerned: different authentities being equivalent to different signatures).

3) Using REFRESH a representation can be either deleted or maintained indefinitely. It is necessary to prove that the representation being REFRESHed is owned before this can be done. That is, tokens t and d must be provided for which

$$d \; V \; i(t, \; A\backslash N)$$

The timeout △ of the tuple <d,t,A,N,△> can then be changed. When the new timeout is zero the tuple is always deleted before a reply is made. This prevents there being a brief window during which the tuple might get revived.

Of these entries it is possibly ENHANCE whose use is the least obvious. Although it has several possible uses to the holder of an authentity representation, such as giving a simple TUID more than one PUID (alias) with which to identify itself, or giving a representation generated by another authority an alternative name under its own authority, its most important use is the generation of representations with more than one authentity . Such compound authentities arise naturally when an object is partially created by each of several creators: as each new part of the represented object is created an authentity for the part-creating authority is added to the existing TUID using ENHANCE. For example, suppose that a car is to be represented which is created half by a factory with a representation for 'auth\factory1' and another with one for 'auth\factory2'. The first factory, after making its bit of the car, would use GETTUID to create a representation for 'factory1\car' the TUID of which would be sent to the second. This factory would make its part and then ENHANCE the TUID so that a representation for 'factory2\car' also exists. The resulting representation could well be called '(factory1,factory2)\car'. If such a compound authentity is in use it should be verified using VERIFY first with authentity 'factory1' and then with 'factory2' to check that both components are present - if one is missing the TUID cannot be said to be a representation for '(factory1,factory2)\car'.

To give an example illustrative of the operation's similarity to adding a "signature" consider the representation of a European parliamentary bill. Suppose that such a bill is not recognized unless each constituent country agrees with its content. The proposing country would create the bill as a document and a representation for it (using GETTUID). It would then pass the bill to each other country with the TUID. Other countries would inspect the bill and, if it was thought reasonable, they would "sign" the bill by adding their name to the list of authentities given to the TUID (using ENHANCE). The compound authentity consisting of the names of all countries would then be used to verify the TUID where ever it was used. (An interesting concomitant is that any country could withdraw its support for the bill at any time by revoking its representation.)

6.3.2 Interface design

In designing an SSP interface for VERIFY, IDENTIFY, GETTUID, ENHANCE and REFRESH the following considerations must be borne in mind:*

1) Each entry should be repeatable.

2) All the tuples relating to a given TUID should reside in the same AOT (since the AOT's identity is built into each TUID).

These considerations do not affect either VERIFY or IDENTIFY since they can be repeated arbitrarily often without harm. Refresh is also repeatable since when a reply is eventually received the relevant tuple's timeout will have been set to the requested value. However, the requester must be prepared to get a return code indicating that the tuple does not exist if a new timeout of (or near) zero seconds is requested.

GETTUID and ENHANCE are not, strictly speaking, repeatable since repeated correct applications will produce spurious tuples in the AOT. However, this is (practically speaking) completely indiscernible to the user since the TPUID and the TUID are lost. In both cases the eventual reply will be correct and unaffected by any previous replies which have been lost. After the initial timeout value specified in the request spurious tuples will time out (since the TPUID necessary to maintain them will have been lost). The major cause for concern is that too many spurious tuples will be generated causing the table to fill. It is necessary to ensure that the mean time between lost replies is greater than the timeout value of each set of tuples with the same timeout. Since the mean time between lost replies is quite long (due in part to the Ring's low error rate) SSPs are quite reasonable as long as very long initial timeouts are not often requested. To this end the initial timeout that can be specified with ENHANCE and GETTUID is limited to 2^{16} seconds (about 18 hours) and must be set using REFRESH if higher values are necessary. Note that in the case of ENHANCE the creator of a new representation cannot guarantee to make

$$t \; V \; p(A \backslash N)$$

cease to hold after deleting the representation that he has generated if it is deleted before its initial timeout period. This is because the relation may be holding through a spurious tuple that was generated by the ENHANCE attempt.

---

* See the article called "Base Network Structure" about SSP.

In order to ensure that TUIDs, once generated, exist only in one AOT, ENHANCE checks that the TUID provided for the creation of a new representation already exists in the AOT.

In order to ensure the uniqueness of TUIDs, new TUIDs (generated by GETTUID) are always checked against those already in the AOT.

The actual interface as implemented is as follows:

entry	arguments	results
GETTUID	N, y, x, A, △	t, d
VERIFY	t, N, A	<none>
IDENTIFY	t, d, N, A	△
REFRESH	t, d, N, A, △	<none>
ENHANCE	t, N, y, x, A, △	d

where d V i(t, A\N)
and x V i(y, AUTH\A)

6.3.3 Implementation details

The address of the only AOT server in the Ring is given by the name server in response to the string "AOT-1". The "1" refers to the fact that all TUIDs from this service are marked with the value 1 in the AOT identification field. The number in the string is notionally decimal with no leading zeros although this is not well demonstrated in this example. The service is actually provided in a Z80 microprocessor using 32K bytes of memory.

The software was designed to minimize memory use and to make the VERIFY entry as fast as possible (since this should be the main manifestation of the server).

In order to reduce the memory requirement PUIDs and authentities (which have the same format as PUIDs) were kept in a table, each with its own use count, so that multiple copies did not have to be kept. There is a similar (though, naturally, not so effective) system for TUIDs and TPUIDs (which have the same format as TUIDs).

In order to reduce the time that the VERIFY (and IDENTIFY) operations take a hash table is used - the key being a function of the PUID, authentity and TUID of the tuple (the only information guaranteed to be available).

The result is that the VERIFY entry is faster than the entries involving the creation of new tuples. In many operating systems this efficiency cannot be well appreciated, since most of the time taken to perform a VERIFY will be taken in the host's software.

The NEWPUID server (which is provided by the same Z80) uses this random number generator for the random 48 bits of a PUID. The generator is based upon a 128-bit pseudo random number generator: a 48-bit field being selected when creating a PUID. A new tuple, however, requires both a new TUID and a new TPUID. Since it must be impossible to deduce the one from the other they must be independent (so that simply running the random number generator twice would be useless). 96 bits are used in the simultaneous generation of a TUID and a TPUID for a new tuple (when GETTUID is used) - the extra 32 bits in the generator being provided to make it difficult for new TUID & TPUID recipients to guess the value of the random number generator's seed and therefore be able to deduce the succeeding numbers. (If all the bits of the seed were in the TUID and TPUID the recipient of these tokens would be able to deduce the succeeding TUIDs and TPUIDs if he knew the algorithm used for the pseudo random number generation). Since the random number generator is a simple multiplicative type, care is taken to select the most, rather than the least, significant bits from the seed when taking a sample. Otherwise, no matter how large the seed, the next sample would always be deducible from the current one. To increase a user's difficulty in this respect the seed is incremented "every now and again" (once a second under normal circumstances). This combined with not knowing how many times the random number generator may have been used between calls is judged to be sufficient protection against prediction.

Since the lifetime of a tuple (up to six months or so) can be very much longer than the mean time between failures of the Z80 it is necessary to "back up" the AOT to stable storage. The information constituting the AOT is therefore periodically written to the Ring's file server. When the service is rebooted the last AOT saved is read back. This places a rather undesirable reliance upon the file server since if it is not functioning when the last AOT is to be read the service must be suspended until the relevant file can be read. The file used is secret (being a file server unique identifier) and known only to the service*. This method has maintained the AOT very well, there currently being a tuple which has

---

* The unique identifier for this file is currently built into the service's source. See "Service Representation" for a method of avoiding this.

remained verifiable for over two years.

Timeouts are uniformly decremented every second. Under normal circumstances the timing is supplied by a server on the Ring whose address can be obtained from the name server when presented the string "CLOCK". In fact, this server uses a radio timing signal broadcast from Rugby where an atomic clock is used. In this case such precision is not strictly necessary. Under normal circumstances it provides a signal every second including the time and date. If this server fails a less accurate internal software time is used until the clock server becomes available again [Johnson80a].

There is currently sufficient space for 500 tuples although this could easily be extended since not all of the memory available on the Z80 has been used.

6.4 Conclusion

The object naming system described in the article called "Object Representation" and designed in the chapter "Design of an Object Representation System" is implemented using:

- A naming system made from PUIDs providing both names of objects and the names of authorities (that is, authentities).
- Unique unforgeable tokens made from TUIDs used either to prove the validity of a representation (as a TUID) or to prove the ownership of a representation (as a TPUID).
- A relation made from AOT servers which provide the following entries:

VERIFY	-	to prove t V p(A\N)
IDENTIFY	-	to prove d V i(t, A\N)
GETTUID	-	to create a new tuple
ENHANCE	-	to create a new tuple with an already existing TUID
REFRESH	-	to change a tuple's timeout

The practical use of such a system is described in the following articles.

Article 7

THE SOURCE OF ALL POWER

This article investigates the structure of authority and establishes the desirability of a single Source Of All Power (abbreviated to SOAP). A service which utilizes the power available at this source in a well controlled manner is also briefly described.

7.1 Authority in Representations

Trusting a representation for A\N from the object representation system described in the article "Implementation of an Object Representation System" consists of the following checks*:

(a) that the expected representation is verifiable under the authentity A and name N at the AOT indicated

(b) that you trust the representation of objects created by A

(c) that you accept A\N as trustworthy

(d) that you trust the relation supplied by the appropriate AOT

(a), (b), and (c) can be taken together as a check that the representation for A\N is trusted. (d) could be checked using the techniques of static service authentication (assuming that AOTs are static) described in "Service Representation". (b) boils down to trusting that the possessor of a representation for auth\A does what is expected of auth\A and that the AOT properly verifies its representation before creating A\N (in the way given in (a), (b), (c) and (d) above). That is, it is necessary to trust auth\A before A\N can be trusted.

A hierarchy of authority is being developed here. The name auth\A is the name of a more "powerful" entity than A\N because auth\A could actually create a representation of A\N (as well as a large number of other representations of the same general form). The possessor of a

---

* For example, these could be the checks that an operating system performs on a representation for A\N before it allows A\N to log on.

representation for auth\A can, therefore, always do at least as much (by virtue of a representation) as A\N can. In effect, any power that A\N has, has been delegated to it from auth\A.

Reapplication of the same argument reveals that in order to trust auth\A one must first trust auth\auth and it is this that has delegated a portion of its authority to auth\A. In this implementation this is where the list of names to trust comes to a dead end because (by the same argument) in order to trust auth\auth it is only necessary to trust auth\auth itself. The possession of a representation for this name enables any representation to be created; it is therefore the most powerful name in the representation system and is distinguished by the term "SOAP name"*.

Supreme authority rests, therefore, with the SOAP name auth\auth. If a network is started up with only one trusted service, which possesses a representation for auth\auth, the authority structure of the rest of the network can be recreated from scratch simply by having auth\auth delegate its authority appropriately.

7.2 Authority in Reality

Although the names of things to trust seem to have come to an end with auth\auth, this is only so because higher levels of authority exist outside the network's representation system and therefore do not necessarily use names within it. It has been established above (implicitly) that it is the ability to create the representation of another implies authority over it#. To find a name with greater authority than another, it is only necessary to find a name that could be used to create a representation for it.

---

* Note that every name to be trusted, (auth\auth and auth\A in these two part names) can be deduced from the name A\N. This kind of name (A\N) can be thought of as an ordered list of objects which must be trusted before the name as a whole can be. This concept is not very clear where a name can have at most two parts (for example, one\two) but is made clearer when it is generalized in "Variants in AOT Design".

\# If you can create a representation for another object that representation can be used to pretend to be that object. It is therefore possible to do anything that anything you can create can do.

The representation system supports itself from within, since the name auth\auth can create itself - so once that name is represented, a representation for any name can always exist without the need for external authority. However, if it is supposed that there is ever a time when no representation for the SOAP name exists, and that it is then created, then the creator will, in effect, have greater authority than the SOAP name. In a real representation system allowance must be made for both the initial "bootstrapping" of the representation system (which, in theory, might only be once in the history of the representation system) and for subsequent loss of authority (literally) through accident, error, or theft; so there must be some way for a higher-than-SOAP-name authority to start things up again. The least that is necessary is that something must be able to create a representation for auth\auth and then send it to an initiating service - which could delegate the authority as appropriate. It is obviously best to have as few things with this ability as possible since each poses a risk to the integrity of the entire representation system: one would suffice (preferably incapacitated for most of the time). SOAP is the name of the thing with this ability.

SOAP can be thought of as a gateway between internal authority as carried by an internal representation system and external authority (such as line management or a military hierarchy). Since the security of the network is supported by object representations and the responsibility for each representation resides with SOAP, the security of the entire network could be enforced by protecting SOAP adequately. For example, if SOAP were a particular place on the network, that place could be enclosed in concrete and iron bars and surrounded by security personnel and so forth in order to ensure that the hierarchy of authority in the outside world is correctly administered.

On the Ring (described in the article "Implementation Environment") a particular station was chosen for SOAP. The station number of a sender is unforgeable and can be used to verify whether or not a request has emanated from SOAP. The AOT service described in "Implementation of an Object Representation System" will produce UID sets without checking an authentity representation if the request comes from SOAP.

7.3 Autonomy versus Heteronomy

In "Base Network Structure" autonomy was listed as one of the goals of a distributed operating system. That is, services should be at liberty to be as independent of each other as they please. It is true that services can choose either to use or not to use the representation system, and that if they do choose to use it they can choose which authentities to trust and which not. It has not been mentioned however to what extent more than one SOAP could be recognized.

There is no reason why the mechanism for the representation system could not be completely duplicated with one set of services having a SOAP name auth1\auth1 and another with auth2\auth2, for example. In the absence of any interaction between the two systems there is no reason why this approach should not be completely successful. As soon as there is some interaction between the auth1 authority domain and the auth2 authority domain however the meaning to one system of representations belonging to another becomes a problem. Given a representation for an object A\N it is necessary to know to which system it pertains in order to know which other names to trust (as discussed above: either auth1\A and auth1\auth1 or auth2\A and auth2\auth2) and, more practically to know to which AOT to go in order to verify it.

The crux of the problem is that two (or, in general, more) separate representation systems are being used. These systems could just as easily be completely different and the same problems would occur. The problem is that of assigning meaning in one authority domain to a representation from another. There are two possible ways that a particular authority domain can deal with it.

The first is to become simultaneously a member of both (or all) domains verifying each representation in the relevant authority domain. Note that it is always necessary to be able to identify the particular authority domain being used - whether it is explicitly given, or available only through context. In effect a representation's authority domain identity becomes part of the object's name in much the same way as an authentity. If the different representations all take the normal form the effective result is that a system using three part names is actually used*:

---

* This kind of variation in naming structure is discussed in "Variants in AOT Design".

<domain>\<authentity>\<name>

If different forms of representation are used this solution becomes rather unwieldy, since more than one way of verifying and handling representations will have to be supported everywhere, as will the recognition and comprehension of the different naming structures used in each domain of authority.

The second way is to convert external representation to internal ones using one of the methods of authentication proposed in the article on authentication. This has the disadvantage that any communication with another authority domain will have to be via the appropriate type of authenticator. Communications would need a "representation primitive" similar to the "address primitive" defined in [Yellowbook80] that representation gateways could perform the appropriate transformation on. Also to a greater or lesser extent the onus for deciding which representations are trustworthy is passed from the representation's ultimate recipient to the authenticator (which has both advantages and disadvantages). In addition, the authenticator has to be quite complex if it is to ensure that the two representations (internal and external) for any given object that it authenticates stay in existence for the same period of time.

In either case the costs and inconvenience can be large. Some degree of heteronomy will be necessary between the various authority domains, even if it is just to agree upon different names for themselves. Where possible it would seem preferable to adopt just one global representation system and to implement autonomy by dividing up the authentity name space among the different authority areas.

7.4 The SOAP Service

As discussed above it is SOAP's responsibility to initiate the authority structure of the network. Although this could be done simply by "passing the buck" to another service by sending it a representation for the SOAP name, in practice the distribution problem would still remain - but at this other service.

During the operation of the network objects and services will come and go - each requiring different authentity representations. It would clearly be rather inconvenient if external authority (such as that of an operator) were sought before granting a service (say) with the authentity representation it needs for its operation. Some automatic means of providing authentity representations to particular, well identified, objects is necessary. For this reason there is a service on the Ring which resides at the SOAP station which manages SOAP's responsibilities. It holds a list of objects (A\N) and the authentities that they are allowed. After checking that a particular object is allowed a given authentity representation it creates one for that client and passes it back. The interface to this service is precisely the same as that to the privilege manager (see "Privilege Management").

Using this mechanism a service, newly booted with its own representation*, can obtain the authentity representations that it needs by presenting its identifying representation to the SOAP service and making the appropriate request.

---

* The article "Service Representation" describes how this is done.

Article 8

EXAMPLE SERVERS

Some examples of the use of object representation are given in this article. In particular an accounting server, a server to provide environments necessary for the construction of new services and a server to provide homes for users on a fully distributed system are hypothesized.

8.1 An Accounting Server

Accounting traditionally becomes a problem when previously centralized functions become distributed. One of the reasons for this is simply the lack of a reliable form of representation for the objects being accounted against. This problem has already been overcome by the use of an object representation system. Any server can insist on a valid representation of its user and keep accounts against it internally.

When each server keeps its own accounts, information about the "cost" incurred by a particular user* is distributed around the network, so that its current deficit cannot easily be found (a scheme for revoking a user's resources when it exceeds its credit limit, for example, would be difficult to implement).

This problem can be solved using an accounts service which keeps different resources' accounts centrally. Such a service would possess a representation for 'auth\account', say, which it would use to produce representations for particular accounts (for instance, a "pages of printout" account (account\printout), "CPU usage" account (account\usage), and so on). Representations for the relevant accounts are allowed only to the services entitled to use them. For example, a printer server would be allowed the representation for account\printout and would then be able to charge its clients on that account.

---

* In this context "user" does not refer exclusively to people - it may include services or other objects on whose behalf the server is being used. The word "client" is often used instead of "user" to make this distinction clearer.

Typically a service, once started, would approach the accounts server with its own identification and request the representation of the accounts relevant to its operation. These would be generated by the accounts server and returned to the service, which would then be responsible for their maintenance until it died. Each time the service performed work on behalf of some principal* it would send the name of the principal, the amount to be charged and the representation of the account on which the charge is to be made to the accounting server which would decrement the relevant entry accordingly (note that this is not an idempotent operation).

The service might take the precaution of finding the principal's current credit before undertaking any work. This would be done by sending the account representation and the principal's name (probably as a PUID) to the accounting server. Additionally the same result might be available by sending a representation for the principal and the name of the account.

8.2 An Environment Server

There are several different ways in which a service can obtain data with which to work, each corresponding to different times at which this data was bound to it. The most implicit binding occurs when the service has access to data which is bound into its code when that code was created, and the most explicit binding occurs when the service is dynamically sent data in a request at some entry. Intermediate between these two extremes is data that is bound into the service when it (as opposed to its code) is created. A representation for the service itself is an example of the latter (see "Service Representation" for an explanation of these representations and why this binding can take place no later).

The environment in which a service runs is a combination of each of these kinds of data and is referred to as the service's local information (defined in "Base Network Structure"). In "Service Representation" a distinction is drawn between a service's period of activity and its period of existence - inactive services being allowed to exist. The distinction between an inactive and a non-existent service is simply that, although neither resides at any address on the network, an inactive service still has access to the environment in which it has been working. This distinction is similar to

---

* The principal of a service entry is the thing on whose behalf it was instigated. See "Authenticated Communication" for further discussion.

that between a suspended process and a job in a conventional operating system.

Clearly when a service is re-created it must be able to reinstate its previous environment - to which it must have access. This environment may actually be arbitrarily large but, using methods of indirection, the information needed to obtain a handle on it can always be made quite small (the name of a file in which the relevant information is kept, for example). Therefore, in order for a service to obtain its environment, it is at least necessary for it to have unique access to some (small) datum which it can set before expiring and can read after re-creation.

An environment server is a service which provides exactly this function. It acts as a small filing system, accessed by service name for services (principally, but perhaps also for other objects) to obtain a handle on their former environment.

It is used in the following fashion: when a server is preparing to cease activity, it saves its environment in some server-specific way and obtains some sort of handle for it (a file server file name would be typical on the Ring). It then sends this handle, along with its representation, to the environment server which "remembers" it against the service's name (as a PUID). When the service subsequently begins activity on re-creation it sends its identification to the environment server with a request for the return of its environment handle. The environment server checks its identification and, if it is valid, replies with the requested information which then enables the service to restore its former state.

This information could range from a small block of data for a small server to the entire state of an operating system (including, for example, its filing system and a memory dump). A service which uniquely remembers handles for large amounts of information from incapacitated services must possess the following properties:

- each handle should be well protected so the security of services cannot be compromised;
- no handle should be lost so that it cannot be recovered (the loss of a filing system would be disastrous, for example).

In order to support the second of these requirements an environment server should provide very secure storage (but not necessarily very much of it).

8.3 A Person Server

On a hypothetical network in which all resources are obtained dynamically the question of where to represent an active user has no obvious answer.

First, consider what representing an active user entails. A user, having logged on, will possess a representation for himself. It would be very inconvenient for him to remember this representation himself - so it should be remembered on his behalf somewhere on the network. When activities requiring his representation are undertaken on his behalf the initiative to send it must be taken by the user since there will be no other suitable authentication. Hence, there must be some way for a user to instruct his representation holder to send it somewhere. In addition to the user's representation the same place must have access to any services, privileges or other resources the user has collected - this is simply because the user may be represented at this place only (so that the proof of ownership necessary in any access can be established only there). Unless the user obtains and uses resources capable of storing the representations of the other resources he collects they will have to be kept with his own.

Taking a typical model in which a user finds a terminal connected to a terminal concentrator, logs on, requests connection to a machine allocator, obtains a processor of his own, connects his terminal to it and finally, having completed his work, releases his machine and terminal, there are several candidate places to position the user's representation.

First there is the terminal concentrator. Not only does it seem inappropriate to store the user's authentication (and possibly other representations) in a service which deals in terminal traffic, but this solution would make it difficult for a user, who has logged on only once, to possess more than one terminal (which he may legitimately require) especially from two different terminal concentrators.

Second the processor obtained by the user could be considered. This is certainly very attractive since the user is liable to have most control over the representations that he owns here - being able to write his own programs to manipulate them and so on. Unfortunately, the duration of this processor's allocation to the client is not guaranteed to encompass that of the object representations needing storing. In particular, the user may be active for longer than the processor.

Third the resource allocator, from which a processor is obtained, might be suitable. This server already needs to know the user's identity (to give to services that it has created on the his behalf) and have access to his

resources (in order to help in the construction of new services and so on). It is also capable of taking the initiative via commands given by the user, since it is used via a terminal connection. However, as with the terminal concentrator, there are problems if more than one resource allocator is used, and the function does not seem very relevant to an allocator of resources whose clients are not exclusively people.

The conclusion is that a separate place is needed to fulfil this function. The following properties would be desirable in such a place:

- It should have access to all the user's object representations (including that of the user himself).
- It should be accessible through a secure terminal connection (so that the user can give it commands directly).
- It should authenticate the user on initial connection.
- It should provide a set of commands allowing the user to cause the representations of objects that he owns to be available to selected services.

To the rest of the network such a service provides a source of work requested by its owner. For this reason it is called a "person server". Two parts of it can easily be distinguished: one that gives a command interface to the user's terminal, and another in which representations of objects can be stored. These two functions could be separated if there existed a service which provided "representation holders" in which other services could store the representations of objects that they owned. Such a service is called a fridge server*: the objects in which representations are kept being called a "fridge" (see below).

A person server would have the following attributes:

- On connection it would log the user on using the user authentication server.
- It would obtain a fridge from the fridge server and store the representation of the user in it.

---

* The article "UID sets on TRIPOS" details the use and construction of a fridge on TRIPOS.

- It would provide commands to manipulate the contents of the user's fridge such as listing it or using individual UID sets as the arguments to various AOT service entries.
- It would provide a small number of commands for sending representations to services using a small number of protocols - accessing the fridge in a similar way to a little filing system.
- It would provide the same kind of command to send the representation of the fridge elsewhere.
- The person server would cease to exist when the representation of its fridge was revoked - not when its terminal connection was broken.

In this way a user can either do simple things with the objects he owns at the person server or he can pass the entire fridge to some other, more able, service.

8.4 A Fridge Server

As described above, a fridge server is a service which serves fridges, fridges being containers to which new representations can be added, and from which representations can be copied and which, where possible, ensures the continued existence of each representation that it contains. A fridge is to a representation system what a directory is to a filing system, with the difference that a fridge may need active interest to be taken in it in order to maintain it and its entries (that is, it is an active object, rather than a passive data structure). As such it would be useful in more applications that simply the one given above. It could be used as a convenient way to pass multiple representations from one service to another, for example.

When a new fridge is created, the fridge server should create a representation for it which is passed back to its user. The fridge's representation thus created will exist until it is deleted or times out. It could be used to insert new representations into the fridge, and remove or copy them from it. Whilst a representation resides in a fridge it will be periodically refreshed if its TPUID is available so that it does not time out (TPUIDs are explained in "Implementation of an Object Representation System").

Note that, if the representations within it are access capabilities, a fridge becomes an implementation of a "domain" as explained in [Lampson71].

Article 9

PRIVILEGE MANAGEMENT

This article describes the use of privilege representations. Privilege management is carried out by a service whose address is held in the nameserver under the name "PRIVMAN".

9.1 Privileges

Services on the Ring, unless they explicitly wish to restrict their service to a particular one or two users or services, characterize the set of their valid clients as those that possess a representation for a particular privilege. For example, if service A can accept valid requests only from service B it might choose to require a TUID for 'service\B'. However, if A can accept any one of a set of users, it might require a TUID for 'privilege\A-user'. A privilege is an abstract object whose most important quality is that only certain objects are allowed a representation for it. The names (authentity\PUID pairs) of these objects and the privileges they are allowed are held in a table which is internal to the Privilege Manager which has the capability to create privileges (that is, it owns a representation for 'auth\privilege').

Privileges can also be used by system programs in order to ascertain the status of a user (or other represented object) on a network.

The interface has been designed to be used in one of two ways:

(a) Using the service to check whether or not a particular object is allowed a given privilege.

(b) Using the service to grant a privilege that is allowed.

Systems with independent authorization systems can use (a) and those which use the Ring's system can use (b). The name of any object which is allowed a given privilege is referred to as a **virtue.** A representation for a virtue is necessary for (b), whereas only its name is needed in (a).

Note that (a) and (b) above are slightly different. The entry for (a) checks that a privilege would be allowed to a particular object if it were to request it. The entry for (b) actually returns a granted privilege which can, of itself, be used to prove that the privilege was allowed to that object. It is preferable that services, in general, use this latter logic to verify that a privilege is possessed. The reasons for this are two fold:

(1) It allows the owner of a privilege to "pass it on" simply by passing on its representation. Thus an intermediate service can be delegated to perform some action that the possession of the privilege allows (without being able to impersonate his caller).

(2) It allows an object (a person, for example) to vary its ability to do privileged operations by selectively claiming and deleting privilege representations. Thus, a user could claim a privilege for a short time, during which he performs a set of privileged operations and could then delete that privilege to ensure that, like the rest of the user population, he can not accidentally perform potentially "dangerous" privileged operations.

Having obtained some privilege, A, from one of the services, a user could then submit a request, presenting A as a virtue, and claim any privilege that he is allowed by virtue of the possession of A. For example, suppose the user has a TUID and TPUID for 'user\GSM' and only one privilege is available to 'user\GSM' (say it is for the privilege 'ringuser'). Having claimed this privilege the user can do anything that the 'ringuser' privilege allows, and although no more privileges may have been available to 'user\GSM' some may be available to 'privilege\ringuser'. Thus, by presenting privilege 'ringuser's TUID and TPUID, the privilege 'IBMuser', for example, may be obtained (whereupon there may be more privileges available when using 'IBMuser' as a virtue, and so forth).

9.2 Interface Design

The PRIVMAN service has an SSP interface* providing entries of two general kinds. The first category of entries can be used by anyone representing the main use of the service. The second category of entries can be used only by a restricted class of users, distinguished by the possession of certain privileges, and are used to maintain, correct and

---

* The article called "Implementation Environment" gave the implications that SSP has on an interface.

update the service.

9.2.1 The user interface

The privilege manager controls the generation of UID sets with the authentity 'privilege'. It controls to whom these UID sets are given by maintaining a list of object names (PUID, authentity pairs) and the allowances (privileges) that each object may request. Requests are made of the form "Please give me privilege <PUID> by virtue of my holding a valid UID set for this object (authentity\PUID): here is the object's TUID to prove that I can use it and its TPUID to prove that I am its owner: <TUID, TPUID>". The service then checks that possession of that object entitles ownership of the requested privilege and, if so, generates a UID set for the privilege and passes it back to the user.

There are two ways in which a new UID set can be created for the privilege; either with an existing TUID (using ENHANCE to generate the new privilege) or with a new one (using GETTUID). These two methods are supported by the entries BESTOW and GRANT respectively. Since, in general, only a TUID will be necessary there is an advantage in generating privileges using BESTOW: namely that all the privileges that a user (or other object) possesses will be verifiable from his TUID (which may be passed around anyway, in authenticated BSP for example - see the article on "Authenticated Communication"). This is also its main disadvantage since the more things represented by a single TUID the greater the scope for its misuse if it is lost. There is also an entry to check that a particular virtue (PUID, authentity) would allow a given privilege if presented.

9.2.2 The maintenance interface

There are also entries for editing the (virtue, privilege) pairs held by the Privilege Manager. In order to use these entries a special privilege is needed the TUID for which is referred to below as 'auth'. This privilege is called 'privpriv'. The first, NEWPRIV, enters a new pair into the table. It has a null effect if such an entry already exists - and, as such, is repeatable. The second, KILLPRIV, simply deletes a given (virtue, privilege) pair if it is found.

The information about which virtue is allowed which privilege is held constantly within the Z80 which provides these services. It is "backed-up" to a file server file whenever the information in it is changed. This file

is secret and inaccessible to anything but these services.

9.2.3 **Interface summary**

There are five entries provided by the services: one to check that a privilege is allowed to be granted without calling an AOT; another two to grant the requested privilege if the check succeeds (using either GETTUID or ENHANCE); and another two for either adding or deleting (virtue, privilege) pairs.

entry	arguments	results
ALLOW	N, A, P	<none>
BESTOW	t, d, N, A, P, △	d'
GRANT	t, d, N, A, P, △	t', d'
NEWPRIV	N, A, P, auth	<none>
KILLPRIV	N, A, P, auth	<none>

where	d V i(t , A\N)
and	d' V i(t', PRIVILEGE\P)
and	auth V p(PRIVILEGE\PRIVPRIV)
and	P = request privilege
and	A\N = name of the virtue required

Article 10

VARIANTS IN AOT DESIGN

This article describes some alternative designs for the Active Object Table used to implement object representations. Variations of the structure of names are discussed: both an historical one-level naming scheme and a multi level scheme are described. The meaning and advantages of a "dynamic" as opposed to "static" name structure are given. The general problems of unknown trust and representation confinement are discussed and an implementation of "pass once" representations is proposed.

10.1 One Level Naming

The implementation described in the article called "Implementation of an Object Representation System" supports a two-level naming system. That is in order to uniquely identify any object, two name parts are necessary - a PUID for the object's authentity and a PUID for the object within that authority's naming domain. Originally the Ring's AOT server supported one-level ("flat") naming in which the concept of authentity was absent or at most represented by a constant. New object representations could be created upon the provision of a representation of a particular object ('auth') and using it an object with any name could be generated.

Since there were no "sub-name" domains (that is, authentity controlled domains) in each of which the same object could be named differently, the naming scheme was necessarily global, with a fixed identity between every object and just one PUID. In the current two-level naming scheme there are as many sub-name domains as there are authentities (in each of which a single object could be given a different name). Since a single mapping between PUIDs and objects is desirable, the same object should be given the same PUID under however many authentities it is represented (in the two-level system). The one-level system had the advantage that such a convention was unnecessary since it was implicitly enforced - there being only one "authentity" (the null authentity).

The major problems with one-level naming stem from size of the an object producing service's domain of responsibility. If some service is to produce objects, it must be given a representation for 'auth'. It must therefore be a trusted service since it will be in a position to produce representations of any object (even those that it is not expected to produce) and could, at the worst, therefore freely give away representations of any object requested - potentially invalidating the entire representation system (since the recipient of a representation is unable to tell whether or not it is in possession of a genuine representation or a bogus one produced by the rogue service). Thus, the set of representation-producing services must be a tightly protected and well known collection (constituting the core of a distributed operating system perhaps). The loss of an 'auth' representation would be disastrous to the security of the whole system. An analogy between the possession of an 'auth' representation (or not) and being in either "supervisor" or "user" state in a conventional operating system is easily drawn.

The problem of having to trust services that create representations is not so acute in the current two-level naming system. The responsibility for ensuring the trustworthiness of such a service has passed from the donor of the authentity representation (with which representations are created) to the recipients of objects that have been made under a given authentity. As an example: if a service has an X authentity representation and is known to be untrustworthy, it is up to clients who receive objects with X authentity not to trust them on the basis that they are likely to be of dubious origin. Note that, since trust is a relative concept, producing objects under the X authentity is not completely useless* since there may well be services which do trust the X authentity (X itself for example). The loss of an X authentity representation is not detrimental to the entire system - only to its valid possessors; thus a "fire wall" is placed around each of the separate types of objects which is produced in a system. An analogy can be drawn here between the possessors of particular authentity representations with members of a single level of protected processes in (for example) a conventional capability-based operating system.

---

* If it were useless the one-level naming approach would have the advantage that such a wasted activity would be impossible in a secure system.

10.2 Multi Level Names

An obvious extension from the above would suggest the use of n-level naming in which n name parts are necessary for the complete identification of an object - the first n-1 of them comprising the object's authentity. Here n denotes a number which is fixed and does not vary with each name. The creating object itself must, therefore, also have an n part name, one or more parts of which are dropped to become an authentity for a new object - to give room for the name parts specific to the new object. Each authentity will lose the same parts of its name when passed on to a created object's name. These lost name parts must be fixed and common to all objects with the capability for creating representations, otherwise the whole name of a representation's creator will not be deducible from its authentity.

Assume an authentity is recognized by fixed values in one or more of its name parts. This clearly divides any name into two parts: the m name parts that comprise the authentity and the n-m that comprise the rest. This is equivalent to the two-level naming scheme with larger authentities and basic names (with a single name part possibly being replaced by several in each case). In other words, given that the size of name parts is already adequate, the two-level naming scheme is representative of all n-level naming schemes (for a fixed n greater than one).

Names that do not have a fixed length, however, are not equivalent to the two-level naming scheme. In an n-level scheme parts of a name are necessarily divided up into a new part and some part(s) of the constructing name - the rest of which must be assumed. Only objects with the assumed name parts (which identify an authentity) can create objects. Using **multi-part names** there is no need for a generated name to be of the same size as the generating name. The whole of the generating name could be included in the new name. Since no part of the generating name has to be assumed there need be no restrictions as to which objects can create names. A representation's authentity would be the full name of its creator. This type of multi-level name will be referred to as a multi-part name in the following subsections. An active object table could easily be implemented to support such names - holding a list of PUIDs in the authentity field of each entry and allowing any verifiable object to create another representation (with one more component to its name).

10.2.1 Flow of authority

The most important implication of the use of multi-part names is that there need be no restriction upon names that are allowed to create new representations. Since the full identity of the representation's creator becomes part of the new object's name, it is impossible for one object to create a new representation that looks as if it were actually created by something else.

Free name creation is a very natural way to pass authority down a hierarchy. Any object's domain of responsibility* can be carved up and each portion assigned to deputized objects to whom the authority to manage their portion is delegated. Such deputation can neatly and representatively be achieved by the creation of deputies whose names include that of the deputizing object as their authentity.

The flow of authority and hence of responsibility is not as easy as in a two-level naming scheme. The relevant mechanisms for two-part names (as described in earlier articles) are as follows:

1) The SOAP machine may delegate overall authority, or portions of it, to objects with particular names.

2) Holders of authentity representations can delegate their authority to objects with the given authentity.

3) Holders of other representations can effectively delegate their authority by obtaining an authentity representation from the SOAP server and using 2) or if there are a fixed (small) number of deputies each can be identified singly using privileges from the privilege manager.

4) The possessor of any representation can delegate its entire authority simply through communicating that representation.

Multi-part names, by effectively making 3) unnecessary because of 2) partially remove the need for the SOAP Server and the Privilege Manager.

---

* In a safe hierarchy responsibility must always balance authority. The fewer things something has authority over, the fewer it should be held responsible for. If something is responsible for more things than it has authority over or if it has authority over more things than it is responsible for, corrective action to reinstate this ballance is always possible. Such action is rarely desirable.

10.2.2 SOAP & Privman servers

Using multi-part names all representations can create new ones. It is no longer necessary to use the SOAP Server to obtain the ability to do this and so, to some extent, this server has become redundant. The standard use of the privilege manager is to allow certain objects to claim privileges which services check when sensitive entries are used. To use an analogy with the locksmithing trade, each service has just one lock for each of its sensitive doors and each valid client is given the key: the privilege manager is simply responsible for keeping each client's key ring. Strangely, using two-part names, a service cannot do the equivalent of keeping key rings of keys to its own doors it would have to be able to create privilege representations and, by default, it would not have an authentity representation with which to do this. However, using multi-part names, services would be able to use their own representations to generate privileges with*. Thus these two servers would no longer be strictly necessary, although their existence might well still be of use.

Both of these servers currently experience some degree of inelegance in updating their tables, since it is not possible for the service to which a particular privilege or authentity "belongs" either to delete or to create anyone's ability to claim it. These operations can only be executed by certain trusted objects when they obtain a particular privilege. This inflexibility is because the concept of a privilege or authentity "belonging" to certain objects (services, for example) is not made explicit. It could be made explicit, for example, by the inclusion of the "owning" object's name in the authentity of the privilege used to update the server's tables.

---

* Note that if a server is also going to use its own representation to create other representations for the resources that it may be serving, it might first create two new representations; one with which to create new resource names, and another to use when creating new privileges ('<service>\privilege' and '<service>\resource' for example).

10.2.3 Control of the Privilege Manager

The difficulty in controlling the Privilege Manager could be solved using multi-part names. Possible enhancements using the two-part name systems are considered first. If the Privilege Manager were to allow any service to create lists of objects which can request a particular privilege (rather than just the possessor of a privilege-mananger-updating-privilege) then the identity of the creator of such lists would have to be recorded so that the "ownership" of that privilege could be made explicit (and, therefore, enforced). The Privilege Manager would then have to ensure that:

(a) No other object could make entries for the same privilege - otherwise anyone would be able to create an entry which allowed themselves that privilege and thus obtain it.

(b) Only the recorded object could delete the entry. Otherwise anyone could delete all entries involving a particular privilege and then create one themselves to give them the privilege as in (a).

Even with these restrictions, at the "start of day" a rogue object may be the first to claim certain privilege names itself which by rights "belong" to another object - thus preventing the rightful owner from claiming the names and leaving them to be claimed as privileges by a set of clients dictated by the rogue object.

This is, in fact, similar to the problem that holders of 'auth' representations had in the flat (one-part) naming system to ensure that the sets, of names used for each different kind of object they produced were mutually exclusive: a problem neatly solved by the introduction of authentities (and the use of a two-level naming system). A similar solution could be used to split the names of privileges into sets each of which "belongs" to a particular client. The choices are threefold:

i) A special three-part name is used for privileges alone (<name of the object it belongs to>\<name of the privilege itself>, for example:

<authentity>\<PUID>\<privilege>

where <authentity>\<PUID> is the name of the owning object).

ii) A generally adopted three-part naming scheme is used, allowing only certain objects distinguished by their authentity (only names of the form

system\service\<service>

for example) to name their own privileges (with names of the form:

privilege\<service>\<privilege>

perhaps).

iii) A multi-part naming scheme is used in which each privilege incorporates the name of the object to which it "belongs" (for example:

privilege\<full name of privilege creator>\<privilege>

which is a name two parts longer than the creator's name).

The adoption of multi-part names would, therefore, solve the problem of updating the Privilege Manager (and similar managers which produce objects on another object's behalf, such as the SOAP Server).

10.2.4 The SOAP name

Using multi-part names, the name from which SOAP theoretically obtains its authority is different. In the flat naming scheme this name is 'auth' and in the two-level scheme it is 'auth\auth'. Possession of valid representations of these names enable any other object representation to be created. Using multi-part names each name can be broken down and analysed for trust on hierarchical basis. In

a\b\c\d

for example, d is the name of the object which is positively identified if the authentity 'a\b\c' is trusted. This is trusted only if it is known to be trustworthy and it, itself, is positively identified. Thus the trust that one has in 'a\b\c\d' relies on the trust that one has in 'a\b\c' which in turn relies upon trust in 'a\b' and so on. Eventually this recursion results in trust in 'a' relying upon trust in '<nothing>'. Clearly, it is necessary for all clients to trust the authentity '<nothing>' otherwise no name generated would be trusted.

Since the possessor of any name can create any other name lower in the hierarchy (that is, with more name parts but beginning with the same ones) then the holder of a valid representation for the name '<nothing>' can create any representation. It is the possession of a representation for this name, therefore, that is equivalent to the authority of SOAP.

10.2.5 The structure of multi-part names

Using the two-level mechanism, names of the following form can readily be produced

a\b

Also, using ENHANCE (see the article entitled "Implementation Of An Object Representation System") an object may, in effect, be given two (or more) authentities. That is, an object's representation could be verifiable under two authentities simultaneously. In the case of objects which are created by a combination of more than one object (for example, semi-fabricated by one and completed by another) it may be necessary to ensure that each object is represented as an authentity before the new object is considered valid. In such a case it would seem reasonable to include both authentities as part of the name. This means that the general form of a two-level name is

(a1,a2,a3, .. aN)\b

By proceeding to consider multi-part names (in which the above becomes the general format of a name with each of a1,a2,a3 .. aN and b potentially being replaced by multi-part names), the range and structure of names becomes quite complex. As an example taken from a more familiar naming situation consider a child, William, whose parents, Charles and Diana, had parents called Elizabeth and Philip, and Edward and Frances, respectively - and, for simplicity, suppose that William's grandparents were numbered amongst the original inhabitants of the earth. Then William's full name would be:

((Philip,Elizabeth)\Charles,(Edward,Frances)\Diana)\William

In general an object's name will include its entire "family tree"!

In a network of computers objects are unlikely to have very long histories and so this would not be as bad a failing as would appear from the above example. However, in this context, there is a greater problem. When naming human offspring, someone cannot be his or her own ancestor or descendant. Unfortunately this is not so in a computer network and names more suited to a "graph" structure are possible. For example, consider a network on which there are two "boot servers" each of which is able to create new services represented by names that the boot servers themselves have generated. Also suppose that, for the sake of robustness, each boot server is able to reboot the other in the event of a crash (or some other reason). If the services initially have names 'boot1' and 'boot2' they will

initially produce names of the form 'boot1\<service>' and 'boot2\<service>' respectively. However, as soon as one of them boots the other, one of these names will be changed: if boot2 crashes it will be re-created by boot1 with the name 'boot1\boot2' and it will start to generate names of the form 'boot1\boot2\<service>'. If boot2 subsequently reboots boot1 it will, therefore, be given the name 'boot1\boot2\boot1' and will, in effect, have become its own parent. Using [X] to denote repetition of "X" zero or more times separated by "\" each time, this means that the names denoted by:

[boot1\boot2]\boot1

all refer to exactly the same object (boot1).

To summarize the graph like nature of multi-part names: they could be constructed out of component parts in three main ways:

(1) By representation creation - in which a new name part is appended to the end of the creator's name. For example: a\b\c\d\e\f

(2) By representation enhancement - in which a new authentity is added to that of an existing representation. For example: (a,b,c,d,e)\f

(3) By arbitrary repetition of parts of the name - caused by the ability of some objects to re-create themselves or parts of themselves. For example: [a\b\c\d\e\f]

Names constructed just using (1) have a structure similar to a list of name parts, those using (2) in addition have a tree like structure in which there are several different names at each node representing different authentities. Names using all three methods have a general graph like structure since a cycle might be a repetition of any sub-pattern in the tree.

10.2.6 Precision versus generality

It is unlikely that the naming mechanism, itself, can automatically help in the above case of name repetition since it cannot know that 'boot1\boot2\boot1' and 'boot1' are actually the same object: it would certainly be wrong to assume so simply because the same name part appears twice. However, if a new representation's authentity were structured in such a way that cycles could be specified explicitly, this type of name could be dealt with. For example, if boot1's name were

[boot1\boot2]\boot1

rather than any particular instance denoted by this pattern it would

re-create boot2 as

[boot1\boot2]\boot1\boot2

which would then, eventually, rename boot1 as

[boot1\boot2]\boot1\boot2\boot1

and there is no reason why this object should not be able to produce objects under the authentity

[boot1\boot2]\boot1

which is equivalent (though not as precise). Note that the reverse, that a less precise name could create a name with a specific instance of its name as authentity, would not be allowed: for example, the object 'boot1\boot2' could create things with authentity '[boot1\boot2]' but the object '[boot1\boot2]' could not create objects with authentities 'boot1\boot2' or 'boot1\boot2\boot1\boot2' and so on. If this rule were not imposed the following might occur: 'a\b\a\b' creates objects under authentity '[a\b]', for example '[a\b]\x'; '[a\b]\x' could then, if this rule is broken, generate things under authentity 'a\b\x' (amongst others) - and has therefore been able to create names under a "higher" authentity ('a\b', that is) than that to which it was entitled.

The square brackets mechanism has allowed a degree of generality which can be used to cope quite accurately with the two boot servers case. However, the degree of generalization may have to be greater in some examples. The AOT could automatically allow an object with a name like

a\b1\ .. \bN\b1\ .. bN\c

(where a and c are arbitrary strings of name parts) to use the authentity

a\[b1\ .. \bN]\c

simply by checking that the first name is a specific instance of the second, generalized name. However, it is not obvious that the AOT could always give a yes or no answer to the question "can name A use authentity B" when both A and B could contain generalizations.

The crucial point here is that the AOT has to be able to compare two names (structured using some syntax) and be able to decide whether one is a generalization of another or at least whether they are the same or not. If the form of the names has become so general that no algorithm can be produced which can guarantee to determine the equality of any two names then

the naming system (with that syntax) is of rather dubious utility.

Even if comparisons can be made on names using the square bracket syntax there is no guarantee that a characteristic name can always be found for any given set of related names, without extending the syntax. Consider, for example, a simple extension of the example used above in which there are three boot servers instead of two. Each of the boot servers might, without using the arbitrary repetition construct, have names of an arbitrary length the last name part of which is 'boot1', 'boot2' or 'boot3' depending on which boot server the name is for and would have an authentity in which no two adjacent name parts are repeated (assuming that it is impossible for a boot server to boot itself).

To characterize such a name using only the square bracket notation is difficult. It would need more powerful pattern definition devices. The pattern needed is one in which identical consecutive name parts are disallowed. However, an over generalization can easily be made by using the authentity

[[boot1]\[boot2]\[boot3]]

which does not specifically exclude consecutive identical name parts. This authentity thus includes a group of expansions that never arise in practice. If it is used, this information (that consecutive names are never the same) will be lost to a recipient of a representation. In this case the loss of such information may be of no importance. However, the principle is demonstrated that the degree of verifiable information available in a name can the traded off against the complexity of the pattern matching syntax used to specify it.

Consideration of the kinds of name that would be produced if bootservers had a dual authentity (both its booter's and the name of the machine or interpreter on which it runs, for example), such as might be generated using ENHANCE, where the pattern in a name is only partially repeated suggest that an adequate syntax for all uses would have to be rather sophisticated. Suffice it to say that the job of an AOT in reconciling the actual name of an object, and the name that is to be used as the authentity for representations that it wishes to generate, is beginning to look rather difficult.

An alternative to the explicit representation of cycles within names is the use of SOAP to "flatten out" these cycles (itself a generalizing operation). This can be done by giving the first name including one cycle the ability to claim the original name. For example allow 'boot1\boot2\boot1' to claim the authentity 'boot1'. In this way cycles need never form. However information associated with previous authentities will also be lost. In effect the job of evaluating the trustworthiness of each boot server is delegated to SOAP when making an appropriate simplification. This solution has the disadvantage that each possible cycle must be predicted in advance and that the "proper" name of each object is unavailable for a full evaluation of its trustworthiness.

10.2.7 Conclusions about multi-part names

Multi-part names are a more "natural" way to name objects than two-level names. However, imposing no restriction upon who it is that can create new names can be viewed as a disadvantage. The control that the SOAP Server currently provides over which objects it is that can be trusted with the ability to create other objects is lost, and it can no longer be guaranteed that comparatively untrustworthy objects will not become entrusted with this capability. As already discussed, this does not affect the reliability of the names generated since the untrustworthy component will be manifest within the name. However, a certain amount of irresponsibility in the use of the AOT server could fill it with useless names or permanently account for its available bandwidth or table space denying the service to others*.

Multi-part names would enable much finer control of SOAP and PRIVMAN, but, on the other hand, would reduce their usefulness.

The variability of name size in a multi-part name is another obvious disadvantage. The basic unit of transmission on a network is likely to have some maximum size above which the transfer of information is bound to require more complex protocols. A multi-part name can always potentially require more room than that provided by this maximum size. Quantities with unknown sizes are universally more difficult to deal with than those with fixed sizes.

---

* This problem is less severe in the two-level name system than in the multi-part one because the possessor of any representation can create new representations in the latter, whereas, in the former, only holders of valid authentity representations can.

Multi-part names bring with them the possibility of graph-like naming structures along with the problems and disadvantages associated with them, although these can be largely avoided through careful use of the SOAP Server.

10.3 Dynamic Name Structure

There are two ways in which an authentity can be recorded in an AOT - giving the resulting representation different properties. The current implementation on the Ring simply records the name of the representation presented to authorize the creation of a new representation. The alternative is to record the entire representation as the authentity. This would enable the authentity representation to be checked each time the main representation is verified, the practical upshot of which is that all representations created by a given authentity representation would be deleted when the authentity representation was itself deleted. Such a representation structure is termed "dynamic" (the other option being "static").

Dynamic names avoid the possibly anachronistic effects of having a representation which refers to a non-existent authentity. In the time since the representation was created its creator may have changed its level of trust, which might possibly give the client a false view of the trustworthiness of the represented object. The property whereby dynamic representations disappear when their creator disappears might also be considered useful in the typical case in which the creator is responsible for the upkeep and maintenance of the objects to which the representations refer. In this case, representations should be deleted when their creator disappears.

However, this is not always desirable. To take an example from the human object domain it would be disastrous if someone lost his name as soon as either of his parents died (or any of his ancestors in the case of multi-level names). It is quite possible that a creating object may be expensive to run, existing, therefore, only for the duration of a creation - the upkeep and maintenance of the ensuing objects being given to some other object. A further consequence of the use of dynamic names is that, if all objects can trace their authority back to a single SOAP name representation, the safety of that representation is of paramount importance, since if it is deleted or is allowed to time out all the objects on the network will be

deleted. In any case, the action of dynamic names can always be emulated. The creator of an object is the first recipient of its TPUID, and and so can manually ensure that its representation's existence is terminated at the time (or a little before or after) at which it itself is deleted simply by using the TPUID in the normal way.

10.4 Pass Once Representations

If a representation is sent from A to B which then passes it on to C it can appear to C that the representation originally belonged to B, rather than to A. This use of representations could cause a breach of security. However, if a representation system were to use explicit mechanisms for

a) the passage of a representation from one object to another, and

b) checking that a particular representation is currently held by a particular object

then such deception can be made impossible.

10.4.1 Simple mechanism

A representation system could be built upon similar lines to that described in the foregoing articles. Each representation could be kept in a central table and object names could have the same general format. In order to provide an entry to the service which holds this central table (called OPRT for one pass representation table), so that it can verify that a particular representation is currently held by some object, it will be necessary for the OPRT to hold, the name of the object that currently owns it against each representation. Thus, a single entry in the OPRT will have at least the following fields:

< name of > < name of object > <name of owner of>
<representation> <being represented> < representation >

Transferring such a representation to another owner (that is, changing the contents of the third field) would be quite a different operation to that of checking that a particular object possesses a particular represented object (that is, checking that there is a OPRT entry which includes that object against the name of the represented object).

Note that, once a representation is passed to a new owner, it will no longer be possessed by the donor. The representation can, therefore, be passed from the donor only once (hence the name) and will exist at only one position on the network at any one time. This is a desirable property of a representation since it reflects an important attribute of the represented object (that is, its unity). It could be exploited for use as an "interlock" upon that object for example.

The price paid for using these representations is in the protocol necessary to manipulate them. In order to pass a representation from a donor to a receiver "pass by reference" rather than "pass by value" must be used. That is, the recipient is not passed the value of the representation explicitly, it is passed a reference to it in the OPRT and the OPRT is updated reflect the change of ownership. Thus, two transactions are always necessary: one to the recipient and one to the OPRT server. Also in communications in which a receiver is required to verify that the sender possesses a certain representation, it must ensure that the identity of the sender is correctly known, and this might entail the use of two-way authentication (see the article called "Authenticated Communication"). Otherwise the receiver might verify that someone other than the true sender possesses the representation and may allow the sender to operate under a false identity.

10.4.2 Monitors

Once a representation is given away, control over it is lost completely. This is desirable to the representation's current holder because it guarantees that others cannot duplicate, delete or artificially maintain it. Ideally, the functions just mentioned should be segregated from the ability to pass the representation on, so that the state of the object's representation can be made to reflect the actual state of that object by a completely independent service, without fear of interference. For this purpose the concept of a "monitor" is useful.

With each representation held in the OPRT an extra field giving the name of that representation's monitor is kept. Instructions to the OPRT server to change the timeout on a representation would only be valid if it can be shown that they originate from its monitor. Initially, the object that created the representation (given by its authentity) will be its monitor but, in the same way that the representation can be sent from one object to another, it would be possible for the current monitor to designate its

successor and pass the responsibility on.

Revocation is possible in this scheme by allowing the monitor to change the name of the representation's owner. In this way, for each representation it creates, the monitor could, for example, forcibly return any of its resources either to itself or to any other object.

The structure of a OPRT entry has become:

name of <represen-> tation	name of <object being> represented	name of <repr's > owner	name of <repr's > monitor	<timeout>

10.4.3 Copying

With each representation there are two attributes that can be passed about a network. One is the owner of the representation and the other is its monitor. Either of these attributes could be duplicated simply by holding a list of owners or monitors instead of just one. Since it is undesirable for every owner or monitor to be able to duplicate itself a status would have to be associated with each owner or monitor name in a OPRT entry indicating whether or not it is allowed to duplicate itself, the status being set when that owner or monitor was created. In this way an authentity can choose -

a) How many valid monitors to set up. (This could reasonably be more than one in a distributed application, and could also be zero if the initial timeout on the representation is large enough).

b) Whether to allow the number of monitors to vary dynamically (by either allowing or disallowing them the ability to duplicate themselves).

c) How many valid representations for the object there are initially (it may or may not be meaningful to have more than one of these).

d) Which representations (if any) can duplicate themselves (for whatever purpose duplication is meaningful): if there are any duplicates, the representation may not be able to be used as an "interlock" upon the object that it represents.

10.4.4 Path control

A limit upon the number of times that an attribute can be passed on could be established. It can be useful, for example, to give away a representation that cannot be passed onto any other object or to give one away which can only be passed on once. This type of control is called path length control and can easily be implemented in the OPRT by holding the maximum allowable path length that any particular object is allowed with its name in either a monitor or owner position and decrementing it each time that name is changed (that is, the attribute is passed on), refusing to make the change if the count has reached zero. This number can be decremented each time it is passed in one of two-ways. It can either be automatically decremented by one, in which case paths cannot be shortened, or it can be set to any number strictly less than the current path length, that the sender desires. A way of indicating an infinite path length would also be appropriate.

A second way to control what becomes a monitor or an owner is obtained if, with each OPRT entry for a representation, two lists of objects to which ownership and monitorship of that representation could be transferred are kept Attempts to change names in either the owner or monitor lists are allowed only if the recipient is in the relevant list. Alternatively the lists could be of objects that cannot be transferred to.

Such modifications are obviously endless (controlling the path lengths given to different objects if they are transferred to, for example, or including a list of objects that have the ability to change the access lists and so on). It is difficult to decide at what point to stop in the development of such a system. There are two conflicting arguments:

- for simplicity

  a) for the sake of relieving programmers of the tedium of having to decide amongst several different methods of accomplishing the same thing.

  b) because a simple mechanism is more reliable, smaller and possibly faster.

- for complexity

  a) flexible control enables applications to be written more conveniently and easily.

b) if used from a large operating system communication with the OPRT server will be moderately expensive in terms of interprocess communication, protocols and so forth. It is therefore more efficient to do a lot of work per "call" because the investment is high.

Although path length control looks attractive at first sight, in practice, it is not very easy to use. By limiting the number of steps in a path or limiting who may use a representation the implementation of information hiding interfaces is made very difficult. The details of the construction of these interfaces must be known for the user to determine, for example, who or how many services are involved in carrying out any given operation at an interface. Otherwise the object representations constructed as arguments may not be suitable for use. The only safe approach a client could take would be allow any service to use his representations and to give them an infinite path lengths. Even if the details of such interfaces were always available it is doubtful that programmers would construct representations to fit exactly.

10.4.5 **Problems**

It is rather important to discuss a question that has, so far, remained unasked. That is, what are the names used in the OPRT for representation owners? As noted under "Simple Mechanisms" above, some way of verifying these names must be available. When something wishes to give its representation away it must first prove that it owns it, which is done by verifying the identity of the sender (possibly using two-way authentication). This presupposes the existence of a more primitive representation system such as that provided by an AOT server. The alternative is to restrict the names in these positions to be some other easily authenticated value, such as station numbers on the Ring*. Hence an unstated assumption in the OPRT system is the existence of a less powerful representation system underlying it.

In addition, the protocol expenses noted above should not be underestimated. If the cost of using a representation system is too great in terms of time or ease of programming it will simply not be used. It would be difficult to promote the universal use of such a system in a research environment, for example. Also the need for any complex code at this level of function does not fit in will with the aim of autonomy mentioned in

---

* Station numbers are described in "Implementation Environment".

"Basic Network Structure".

10.5 Summary

Several orthogonal aspects of representation system design have been investigated and evaluated. In some cases problems have been found and in all of them at least some benefits have been noted. Some variations have been tried but most have only been subject to theoretical consideration. In general the choice of implementation was a direct consequence of such considerations.

Three different naming variations were discussed. Single-level names have the disadvantage that the representation system is rather vulnerable to attack and two-level names, which were shown to be equivalent to n-level names, prove to be rather inflexible without such servers as a Privilege Manager and the SOAP Service (in which this lack of flexibility is, ironically, rather manifest). Multi-level names bring format orientated problems and, although they show a great deal of flexibility, they can become too precise for practical use.

Dynamic names were briefly considered but were found to impose unacceptable reliance on the representations for highly powered names which may be required to exist indefinitely.

Pass once representations undoubtedly provide a better representation service but need relatively complex protocols for their use. There was some question as to where, exactly, the facilities that could be provided should end. In addition, the schemes considered could not really be considered as a variation of AOT design since they depend on something with a similar function to it as an underlying mechanism. These variations can, more realistically, be thought of as an extension or an additional layer of mechanism which can be provided once a representation system is already established.

Article 11

AUTHENTICATION

This article discusses the meaning taken for the word "authentication" and, having arrived at a definition, discusses the use of authentication and its implementation on a network.

11.1 Authentication

Consider the period for which a real object actually exists (for example a person). There are three possible types of interaction that this object may have with a given network.

(a) It is never represented on the network at all - in which case this object is of no interest to the network.

(b) It is represented on the network for the entire duration of its existence.

(c) It is represented on the network only during certain periods of its existence.

The vast majority of all objects known fall into category (a) and can be disregarded as uninteresting. Those objects which fall into category (b) pose no special problem with respect to their representation since this can be created at the time of the object's creation and kept valid until its existence is terminated. The objects in category (c), however, pose a special problem since they must be represented on the network for several different durations within the overall period of their existence: at other times they exist but have no representation on the network.

If an object from category (c) is to reclaim a network representation, some means external to the network must be found to identify the object. This identification will be necessary for the network's internal representation system called **external authentication.**

In general, authentication is the identification of an object that is external to the immediate environment of the identifier. "External authentication" will be used for authentication of objects external to a network, and "internal authentication" for the authentication of objects internal to a network but external to services.

Internal authentication can be provided, in quite a general fashion, by the use of the object representation system: it is just the verification of a representing token. Perhaps it is possible to provide an interface to the network from which things external to it could manipulate the equivalent of an AOT for external objects. This is quickly seen not to be a practical proposition because of the difficulty that some objects would have in manipulating such an interface (for example, consider the plight of a terminal which must remember a "TUID" that it was given on disconnection from the network whilst it resides in the workshop for repairs).

Since the objects that fall into this category are so varied (for example, disc packs, terminals, interfaces to other networks, the time, users, and so on) it would seem more practical to adopt a more flexible scheme and authenticate each type of object using its own type-of-object dependent authentication service.

11.2 External Authentication Methods

Since we cannot draw a useful analogue between the way in which internal and external authentication is best implemented, the different methods of external authentication must be investigated by their individual merits. Four basic methods of external authentication are considered:

(1) implicit

The network is constructed so as to regard the object as authentic without any formal checks.

(2) measurement

Some well defined static attributes considered to define the object uniquely are known and are checked against those of the object to be authenticated.

(3) memory

A previously authorized object can have some dynamic attribute (to "remember") that can be checked to facilitate subsequent reauthorization.

(4) trust

A trusted entity proffers an object as being authentic.

Note that methods (3) and (4) depend upon an authorizing mechanism already being available so that, in any network at least some of the authentication must be provided using either method (1) or (2).

Method (1) should apply to as few objects as possible. The more objects that are implicitly authenticated the larger the scope for error and system attack.

The network may implicitly authenticate itself, since it may either have no access to, or not recognize any higher authority. Any number of objects the identity of which it is not felt necessary to check, must be considered as implicitly authentic. This may very well apply to "fixed" lines, to external hardware and other objects that are considered an integral part of the network, even though they are external to it. (For example, on the Ring there is a service which relays information transmitted by radio from Rugby (the time) - no explicit check is made that it is Rugby that is transmitting on the particular frequency used: it is taken as implicitly authentic).

Method (2) can be used only for objects with measurable attributes that imply their identity. For example, if it were possible to automate the measurement of a person's fingerprints, this might provide adequate proof of his identity. The class of objects which are authenticated by their "address" falls into this category: their address being taken as a static attribute in this case (note the implication that the address may not change).

Method (3) can be used only with objects that can retain state (that is, those which have a "memory"). For example, another network, an intelligent disc controller and people are all able to remember some number, password or other piece of information to be used when they next need access to the network.

Notice that the simplified case in which the network always proffers, to the same object, the same number or password to "remember" is equivalent to the conventional password mechanism. In this simplified case the restriction on the objects which can use the scheme is less severe since the object's password can be "built into" the object without providing mechanisms for changing it. A terminal might have its password encoded on a set of hidden switches. This example shows that the distinction between methods (2) (static attributes) and (3) (dynamic attributes) becomes rather blurred in this area - where attributes may change, but only "slowly".

Method (4) refers to the following logic:

i) an object is authenticated as X if
object Y is trusted, and
object Y has authenticated the object as X

ii) an object is trusted if
it has been authenticated as some object Y, and
object Y is known to be trustworthy

Notice that this argument describes the circumstances under which method (4) works. It does not describe how those circumstances could arise. For example, it does not go into how object Y is to be found in i) nor how to reliably discover that Y has made the appropriate authentication. One of the ways in which this can be achieved is to use an "audit trail" in which each of the objects that have relayed the fact that an object has been authenticated (rather than authenticated it themselves) is listed. That an object has been authenticated by Y can be verified by verifying the audit trail. The audit trail can be verified by checking that each object listed within it is trusted. Such a mechanism is described in more detail in [JNT80].

The above also makes the difference between authentication and trust apparent. That an object is authentic merely means that its name is reliably known: this does not imply that the object with that name is necessarily trustworthy.

11.3 Authenticators

An authenticator, then, is something that makes the mapping from one domain of representation to another. In the case of a network the ways in which the above four methods make this mapping are as follow:

- method (1)
  The mapping is between the null domain and the network domain. (The object is inherently authentic and so is always authentic in the network domain - they need no representation in the source domain).

- method (2)
  The mapping is between the physical, measurable, domain (in which things are represented by their shape, weight, and other, perhaps more abstract static attributes) to the network domain.

- method (3)
  The mapping is between the network domain of representation (at some former time) to the same domain (at a later time).

- method (4)
  The mapping is between some other authenticated object's domain of representation to the network domain.

An authenticator on a network can use any of schemes (1) to (4) depending upon which is most appropriate for the particular kind of object that it is to authenticate.

No matter how the authentication is done, however, the result is always the representation of the authenticated object in the network domain. In mathematical notation,

$$\text{authentication: domain} \times \text{representation} \longmapsto \text{network representation}$$

In terms of the mechanisms developed in previous articles, this implies that each authenticator must be able to generate network representations of the objects that it authenticates. It must, therefore, have a valid representation for an authentity which will become the "type" of the representations that it generates.

A T-authenticator, which is defined as something that authenticates objects of type T, will consist of:

i) some way of authenticating objects of type T (for example, based on methods (2) - (4))

ii) some way of representing objects of type T (a representation for the T authentity, perhaps)

Article 12

USER AUTHENTICATION

This article describes the design and implementation of a user-authenticator using the ideas developed in the article called "Authentication".

As outlined in the previous article a user-authenticator consists of:

- some way of authenticating users
- some way of representing users

The second of these is solved by using the UID sets (AOT representations) developed in previous chapters.

12.1 User Authentication

There are many ways in which to check a person's identity. Some examples from the categories listed in the previous articles follow:

1) implicit

   If the user is communicating via a particular device (such as a terminal or a personal computer) to which he alone has access, then no check on his identity is necessary.

2) measurement

   A user can be identified by his fingerprints, voice pattern, the shape of his face or his signature. None of these measurements, however, can be trivially automated.

3) memory

   An authorized user could be given a number or password to remember each time he logs off which he must quote when next logging on. A common simplification of this method, pointed out in "Authentication" is for the system, effectively, to give a constant password to be remembered. Alternatively he may be given a unique credit card with a suitable magnetic encoding with which to regain access to the system.

4) trust

A user may already be logged on at some external system which proffers him as authentic by that system's standards. If that system is trusted this may provide the required authentication.

Similarly an already authenticated and trusted user may be allowed to proffer the user as authentic.

Naturally, it is not necessary to provide all of these modes of authentication. Indeed, since it was only necessary to demonstrate the principle, the password method (one of the simplest) was chosen.

12.2 Interface Design

The interface is divided into two parts, one set of entries which is universally available and another which is available to those that possess a certain "privilege"*. As in the AOT service the interface is supported by a set of SSP entries.

12.2.1 The user interface

The most important entry to a user authentication service is one which takes a user's name (a PUID) and a password and, if they match, returns a UID set representing that user which will be marked by the authentity used to generate it (the USER authentity). This entry is called AUTHENTICATE. For practical reasons AUTHENTICATE must also be passed a time for which the created UID set is to be initially valid.

In order that other systems on the Ring can exercise their autonomy (a network aim) to the extent that they need not use UID sets at all, a check entry is also provided in which a user's name and password are given and an indication of whether or not they match is returned. In this way external systems may share the same criteria for the authenticity of a user without providing the software necessary for using UID sets.

In both of these cases the return codes given do not distinguish the case of a user having no password at all from the case in which the user does have a password but it does not correspond to that given. This is deliberate in

---

* The privilege mechanism is described in the article called "Privilege Management".

order to increase the difficulty that an illegal user has of proving himself authentic, since it does not indicate whether the named user even exists.

In such a scheme there is no security lost if the system lets the user specify the password that he must remember to be authenticated when next he needs to be represented on the network. Clearly, however, the specifier must be authenticated as the correct user before he is allowed to change his password. In this service there are two possible means of authenticating a user. One is to check his name against his password and the other is to verify his UID set. Only the former is used because of the following reasons:

a) Verifying the user's UID set would have required his TPUID to be sent to the user authenticator. The user's security is best served by having to pass his TPUID to as few services as possible.

b) Autonomous services that use the user authenticator, which do not possess UID sets, would not be able to use the latter method.

c) As far as possible it is desirable to ensure that, if a user's password is changed, the user himself knows about it. (It seems more likely that a user will have to participate in the former mode of authentication than the latter because a password should not be kept by any network service after it has been verified). An interface to the user himself is therefore more suitable than one to the network.

The entry designed for the user authenticator that enables a user to change his password CHANGEPW therefore requires both the user's old and new passwords. Of the entries so far this is the first in which repeatability becomes an issue. AUTHENTICATE uses another repeatable interface and CHECK is repeatable in any case. In order to ensure that this entry is repeatable it succeeds if the existing password matches either the authenticating password (in which case it is then changed to the new one) or the new one (in which case no further action is taken).

12.2.2 The maintenance interface

Someone in authority who is in charge of the authenticator service will want to be able to do the following things:

(a) to enable a new user to authenticate himself with some initial password (when, for instance, a user joins the network)

(b) to deny a user the ability to authenticate himself (for example, when a user leaves the network)

(c) to change a user's password (perhaps, when a user has forgotten his password)

If entries are provided for these functions some way of checking that the user possesses the required authority is necessary.

Authentication for using these entries is provided using the privileges mentioned above. Privileges are UID sets with the authentity 'privilege'. Each entry requires a privilege for 'pwpriv' before it will be performed.

The functions provided by (a) and (c) are combined in an entry, SYSUSERPW. This entry takes a 'pwpriv' privilege, the name of a user and his new password and ensures that the user can be authenticated only with that password. If the user did not previously have a password he is given one and, if the user did exist, his password is set to the one given. Such an entry is obviously repeatable.

Finally, the entry SYSKILLUSER takes a 'pwpriv' privilege and the name of a user and ensures that he can no longer authenticate himself with any password at that service.

12.2.3 Interface summary

The following table summarizes the entries:

entry	arguments	results
AUTHENTICATE	N, p, △	t, d
CHECK	N, p	<none>
CHANGEPW	N, p, p'	<none>
SYSUSERPW	N, p, auth	<none>
SYSKILLUSER	N, auth	<none>

where d V i(t, USER\N)
and auth V p(PRIVILEGE\PWPRIV)
and p = user's current password
and p' = user's new password

12.3 Implementation Details

The address of the only user authenticator service on the Ring is given by the name server in response to the string "USERAUTH". It is implemented in a Z80 microprocessor. Users' names and passwords are held in a table in memory - there being enough memory to accommodate a reasonably large number of these.

Because the memory is volatile and the service has to be suspended sometimes, the table is written to a secret file on the Ring's file server each time the in-core copy is changed. This file is read in order to re-create the table when the service is reinitialized - the service being suspended until the read has completed. While the table is being backed up, it cannot be written to and so entries CHANGEPW, SYSUSERPW, and SYSKILLUSER fail with a standard return code requesting the client to try again soon.

In order to obtain the desirable features of fixed format request blocks passwords have a fixed size of eight ASCII encoded characters (a convenient size). Since the characters of the password are probably to be typed at a console, with an unknown character set, the syntax of a password is constrained to use letters, digits and the commonly available characters '.', '*', '-' and space. Furthermore the cases of the letters are equated in the matching algorithm.

There is no need to scramble passwords since the file in which they are saved is secret. However, a simple scrambling method is used in order to prevent passwords accidentally being read from memory dumps during periods when the service is being maintained.

12.4 Conclusion

The USERAUTH service authenticates users by their knowledge of personal passwords and can be used independently of the representation system. It relies upon the representation system for its maintenance since it uses privileges. It also relies upon the file server keeping a stable copy of its information which must become available before the service is restarted.

12.5 Post Script: SYSAUTH

Another service, whose address is given by the name server in reply to the string "SYSAUTH", authenticates "systems", such as reliable mainframes connected to the Ring, using passwords in the same way as USERAUTH. It produces UID sets with the authentity 'system' but otherwise has an identical interface to that of USERAUTH.

Article 13

SERVICE REPRESENTATION

This article discusses the ways in which we can extend the representation system to include the representation of services themselves. There are two categories of service which are considered: those that the system has created and those that it has not.

13.1 Service Authentication

If a service has not been created by the system a situation arises in which the service, which may have been operating for any length of time, is to be identified. Each of the four basic methods outlined in the article about "Authentication" bears examination:

(1) implicit

Some services will have to be regarded as implicitly authentic - services from which authority is initially delegated (such as is described in "The Source Of All Power").

Note that, on the Ring, this method reduces to trusting that a particular service exists at a particular address. Furthermore the address, unless it is that of the nameserver, must be found by using the nameserver. At least one service must use this method - the service from which all authority is delegated. This places the integrity of the entire system directly on the nameserver. It is, therefore, very important that the nameserver is trustworthy. It should, preferably, be extremely simple and dependent upon no other service.

(2) measurement

By definition the only access that the network has to a given service is through its interface. Services which have the same interface and which respond identically to the same requests might be considered identical - in which case a service's identity could be checked by "measuring" its interface. This would require all possible uses of the interface to be made and compared with the expected results. Even then, if the interface could be ratified in this way, no guarantee could be made that the implementation behind the interface did not contain a "Trojan horse" which, although seeming to perform the expected service was additionally misusing the data sent

to it. In short the identity of a service cannot normally be implied even by a very thorough investigation of "what it does".

(3) memory

So far "service" has deliberately been described very informally. Whether or not a service can "remember" data whilst it is inactive and not operational on the network begs certain questions about the more detailed nature of a service. In particular: "are services implemented in such a way that the environment of a previous incarnation is restored to it upon re-creation?". If so it is clearly possible for a service to remember something which it may quote to facilitate simple reauthentication when it is "woken". If not, such reauthentication is clearly impossible - since there will be nowhere to remember such information.

It will help to make the distinction between the period of activity of a service and its period of existence; in particular it must be recognized that they may not be identical. The case in which a service is created with a copy of a previous environment must be regarded as the continuation of that service and not as the creation of a new one.

A new service, therefore, cannot "remember" - because there is no previous incarnation from which to do so! However, once a service has been authenticated this mechanism could be used to reobtain authenticity when recontinued*.

(4) trust

If some trusted authority (such as a supporting operating system) proffers a service as one which it believes to be authentic, this may be reason enough to believe the identity of the service.

Thus the initial authentication of services (for example, those which have been newly created) can be seen to be something of a problem. Method (1) is clearly not suitable for the majority of services; method (2) cannot be applied easily (if at all); method (3) cannot be used for initial authentication of services and, although method (4) is applicable, it merely "passes the buck" by making another authority responsible for the authentication of the service.

---

* A server for providing a service's old environment was described in "Example Servers".

13.2 Authentication by Creation

The problems associated with identifying a service which already exists are not relevant when considering the initial creation of the service itself. A creator can represent a service that it has made using the existing representation system so long as it knows the service to have been constructed properly.

The elements through which a new service is trusted by its creator are as follows:

(a) The "recipe" that describes the objects necessary for the service's construction (for example, processes in an operating system, files of code or data, allocatable machines on the network, and so on) must be trusted.

(b) Each of the ingredients obtained to fill this recipe must be trusted.

In these circumstances the creator may create a representation of a new service - the representation's authentity denoting the creating server. Such a representation can be trusted by those who trust the operation of that server.

Note that (a) and (b) are conditions that apply in the context of the the creator, not of the network. Thus the trust involved could be conveyed by any form of representation convenient to the creator. The interesting examples are derived from using a local form of representation (for example, those available within the creator service, typically machine addresses, file names and so on) and the global form (available on the network), respectively.

13.3 The Use and Acquisition of a Service Representation

There are two main uses for a service representation: the first is to grant that the service may be used; the second is to imply ownership of the representation, and, therefore, of the service itself (using the TUID and TPUID of the representation respectively*).

---

* See the article called "Design of an Object Representation System".

Proof that use may be made of the service is likely to be employed at the interface to the service itself (if at all), whereas proof that the service is owned can be used as a means for the service to prove its own identity.

Naturally the service will have to possess this representation if it is to be used in either of the above ways. If the service representation has been created by the manufacturer of the service it will be necessary for the creator to ensure that the new service has implicit access to the new representation. That is, the new representation must be "built into" the new service. This is because, as long as it does not have its own representation, it is unable to prove its identity (as discussed under "Service Authentication" above) and will, in consequence, be unable to support a request to obtain it. And since, in general, the creator will be unable to send the representation to the service because the address at which the service acts may not be predictable*, neither the creator nor the service is in a position to initiate a communication that would result in the service receiving its representation.

Given that a service has its own representation it may be used in communications for proof of identity#. It may also be used to obtain any privileges or authentities that it is allowed (see the articles "Privilege Management" and "The Source Of All Power") and to obtain the environment in which the service customarily runs (see "Example Servers").

13.4 Summary

Practically speaking the identity of a service which does not already possess its own representation can be found either implicitly (that is, by address only) or by trust in some authority that vouches for the service.

---

* Certain parts of the address, such as its port number, may have to be allocated dynamically according to the distribution of addresses extant when the service commences action. Such a service might choose to advertise its address in some public place such as a (dynamic) nameserver.

Alternatively the service created may not possess an address at all if it does not expect any requests from the network (for example, a service which calculates π and prints out the result).

\# See the article called "Authenticated Communications".

The creator of a service may authenticate it if the objects from which it is constructed are authentic. Trusted creators, therefore, can be allowed to manufacture representations for the services they create. Since there is no general mechanism through which an arbitrary service that does not possess its representation can be given its representation it must be one of the component parts of a new service.

Article 14

RESOURCE MANAGEMENT

This article describes how object representations can be used to ease some of the problems associated with the fair and safe distribution of a set of objects around a network.

14.1 Component Parts

The environment assumed here is one in which, (a) there are a certain number of services (possibly only one) capable of creating or reinitializing resources and, at the same time, issuing representations for them and, (b) services that perform functions on these resources (these resources could include the issuing services). In many cases a object-creating services will also have to perform all the necessary manipulations on those objects since they may reside there solely.

Given that a particular creating service can serve only up to a certain number of resources at one time, some form of resource allocating strategy must be employed and embodied within the creator.

14.2 Resource Allocation

For each individual resource generated, a copy is kept of a representation which is sent to the requester. Both the ownership (TPUID) and use (TUID) parts are given to a new user. That resource is then considered to be in use for as long as representation is valid. The allocator may enforce a time restriction upon the use of the resource reclaiming the resource by deleting the representation (by virtue of the ownership part that it holds) after some management-dependent time.

As long as the representation is valid, its use part can be used to manipulate the object and may be distributed around the network at the new owner's will. The ownership part can be used to ensure that the object

represented stays in existence. Since the owner may be charged* for the duration of his use of the object, it will be in his interests to ensure that the object is returned (by revoking the representation at least) as soon as possible. He may choose to run the representation with a very short timeout and refresh it quite often in order to ensure that it will quickly disappear if he crashes.

Once the allocator has perceived that the representation for an object that it previously issued is no longer valid, it may reuse the resources freed in any way it may see fit.

14.3 Allocatable Machines Example

Consider a network in which a pool of computers is kept for general use. Each machine has a simple interface to the network of the form power machine on/off, read/write memory locations, start/stop executing, and so on.

These machines are allocated to appropriate owners by a "machine manager" service. Having been allocated a machine, it is desirable that, subject to management constraints, the owner possesses the exclusive capability to use it. The threat of the owner's program being overwritten, or otherwise tampered with, by an undistinguished network user is unreasonable.

Management constraints might include halting a machine or withdrawing it from the allocation pool for such reasons as: time allocation expired; no longer in use; or, caused a problem elsewhere on the network.

An owner of an allocated machine might want to load it with either his own or with "system" code; he might want to "debug" it; he might want to give the machine away as a parameter to some service entry (for example, a "dump service"); or, he might simply want to keep it and reallocate it to his own client.

A system with the above potential may be formulated by using UID sets to refer to allocated machines: each machine is given a name (a PUID) and the machine manager is given UID sets for authentities corresponding to the different types of machine; 'auth\machineA' and 'auth\machineB', for example.

---

* See the article called "Example Services" for the description of an accounting service.

The interface to each machine checks a UID set given with each request, ensuring that it has the correct authentity and represents the machine being used. Thus only the possessors of a UID set for an allocated machine will be allowed access to it.

The machine manager keeps a list of the UID sets that currently represent each machine for which it is responsible. Machines without associated UID sets, or with invalid ones, are eligible for allocation. After a certain period of time, fixed when the machine is initially allocated, that machine's UID set is deleted (by virtue of the TPUID that is held, thus revoking the UID set currently referring to that machine).

When a machine is allocated the new UID set generated to represent it is passed to the user. The user must use the TPUID in the UID set to maintain possession of the machine since, if it "times out", not only will the user be unable to use the represented machine but the machine manager will eventually reallocate it. The user may, at any instant, "give back" the machine simply by revoking his UID set.

Machine Manager's Table

TPUID	TUID	MACHINE PUID	AUTHENTITY
d1*	t1*	mc1	machineA
d2	t2	mc2	machineA
-	-	mc3	machineA
d4*	t4*	mc4	machineB
d5	t5	mc5	machineB
-	-	mc6	machineB

'mc3' and 'mc6' are machines ready for allocation. * marks invalid TUIDs and TPUIDs: 'mc1' and 'mc4' are also ready for allocation. 'mc2' and 'mc5' are allocated and in use.

This form of resource allocation is not peculiar to machines; it can easily be modified to deal with many other types of resource which can benefit from a network representation. In general one service, designated as that object's allocator, is responsible for keeping information about each of the objects that it has allocated, including the UID sets that it

has produced for them. Other services which can either use or manipulate these objects only do so when a valid UID set is delivered.

Article 15

AUTHENTICATED COMMUNICATION

This article describes the ways in which object representation can be used for identifying each of the parties in a communication. The nature of authenticated communication is discussed, followed by a description of protocols involving participants at just one and then both ends of an unidentified communication. Finally an existing protocol, authenticated BSP, is described.

15.1 Identification in Communication

BISCM (see the article called "Base Network Structure") already offers a mode of communication whereby users have adequate faith that information sent to a particular address will arrive safely. However this is not sufficient to ensure that communication with a service of a particular identity will arrive safely. Identification in communication here does not, therefore, refer to the problem of ensuring that information reaches a given address - this problem is presumed to have been overcome.

"Identification", in the sense used in this article, refers to the identity of the sender and receiver in a communication as expressed in some form of representation (in particular that of previous articles). The problem is to map identifiable communicating entities onto their addresses safely: to ensure that a given service actually does reside at a particular address. It is the solution to the problem of untrustworthy addresses rather than to the problem of untrustworthy service identifiers that is being sought.

Services are divided into two broad classes, static and dynamic, corresponding to the rate at which the mapping between each identity and its address changes. A static service never changes its address (or if it does, only very infrequently), whereas a dynamic service may change its address quite frequently (or disappear from the network entirely). There are, therefore, three classes of communication corresponding to whether:

- the sender and receiver are both static
- only one of them is dynamic
- both of them are dynamic

In any communication a service (either sender or receiver) may want to authenticate the other participant. If the latter is static then it may be authenticated by its address: otherwise its representation can be used to facilitate authentication. Authentication might be desirable because:

(a) The sender may wish to ensure that the recipient has a specific identity.

(b) The receiver might like to know reliably the identity of the sender.

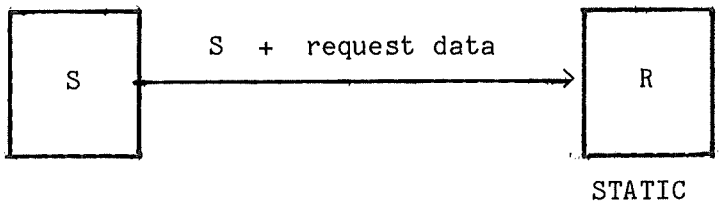
Authentication, when it is necessary, is simple when static services are involved since "authentication by address" simply implies trust in the nameserver (see "Service Representation").

15.2 One Way Authentication

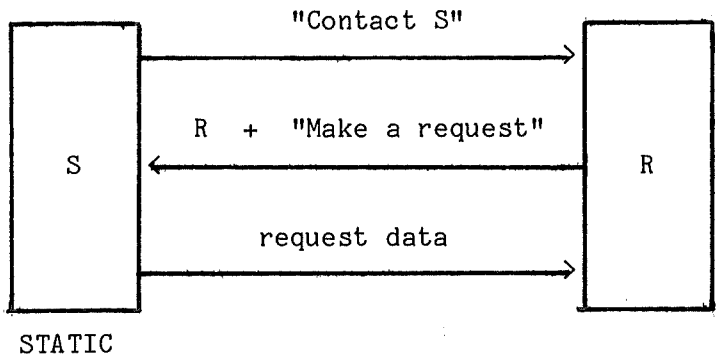
Authentication can be simply achieved by passing the representation of one service to another if either the sender or receiver is authenticated (a static service for example).

It is implicitly assumed above that the act of communicating with a service that has been authenticated (by address or by its representation) implies a certain amount of faith in the recipient in respect of the data sent to it. In other words it is assumed that information (including the representations of important objects) is not deliberately passed to untrustworthy services.

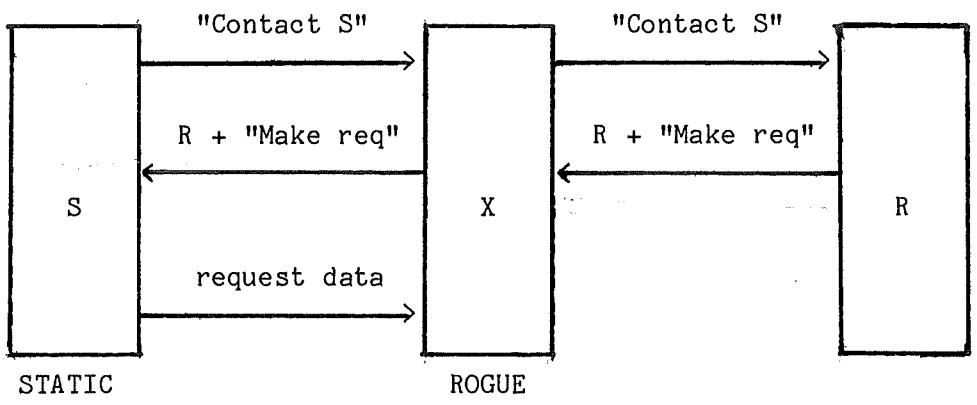
If the receiver is authenticated, part (a) above is already fulfilled and (b) can consequently be achieved by the sender transferring its own representation to the receiver - along with other information such as the data for the request.



If the sender is authenticated, exactly the same mechanism can be used by reversing the roles of the two parties and starting off the transaction with a request from the sender to the receiver for the receiver to attempt a communication with it.



Note: that the latter solution only applies if R gives its identity to static services that it trusts. If there is a chance of R sending its identity to an untrustworthy service then the following deception could be practised:



Furthermore, having obtained representation R, X can practice this

deception without R's aid from then on.

15.3 Two-Way Authentication

If neither party in a communication has authenticated itself to the other, and neither is static (so that they cannot be authenticated by address), a deadlock occurs: although each party has the potential address of the other, neither dare send its representation in case this address is found to be incorrect. The consequences of a service sending its representation to an unauthenticated recipient are two fold:

- Firstly, the recipient may turn out to be a "Trojan Horse" and use the representation to make service requests in the original service's name.
- Secondly, if it becomes known that the service sends its representation without prior identification of the recipient, other services would become unwilling to accept its representation as proof of its identity. In effect the service would "get a bad name" for itself.

Any form of two-way authentication must, therefore, provide some way for one of the participants to find the address of the other reliably. This breaks the deadlock and enables the holder of the valid address to either send its representation to the other or pass it its address in the same authenticated way, so that both services become authenticated by address.

The problem which has to be solved, then, is that of reliably passing an object (an address) from one service to another in such a way that the eventual recipient can verify that that object was sent from (or was created by) a reliably identified service. This is simply achieved using a representation system - giving the address a name (for example, a PUID) and creating a representation under the authentity of the service at that address. The allocation of names to addresses can be done using either a central mapping provided by a trusted service, or by a simple and widely known mapping of addresses to names (packing various components of an address into the "random" part of a PUID for example): it will be seen that this mapping does not have to be reversible.

A two-way authentication protocol would start in the following way. It is assumed that both participants (named by PUIDs A and B, say) are in possession of their own identities (as discussed in the article "Service

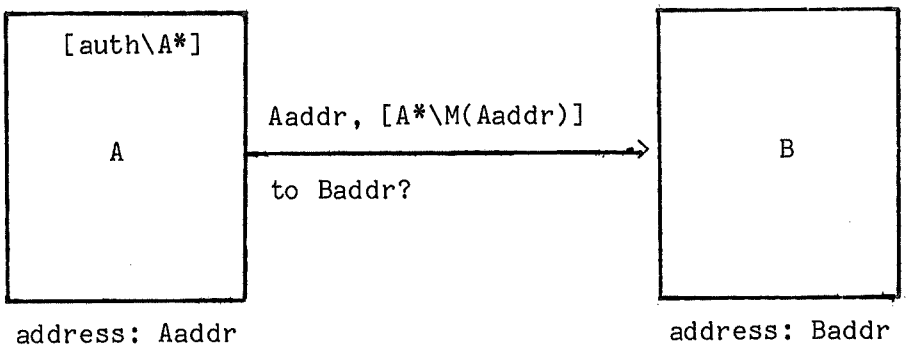
Representation") and that there is a convenient mapping,

M: address $\longmapsto$ PUID (a mapping which finds a name for an address)

In addition, at least one of A and B must be able to create objects (in this case representing addresses) under a well known authentity unique to either A or B - A* or B* say*. Suppose that A's address is Aaddr and B's is Baddr and that A has an authentity representation (for A*, that is). The notation

P(q) $\rightarrow$ Q: a,b,c,d

will be used to indicate that service P sends service Q (at address q) information consisting of items a,b,c and d. X will be used to denote an unknown service.



Nomenclature: [X\Y] V p(X\Y)
(this is, [X\Y] is a TUID for X\Y)

(1) A creates a new (quite short-lived) object representation, t, under authentity A* with a name derived from its address, Aaddr, such that:

t V p(A*\M(Aaddr))

(2) A sends t to the address at which it believes B to exist (Baddr?, say) along with its own address, Aaddr, the authentity A* and a request for authentication.

A(Baddr?) $\rightarrow$ X: "authenticate", t, Aaddr, A*

---

* There is no reason, in fact, why A* should not equal A and B* equal B.

(3) B may receive the request from A for authentication under the two circumstances: the request has come directly from A, or the request has been echoed by intermediate services (possibly posing as B). B must, therefore, check the integrity of the arguments by verifying that

$$t \; V \; p(A^* \backslash M(Aaddr))$$

since if any have been corrupted in transit this will not hold true.

Note that because the address, Aaddr itself, was sent and because the mapping M is available to both A and B it is never necessary to deduce Aaddr from M(Aaddr) and so M does not need to have a simple inverse (that is, M may lose information*). This having been done the following line of argument is followed:

- only service A can get authentity A*
- a service with authentity A* must have created t

  => this message originated with A
- service A is a known and trusted service: it does not lie about its own address
- t represents the address Aaddr

  => A's address must be Aaddr

B is now at a position in which it can choose (on the basis of whether it trusts A) to prove its identity to A. It can use either precisely the same protocol to inform A of its address, or it can send its own identity to be checked in reply. Note that communication with A is via the address Aaddr that B has already validated, and not to the address from which it received the initial communication. If B were to reply to that address it would be possible for a rogue service to interpose itself between A and B reflecting authentication messages in both directions and stealing or modifying the resulting communication. (B should reply even though it did not receive this information from Aaddr: A may have been unaware that it was not talking to B - a reply will update its version of B's address).

---

* The mapping M should not loose too much information. The more addresses that map onto the same result the greater the chance that an invalid address will map onto the same result as Aaddr. In the extreme, if M lost all the information in its argument, all addresses would map onto the same number: this would clearly be unsatisfactory.

The authentication protocol and the ensuing communication are subject to errors resulting from the service at a particular address being quickly replaced by a rogue service which is aware of the current state of the transmission. There are two ways to view this problem:

- It is too unlikely an event to consider seriously.
- It will not happen if communication is only attempted between services provided by (identifiable) trusted and "well behaved" boot services (since it is a boot server which will be responsible for effecting the sudden change).

To summarize the assumptions and prerequisites from the above they are:

a) Having identified each other, A and B trust each other (to act in a predefined way) before the main communication commences.

b) A and B possess their own representations.

c) A and B can find their own addresses.

d) A and B can exclusively obtain authentities A* and B* respectively.

e) There is a mapping M from addresses to names which is reliable.

f) Addresses refer to the same location when used at both Aaddr and Baddr (that is, global addressing is assumed).

a) is an obvious prerequisite;

b) can be ensured (see the article called "Service Representation");

c) can normally be provided trivially;

d) can be ensured if A*=A and B*=B (see the article called "Variants in AOT Design" about Multi-level names) or otherwise by judicious use of the SOAP service (see the article called "The Privilege Manager Service");

e) can be provided by a trusted static service if no obvious mapping suggests itself;

f) can be overcome by appropriate address transformations as Aaddr and Baddr are passed around the network.

15.4 Authenticated BSP

Authenticated BSP is a simple one-way authentication scheme based on BSP (the byte stream protocol introduced in "Implementation Environment") in which the initiator sends its identification to the receiver used in the initiation of a connection. It uses mechanisms similar to those used in "Transport Service BSP" [JNT82].

15.4.1 The Principal

For every communication there is a single "principal". That is, a person, service, or other object that is responsible for causing the communication. It is the principal that is responsible for any consequences of the communication. These include the cost of the communication and responsibility for any "illegal" action perpetrated during it. Conversely, any benefits arising from it are accrued to the principal*.

Obviously, the principal must be known for the entire duration of the communication otherwise there would be nothing on which to blame illegal activities (such as generating an unreasonable number of protocol errors, attempting to use facilities and resources to which the principal is not entitled or simply causing expense for which nothing can be charged). Corrective or punitive action can take place only if the identity of the principal is known when the illegal action takes place.

For this reason it would seem that, for any given communication protocol, the identity of its principal should be sent before the communication itself is established. That is, the principal of the communication is a fundamental parameter of the protocol that implements it: it cannot be part of the data communication itself#.

---

* This use of the term "principal" is explained in [Saltzer75]. It was first used in [Dennis66].

\# This is because other data, which may cause an "illegal action" may precede it - leaving nothing to blame. Naturally this could be overcome by insisting that the identity of the principal is always the first data item sent. However, this is tantamount to extending the protocol to include authentication information in which the principal for the communication has, indeed, become a fundamental parameter.

15.4.2 TSBSP

Transport Service BSP is a transport service based upon BSP which conforms to the specifications in [Yellowbook80]. It supports two kinds of stream simultaneously, one consisting of the data and the other consisting of a series of "messages" which forms the control data. Each message consists of a message type byte ("octet") and a number of parameters. The number of parameters is determined by the message type. Trailing null parameters may be suppressed. TSBSP defines several message types, their function and their parameters. Each parameter can be of arbitrary size and, if necessary, is divided into several consecutive "fragments". Fragments have a maximum size of 64 octets. Each fragment is preceded by a header octet giving its length and specifying whether or not it is the last fragment in the current parameter. A zero header octet is used to indicate the end of a message.

The control messages are first specified in the user parameter part of the BSP "open" block and continue in the 'control blocks' of the BSP. A short message should be able to fit entirely within the open block. The intention here is to define such a message which contains authentication information from the system described in previous articles.

15.4.3 Authentication message

Both the "open" and "openack" blocks of the current open protocol have space for user parameters (these are the first outward and the first returning messages in the communication respectively). Only the open block is discussed here. A message of the proposed size will, if used as the first message, always fit entirely within the open block. Such a message is defined as follows:

Message type octet:	64
Number of parameters:	1
Length of parameter:	24 (octets)
Parameter contents:	1st 8 octets: TUID
	2nd 8 octets: PUID
	3rd 8 octets: AUTY

The deficiency of this message is major. The message type used is necessarily new and, unfortunately, gateways which conform to [Yellowbook80], around which specification the format of this message is based, can not pass messages of unknown message type. (Refer to

"Authentication and the 'Yellow Book'", below, for further details). It was decided, initially, that the format of the message was to be context dependent in that it was assumed that any recipient would know what to do with the data it contained. A more general format would, perhaps, have split the message up into three parameters: the address at which the authentication information was to be verified, the data to be ratified and the (secret) data to use. This would have a better chance of general acceptance but, given the above problem, this chance would remain small.

The AUTY field is included so that the receiver understands exactly what name the principal wishes to be known by. The recipient is free to try to verify the TUID and PUID under a different authentity but the authentity that the principal claims to have is the one given in the AUTY field. It is important that the recipient does not simply check the TUID, PUID and AUTY and then allow access if they verify. An explicit check that the authentity is one which the recipient is prepared to accept should be made. Users, for example, may be given their own authentity representations to experiment with - the representations they produce will be verifiable but may well not be trustworthy.

15.4.4 Replug

There used to be a facility in BSP which allowed an existing BSP to be "replugged" so that one of the participants could be replaced (see "Implementation Environment"). This presented something of a problem since there was no general mechanism provided for passing user parameters through to the other half of a BSP. Such a mechanism was necessary if "messages" were to be relayed by the REPLUG initiator. Also the mechanism so far described is unidirectional. A BSP initiator sends credentials to a BSP receiver, not vice versa. Replug did not cater for such asymmetry in the reconnection, it sent the same request to both ends.

It was true, in this particular case, that the BSP would not have arisen were it not authentic to the initial recipient, but the recipient of the replug could have been unable to tell whether or not the initial BSP was normal or of the authentic variety, and even if it could it may not have shared the same criteria for trust as did the initial recipient.

The problem could have been overcome by appending the authentication message to the end of a REPLUG block thus defining the rest of the block to be "user data". In a TSBSP environment an authentication message could have

been sent automatically upon receipt of a REPLUG block. However, a more promising solution seemed to be the phasing out of REPLUG altogether, and that, in fact, is what happened.

15.4.5 **Example uses**

This protocol is used for file transfer and terminal connection. In both of these cases the TUID, PUID and AUTY mentioned above refer to the user of the given service. In the file transferring protocols access to the filing system being used is that of the authenticated user. In the RATS protocol (see "Implementation Environment") the user does not have to go through normal logging on procedures.

15.5 Authentication and the "Yellow Book"

[Yellowbook80] defines no facilities specifically to support authentication. The area is listed as one needing further study. However, the Transport Service Implementors Group soon found a need to support authentication and accordingly deliberated the point to emerge with a way in which authentication information could be supported by a transport service conforming to the standards laid down in this reference.

For the same reason that the open block was chosen as a suitable vehicle for the authentication message invented for Authenticated BSP, it was decided that the initial CONNECT message should be used. In this way the principal of the communication will always be identified before the body of the communication itself.

The CONNECT message consists of only four parameters: a called address, a calling address, a quality of service parameter and a parameter called "explanatory text". A fifth parameter could easily be inserted into the message for authentication data conforming to the protocol for forming messages but, unfortunately, it would not be passed on by gateways (which will expect only four). The explanatory text parameter could not be used since its contents are likely to be uppercased, printed out or even discarded altogether at different points along the message's route. The quality of service parameter has no definite syntactic content and is, by and large, unused. It is provided for gateways to use when determining the next path of the route to be followed, and possibly to set it up in a particular way. As such it is not suitable for the transportation of

authentication information.

The calling and called address parameters are transformed as they pass through gateways so that the called address becomes shorter and the calling address becomes longer. At each stage the **active part** of the called address represents the immediate destination and the rest of the address, delimited by some character, is passed on as the entire called address (possibly after transformation) for the next stage. The decision made was to define some syntax that can be included in the active part of the name for any authentication information necessary at that immediate address. The syntax used included keyed or positional textual parameters enclosed in brackets immediately after the address part to which it refers. The two main parameters are those for identifier and password.

Since [Yellowbook80] could not be redefined, or, as it turned out, even enhanced upwards compatibly, the Transport Service Implementors Group came to the decision that it made, with very few alternatives. The called address parameter is by no means likely to be exempt from being printed out at various points along the route, complete with names and passwords. Authentication information such as a UID set does not fit well into this scheme. A UID set must be transformed into a character format (one character per hexadecimal digit for example) then the authentity and PUID can be combined to form an "identifier" and the TUID could be used as the "password".

15.6 Encryption

By and large, secure communication can be achieved using encryption techniques. In addition, many authentication protocols have been proposed based on encryption. This section attempts first to point out the difference between these two applications of encryption, and then to advocate the use of an alternative system of authentication (that described in former articles) in preference.

15.6.1 Communication and authentication

The use of encryption in both the above areas has lead to a certain amount of confusion over their individuality.

The field of private communication is concerned with delivering data, without change, to some address on the network (and no other). The existence of a secure communication mechanism implies that it is possible for any service to ship arbitrary data to another in secrecy.

The field of authentication is one of the facets of the larger topic of object identification and representation. The existence of a secure object representation mechanism implies that names can be used, and their validity can be irrefutably assessed.

Encryption can be used to provide private communication simply by rendering data unintelligible to everything other than the services taking part. Such a system relies upon the relationship between data "in clear" and encrypted data being "one-way" - that is: there is a relation between clear text and encrypted text such that the encrypted text is easily found from the clear text but not vice versa (without the aid of some additional information).

A relation such as this can also be used to implement a secure object representation system (see "Representation of Objects"). Possession of a representation for an object implies that the holder can not only name the object but can also check that some authority guarantees that this representation does, in fact, represent the object with a given name. Such a check amounts to verifying that a particular relation holds between the object representation and its name. This relation must also be one-way since, although it could be possible to find the name of an object given its representation, it must be impossible (or at least very difficult) to obtain an object's representation given just its name. Thus, because it supplies this relation, encryption can be used to provide secure object representations. However, using it is rather like using a sledge hammer to crack a nut.

As an example of how encryption could be used to implement an object representation scheme, consider the following method which involves an object representation server to provide the authority that the holder of a representation verifies:

(1) The owner of some object (a physical resource, say) asks the server to provide a representation for some named object.

(2) The server encrypts the object's name with its private key and returns the result.

(3) The result is used as that object's representation on the network.

(4) When the representation is to be checked it is passed back to the object representation server with the name of the object that it is thought to represent.

(5) The server again encrypts the name and compares it with the representation indicating to its caller whether they are identical or not (that is, the representation is or is not valid).

It should be noted here that no attempt is made to solve the myriad problems that arise from an insecure communication medium (which might, for example, result in a representation becoming misappropriated) because an environment in which secure communication already exists has been assumed. In other words the two areas of private communication and object representation have been separated to the extent that they form two separate implementation layers of a secure network - one being based upon another.

It is because encryption can be used in both of these fields that the terms "authentication" and "private communication" have become almost synonymous with "encryption". However, as was noted above, it is possible to segregate these two areas, possibly providing alternative solutions to each. As an extreme, in an environment in which the only communicating entities are the processes of a single operating system, private communication can be provided by simply trusting the message sending software, and object representation can be implemented using capabilities. It would obviously be a mistake to use encryption for both (in fact, either) of these fields in this case.

15.6.2 Segregation of function

In the above example of an object representation server, encryption is neither necessary nor, because of the large amount of effort associated with encrypting and decrypting data, is it desirable. The one-way relation that it provides can be modelled explicitly by a table (as in a relational data base) as discussed in the article "Design of an Object Representation System". The use of an explicit relation in this form can clearly be more efficient than using encryption. It has the following advantages:

(1) It does not incur the overheads associated with encryption and provides a faster, and therefore more useful, service*.

(2) It can be used by services that do not possess the ability to encrypt or to decrypt (thus services can potentially be simpler).

(3) It allows revocation of representations because the representation is explicit and dynamically alterable.

By explicitly segregating private communication and object representation issues (for example, by implementing them differently), many of the problems which used to be viewed as difficult or complex in that combined area reveal themselves to have quite simple solutions when viewed as part of one or other of the component fields.

As an illustration, consider the question "Which aspects of a private communication protocol are necessary for the support of remote user authentication?". The answer is "None." - user authentication is a matter for solution by object representation methods: the only constraint upon the protocol is that it should be able to support an object representation system generally, no specific aspect of it can be pertinent to users explicitly. Similarly, "problems" encountered in accounting, remote job control, resource management, and the authentication of external objects generally are independent of protocol and communications issues. They find solutions when regarded as problems in object representation.

---

* To be fair it should be stated that very fast encryption devices are now available that allow the amount of data in a representation to be encrypted in quite acceptable times [Newman82]. It might also be fair to point out that the representation system could possibly go much faster with hardware support.

Article 16

UID SETS ON TRIPOS

This article describes the use of object representations in an operating system (TRIPOS) and the attendant enhancements to it. In particular, the idea of a "fridge" for UID sets, the use of the Userauth and Privman services during the logging on procedure, the commands for interacting with those services, and some other system changes are explained.

Since TRIPOS runs on simple, single user machines this is a real example of representations being used to provide protection in a, previously, completely unprotected system.

16.1 TRIPOS

TRIPOS is a simple, locally written, portable operating system which specifically aims to support only a single user on an unprotected machine. It is fully described in [Knight82]. It is almost entirely written in the BCPL programming language and, amongst other things, provides a filing system with a multi-level directory structure, easy process creation*, a fast inter-task communication method (messages are passed by reference) and flexible pseudo-stream devices.

All of memory is available to each task. The user is expected to be aware of the unprotected nature of his computer. He is responsible only to himself for its state.

In order to run TRIPOS in the environment described in "Implementation Environment" the user must obtain a connection to a service called the Session Manager to which a small vocabulary of commands can be given. The user then requests a version of TRIPOS, whereupon a computer from a pool of spare machines is selected and loaded with code appropriate to TRIPOS on that particular type of computer. TRIPOS then comes to life and, using information from the Session Manager, opens a virtual terminal protocol BSP

---

* On TRIPOS processes are called "tasks".

to the client's terminal.

16.2 The Fridge

All UID sets currently owned by the user or the system on TRIPOS are held in an easily accessible list called "the fridge". System software periodically refreshes those UID sets that it can (that is, those with valid TPUIDs)*. In this way UID sets in the fridge are kept "fresh" until either the system crashes (in which case the UID sets will "go off" an hour or so later), or the UID sets are explicitly deleted or withdrawn from the fridge.

All tasks have access to the fridge and may insert, copy or delete any of the UID sets therein (the interlock on the fridge being ensured by the use of task priorities). Since all of memory is available to each task, it was thought to be pointless to maintain one fridge per task since each one could still access another's UID sets. However, subsequent experience has shown that it is useful to associate UID sets with an "owner" task even in a completely unprotected environment. This is simply because a task sometimes needs to access a UID set that it, in particular, has placed in the fridge, and not one of a similar name that another task has placed there.

By convention, the first UID set for a task on the list that constitutes the fridge represents the user of the system for that task. Hence whenever this UID set is deleted the rest of the fridge must follow suit (to prevent any other UID set becoming the first).

There are two user commands on TRIPOS that can be used to interrogate the fridge. They are the USER and the UIDEDIT commands.

---

* Currently each UID set is refreshed for an hour every thirty seconds or so. The short timeout is to ensure that UID sets newly placed in the fridge do not timeout before they are first refreshed, and the long timeout ensures that, in the event of a crash, there is enough time to salvage the UID sets manually if necessary.

16.2.1 **USER command**

This command normally types out the name of the first UID set in the fridge - this giving the user of the system. By quoting command line parameters the whole fridge can be listed (optionally with each UID set's TPUID and TUID) or a particular (or every) UID set can be deleted from the fridge. A single UID set to be deleted can be specified by giving both its PUID and its authentity. The first UID set in the fridge with that name is then removed and, if possible, deleted. Since the most popular items for residence in the fridge are privileges the authentity will default to 'privilege' if only the PUID is given.

16.2.2 **UIDEDIT command**

This command enters an interactive "fridge editor" for manipulation of the contents of the fridge. It keeps an independent receptacle (called "CURRENT") into which items from the fridge can be copied and upon which a selection of operations can be performed. The use of an independent receptacle is necessary in order to obtain a fixed handle on a UID set because the contents of the fridge might, at any time, be altered by another task. CURRENT can be copied back into the fridge at any time with the proviso that the UID set it contains must not be already resident.

Aside from being able to transfer its contents to and from the fridge, CURRENT can be edited, written to a file, read from a file, or used as an argument to the various AOT service functions. When writing to a file the UID set is encoded by adding a value derived from a password to the random part of the UID set. The UID set is decoded with the aid of the same password when it is later read from the file. This password can be set and checked during the edit session.

Of the entries to an AOT service two, verify and identify, are used automatically when listing UID sets. The others are used on the UID set in CURRENT explicitly. This UID set can be refreshed for an arbitrary number of seconds (or deleted by refreshing it for zero seconds) or it can be used to generate new named UID sets (with either ENHANCE or GETTUID) that are automatically put into the fridge. When refreshing CURRENT it will often be necessary to remove its duplicate from the fridge in order to avoid the normal refreshing mechanism resetting the UID set's timeout.

16.3 Textual Names for PUIDs

When using utilities on TRIPOS which involve UID sets it is often necessary to quote PUIDs. This can always be accomplished using a 16 digit hexadecimal number. However, this is not always very convenient for users who are unlikely to be able to remember all the numbers necessary. In consequence, TRIPOS supports a mapping between textual names (mnemonics) and PUIDs. Thus, instead of using a 16 digit number, users may chose to use its unique mnemonic. However, if the user does not trust the mapping to give what he expects from any particular mnemonic, or if there is no mnemonic for a particular PUID, the hex number can always be used.

On other systems there may already be names for certain types of object (users, for example) and these can be mapped onto the more global PUID for that object. It is not necessary for these mnemonics, which are likely to be necessary in any operating system using UID sets, to have global significance (although this would obviously be desirable).

These mnemonics are used in the USER and UIDEDIT commands above and everywhere else that a user is expected to provide a PUID.

16.4 Logging On

During system initialization a program called START is loaded and executed in such a way that it cannot be interrupted or avoided by the user. The program requests the user's PUID* and password. The Userauth service is used to verify that the user's password is correct and, if it is, generate a valid UID set to represent him which is placed as the first in the fridge. If the call to Userauth fails, the interaction is repeated up to twice, after which the computer is returned to the free pool.

As well as his name and password, a user may request a particular status for his session. There are several different possible forms of TRIPOS: a version with a restricted set of commands for undergraduate students, and a more comprehensive version for research students. In both of these cases there is a normal and a maintenance version (in which an additional part of the filing system where the sources of commands and so on are kept is

---

* The PUID will normally be specified in its mnemonic form.

accessible). Two privileges are used to distinguish those who are allowed to request the research student version (LABPRIV) and those that are allowed to request the full version of the filing system (TRIPPRIV). In a more general case one could expect START to check the existence of several such privileges. These privileges would fall into two main classes: those necessary by dint of the version of the system that is being run (for example, LABPRIV); and those implied by the status requested by the user during the logon interaction (TRIPPRIV for instance).

To cope with these requirements, START takes a bit map specifying which privileges will be necessary when it is called and adds bits corresponding to those privileges that the user requests during logon. If the user is allowed the privilege for which a bit is set in the bit map the map is returned to the caller indicating which privileges the user has acquired. If the user is found not to be allowed one of these privileges the machine is returned to the pool.

Currently the privileges are checked by a request to the Privilege Manager asking whether the given user would be allowed them on request. This is not really the way in which the Privman service is intended to be used (it is simply being used as an access control list here). In future configurations, the initial connection to TRIPOS is expected to carry with it a UID set for the user. When this is the case the privileges will be checked by accepting and checking additional privilege UID sets.

16.5 Service Interaction Commands

Commands for using the various services described in previous articles are available under TRIPOS. For direct manipulation of AOT services UIDEDIT can be used. To interact with the Userauth and Sysauth services (which have very similar interfaces) the LOGON command is used. Privman and SOAP services (which also have similar interfaces to each other) can be manipulated with the PRIV command. The UIDEDIT command has been outlined above.

16.5.1 LOGON command

The LOGON command is used to manipulate the password tables of the Userauth and Sysauth services. One parameter indicates which of these services is to be used, otherwise the command is identical for either (except that the names used are of users or systems respectively).

If no other parameters are given, a name and a password will be requested, checked by the service and, if the check is successful, a UID set will be returned and put into the fridge. The fridge has its contents deleted before the check is made.

One parameter enables a user's password just to be checked without generating a UID set. A message is given indicating whether a given password is that of the given user. If no PUID for a user is given, it defaults to that of the current user.

Another parameter is used to change a named user's password. Both the user's current password and the password to which it is to be changed must be given. The new password must be typed twice identically in order to avoid users setting the wrong password by mistake.

A further two parameters enable the addition and deletion of new names to or from the password table. The fridge is searched for a privilege (PWPRIV) which is needed for these operations - an error message being generated if it is not found. The privilege is not obtained on the user's behalf if it is not there. This in not simply because the location of a route to a particular privilege may be beyond convenient automation; it is up to the user to define his own level of privilege and to be aware of what it is at all stages of his session.

16.5.2 PRIV command

The PRIV command is used to manipulate the tables in Privman and the SOAP service that say which object is allowed which privilege or authentity. As in the LOGON command, one parameter distinguishes which of these two services is being used and the command line is otherwise identical.

If only the PUID for a virtue (that is either an authentity or a privilege) is given, an application is made to bestow that virtue on the logged on user (that is, check that the user's UID set allows the virtue to

be claimed and, if it is, enhance the user's UID set with a new one for the virtue). Additionally a parameter can be quoted which will grant rather than bestow the UID set for the claimed virtue (i.e. an independent UID set will be returned).* In either of these cases the UID set for the new virtue will be stored in the fridge. The virtue need not be claimed by presenting the logged on user's UID set - the PUID and authentity of any UID set in the fridge can be given instead.

Another parameter allows the user to find out whether he would be allowed a particular virtue if he possessed a valid UID set for himself (or any other named object). The fridge is not used at all in this case (except to find the user's PUID, by default).

A choice of a further two parameters can be made to either delete or insert the ability for a particular object to claim a given virtue. In either of these cases the fridge will be searched for an appropriate privilege which will be used to authorize the operation. Here again this privilege is not automatically claimed on behalf of the user if it is not found.

16.6 Other System Uses

Prior to the use of UID sets in TRIPOS, there was no user identification system. Since then user identification has been used in various places (although, with the exception of the message system, not by the author). These include:

- authBSP

  TRIPOS has a virtual device called "BSP:" which can be used as part of a stream name. The resulting stream will be a BSP to some service or another on the Ring. A similar device called "authBSP:" has been implemented which uses authenticated BSP - putting the current user's UID set in the open block#.

  Services which are supported by TRIPOS and which use authBSP currently accept only 'user' UID sets (that is, those issued by Userauth). They include GIVEFILE and TAKEFILE for transferring files to or from a remote machine, and STAR for connecting into one.

---

* See the article called "Privilege Management".

\# See the article called "Authenticated Communication".

AuthBSP: has the advantage in the former cases that normal access to the user's file space will be allowed and in the latter that no password check will be necessary.

- the message system

  PUIDs are used to identify users and to distinguish their message files. It will be possible, in the future, to implement access controls on the messages that are read and so forth. However, because of the unprotected nature of TRIPOS, it will be impossible to stop users going "behind" standard utilities and avoiding these controls. This problem can only be solved by distributing the message system to another protection domain.

- the filing system

  The PUID naming the logged on user is stored in each file created so that it is possible to find the creator of any file. As in the message system access controls may be implemented based upon these identifiers although the same limitations apply (in the traditional TRIPOS filing system).

Article 17

SUMMARY AND CONCLUSIONS

In this thesis the meaning, fabrication and use of an object representation system have been discussed. This article attempts to underline the most important ideas scattered through the dissertation, under the headings Theory, Implementation, and Distributed Systems which were mentioned in the introduction.

17.1 Theory

Underlying any access control mechanisms there must be some kind of authentication system so that accessing, and perhaps accessed, objects can be reliably identified. A tool of great utility in this respect is a representation system, the only function of which is to provide a vehicle for such authentications. The ability to verify the identity of an object allows both access list and capability kinds of control of accesses to services.

In a distributed computing environment an object representation system can be fabricated from three main constituents:

- Naming System

  A naming system is necessary to label the objects being represented.

- Tokens

  Unique unforgeable tokens are needed as private indications of the possession of an object representation.

- A Relation

  A relation which associates tokens with names is needed to indicate that particular tokens are representations of objects with particular names.

This relation should not be permanent or unalterable because new object representations may be necessary at any time and, since the safety of the

secret token diminishes with time, because representations may need to be subsequently deleted (particularly if the secret token is lost to a thief). Because the tokens are to be passed to computing elements, the nature of which cannot be forecast and control of which cannot be guaranteed, they can only be made difficult to obtain illegally by making them difficult to guess. By randomly taking them from a large name space this can be made arbitrarily difficult. The relation, in a form such as a table, can be kept within a number of trusted computing elements.

Since authenticity is not an absolute quantity, a token should refer not only to the name of the object for which it is a representation but also to the name of the object that guarantees that this is so. This name, called an authentity, can be authenticated through the use of its own representation when it is used to create a new representation. Since this recursion (the creation of a representation requires another representation to exist in advance) must stop, there is a requirement for something capable of creating all possible representations without the use of an existing representation. This thing was called SOAP. It needs to be used only once in the history of a representation system to create the first representation.

The creator of a representation could pretend to be any of the names for which it generates representations and could, therefore, do anything that objects with those names could do. By giving these representations away power is effectively delegated to the new recipients. The initial representation is, therefore, the most powerful on the network and is consequently called the SOAP name.

Because a token refers to both a name and an authentity, the full name for a represented object should contain both of these parts. Using just the object's name may cause ambiguity since it is possible for two different authentities to use the same name for different objects. Both the most general format and the usefulness of a full name in the naming scheme (resulting from different implementations) were examined. If there is only one authentity the resulting widespread distribution of its representation makes the system rather fragile. If an authentity is specified using exactly one name part this problem disappears. If there is any fixed number of name parts in an authentity the scheme is equivalent to the one in which there is exactly one, except for the size of the namespace created. A variable number of name parts in an authentity creates a very flexible system in which the history of names can be exactly specified but which are unwieldy in use and can require the use of generalizing mechanisms to reduce

their precision.

A service holding a number of access control lists was used to create privilege representations which could be used at services as capabilities for the use of "sensitive" entries. Thus, services have the options of either implementing their own access control list mechanisms or using this primitive capability control mechanism (over which they have less control).

Four ways in which the identity of an object could be verified were proposed and a model was developed in which authentication was viewed as a mapping between one representation domain and another. Authentication takes one proof of the identity of an object and creates another proof of identity for the same object. Such a process was found to happen at more than one level in a distributed system. At the lowest level an object is represented within a program by some set of (possibly unforgeable) bits. This representation could have been generated by the authentication of a network representation which, in turn, could have been generated through the authentication of a person in whatever representation is most convenient. This model was applied to both users and services in designing authentication mechanisms for them.

17.2 Implementation

Although very rapid use (memory access speeds, for example) of the service containing the representation relation (the AOT service) was not expected there were practical reasons to make checking a representation cheap, not the least of which is that autonomous computing elements will not choose to use expensive forms of access control. For this reason a simple, fast protocol was used and the software for the representation verifying operation was optimized. The protocol had no error correction in it so each entry to the AOT service was carefully designed to be idempotent in so far as each one could be repeated a reasonable number of times with no overall detrimental effect. In order that the load on the AOT service could be shared, several were allowed, each holding an independent subset of the total number of representations. The random number generator used to create tokens needed special consideration since the current token should not enable the succeeding tokens to be guessed. Because two independent 48 bit random quantities are needed and the random number generator algorithm must be assumed to be known, a 128 bit multiplicative pseudo-random number generator was chosen. Only the top 96 bits after each application were

used, leaving a 32 bit number to be guessed. This task was made a little more difficult by periodically incrementing the seed.

Other table based services were implemented for a Privilege Manager, User Authenticator and SOAP Server, each with idempotent interfaces. They use the representation system to authenticate their own update.

A high level interface to the various servers was implemented on the TRIPOS operating system, including a local repository for representations, called a fridge. The User Authenticator is used to create a representation for the user, and it is stored in the fridge. Subsequent uses of the commands to access the Privilege Manager or the SOAP Server result in additional representations to be stored. A command to manipulate the fridge interactively using the AOT service was also described.

17.3 Distributed Systems

Many uses were found for representations in a distributed system. Stretching the imagination to different degrees, mechanisms have been given for providing representations for:

- Users - in the user authentication service.
- Computers - in the resource management example.
- Services - after their authentication.
- Entries - (a privilege could be considered to be a representation for the entry to a service that it affords access to).
- Addresses - in two-way authentication.
- Representations - TPUIDs are representation representations.

When dynamic services are used it was shown that, even when all communications are guaranteed to be private, unexpected communication paths can be set up which may be untrustworthy. A two-way authentication protocol, using address representations, was devised to circumvent this problem. Using it one service can prove its identity to another without risking the loss of any important or valuable representations (which is not the case in the simple one-way scheme, in which a client simply sends its identity to a server).

This protocol relies upon the service being able to obtain an authentity representation. The SOAP Server (as distinct from SOAP itself) is set up to grant these representations to chosen recipients in exactly the same way that the Privilege Manager generates privilege representations. Thus, given that the dynamic service has a representation proving its own identity, an entry in the SOAP service can be set up to allow it to claim the correct authentity representation.

The means by which a dynamic service could obtain its own representation were discussed and the conclusion made that it could, essentially, be loaded along with the code when the service was created. This, of course, assumes that the booting service can, itself, generate service identity representations. Thus the problem is recursive. Once the first operational boot server is fully operational the mechanisms provided will support themselves. A service for authenticating static services, by password, is provided which can be used by the first boot server (which must, therefore, be a static service). Thus, since they could communicate freely using two-way authentication, there was no remaining reason why dynamic services could not operate as well as static ones on a network with these facilities.

GLOSSARY

Address

The specification of a location (at which a single service can be contacted) on a network.

A\N

Used as a general specification of an arbitrary two-part name with name N under authentity A.

AOT

Active Object Table

A table in which tuples of the form <TPUID, TUID, AUTY, PUID, timeout> are kept: each tuple representing an active object with the name given by its PUID. Such a tuple was created by AUTY, is referred to by TPUID and will exist for TIMEOUT seconds. TUID is the representation of the active object.

AOT Service

A service responsible for maintaining an interface to an AOT. This interface includes the entries: VERIFY, GETTUID, REFRESH, IDENTIFY, and ENHANCE (q.v.).

Authentic

A representation is authentic if t can be provided for which t V p(A\N) where A\N is trusted.

AuthBSP

Authenticated BSP

An enhancement of the BSP in which the identity of principal of a communication is established, using a TUID, PUID and authentity, as part of the opening dialogue.

Authentication

Authentication is a mapping between one domain of representation (possibly null) to another (possibly the same) in which the object being represented remains constant.

Authentication, External

Authentication of an object external to a network (for example, another network or a user) through its representation in that outer domain yielding a network representation (cf. Internal Authentication).

Authentication, Internal

Authentication of a network representation yielding a representation internal to the authenticating service (for example, a hardware address). (cf. External Authentication).

AUTY

Authentity

The authentity of a representation is the identity of the authority under which that representation was created.

Authentity Representation

A representation for the identity of an authority. The possession of such a representation authorizes the creation of new representations.

Authority

An authority can potentially create representations for objects with its identity as their authentity. Its representation is distinguished by the authentity "auth".

Autonomy

The more autonomous that two individual systems (co-resident on the same network) are, the smaller the number of identical rules and mechanisms each system must share.

Auth

"Auth" has been arbitrarily chosen as the name of the authentity that authentity representations must have.

BISCM

Best Inter-Service Communication Method. That is, the method of communication used between services chosen to give the required level of privacy. The probability that the data addressed only to one recipient will either not arrive and/or will arrive elsewhere is known and accepted as "good enough". This probability defines the maximum privacy to be normally offered by the network.

Boot

A service is booted when it is dynamically created from its component parts. Typical component parts in this context might be a body of program code and something capable of executing that code. A service can be authenticated by this operation, and its representation included with its code upon loading.

Boot Server

A service responsible for booting and authenticating new services.

BSP

Byte Stream Protocol

The name of a real protocol used to transfer bytes from one service to another at a speed determined by both ends of the transfer (see also SSP).

Client

A service's client is some entity that makes use of that service. A client is not necessarily a person (cf. user).

Dynamic Service

A service which is not continually resident at any single address (cf. static service).

ENHANCE

An entry to an AOT service which enables an authority to make a TUID represent more than one object simultaneously. Alternatively it can be used to create representations with compound authentities (cf. GETTUID).

Entry

The name of the unit from which the network interface to a service is constructed. Each entry to a service activates a different function of that service.

Fridge

A representation container. Network representations may need continual maintenance if they are not to become invalid. This maintenance is conveniently provided in a fridge.

GETTUID

The entry to an AOT service which enables an authority to create new representations (cf. ENHANCE).

i

i(t,A\N)

The proposition i(t,A\N) implies that t represents the object created by A called N, or more succinctly t V p(A\N). This proposition about UID sets can be checked using VERIFY.

Idempotent

A protocol exchange is idempotent if it will produce the same result no matter how many times it is repeated.

IDENTIFY

An entry to an AOT service which verifies that a given TPUID refers to a specific representation (given by its TUID, PUID and AUTY). (cf. VERIFY).

Independent Service

A service is independent of all other services iff information transfer between it and any other service is always achieved through communication on the network.

Kernel

A set of services (both static and dynamic) which are responsible for the management of a network's resources and for the provision of trusted network primitives. Kernels are independent - several may coexist on the same network.

Monitor

The monitor for a representation is the possessor of the ability to alter the period of existence of that representation.

Multi-level Name

A name from a naming system with a fixed number of hierarchical name parts. Except for the size of the name space a multi-level naming scheme shares the properties of a two-part naming scheme (cf. multi-part name).

Multi-part Name

A name from a naming system with a variable number of hierarchical name parts. If the name has no parts at all it represents SOAP otherwise the name can be divided into two main sections - the last name part which is the named object's PUID, and the other name parts which constitute its authentity: which is itself a multi-part name (cf. two-part name).

Name

An item of information which has no meaning other than that it denotes some particular object.

Nameserver

A static service responsible for mapping textual names onto physical addresses within a Cambridge Ring. Services should be addressed by their textual names on a Cambridge Ring - hence it is a member of the kernel of a Ring system.

Name-part

A single element of a name with many parts. Within the name the name-parts to its left (which constitute its authentity) specify the naming domain in which this name-part has meaning.

Network

A network is a collection of independent communicating services.

N-level Name

A multi-level name.

Object

A reference to an "object" can be consistently replaced by a reference to anything sufficiently integral to warrant a name of its own.

OPRT

One Pass Representation Table

The analogue of an AOT for pass-once representations. In addition to the information kept in AOT tuples a OPRT holds the names of its owners and of its monitors.

p

p(A\N)

The proposition p(A\N) implies that the authority A claims that an object called N exists. Whatever proves this proposition can be used as a representation for that object.

Pass-once Representation

A representation which has the property that it may be possessed only by a fixed number of owners. Having been passed from one owner to the next such a representation will cease to be held by the former (this is not a property of representations served by the AOT server).

Principal

The principal of any activity is the object on whose behalf it is being undertaken. If accounting were used it would be the principal that would be charged for the activity.

Privilege

Privileges are representations created with the authentity PRIVILEGE. They form the currency that buys the use of protected entries in services.

Privman

Privilege Manager

A server which maintains lists of objects (called virtues) able to claim particular privileges. It provides privileges to clients quoting correct representations of listed virtues.

PUID

Permanent Unique Identifier

In an AOT tuple the PUID is the name of the object being represented. Otherwise (in the sense that an authentity is a PUID) a PUID is a particular implementation of a name part (cf. TUID).

R

R is the abstract relation that an AOT explicitly supports in order to provide representations. $R = \{<d,t,A,N,\triangle>: \triangle > 0 \ \& \ d \ V \ i(t,A\backslash N)\}$ in which $\triangle$ is the length of time for which a given tuple is currently to exist.

RATS

Remotely Activated Terminal Session

The name of the protocol used on the Cambridge Ring for terminal traffic. It is founded on BSP.

REFRESH

An entry to an AOT service which enables the possessor of a TPUID for an object's representation to change that representation's timeout. Changing the timeout can result in the representation's deletion or its existence beyond its current life expectancy.

Representation

T is a representation for object N under A's authority if it can be used to prove that A created it for N. That is if, t V p(A\N).

Server

Servers are services which provide some object or perform some job for a user. Resource allocators, printer servers, authenticators, and compiler servers are all valid examples (cf. service).

Service

A Service is the name given to an independent communicating entity on a network. It could, for example, be implemented as a separate computer, a process within a computer, a coroutine within a process or just a module of event driven code executing somewhere (cf. server).

SOAP

Source Of All Power

All authority on a network is initially delegated from SOAP. It is the only thing on the network able to create a representation for the SOAP name without first possessing one. It is controlled by levels of authority outside the network.

SOAP Name

A representation for the SOAP name enables a representation for any other name to be created. It is, therefore, the most powerful name used in the representation system. In the AOT representation system this name would be "auth\auth".

SOAP Service

A server provided by SOAP which delegates SOAP's authority on its behalf. On the Ring it provides authentity representations in exactly the same way that the privilege manager provides privilege representations.

SSP

Single Shot Protocol

A simple protocol in which a request (of small maximum size) is sent to a server and small reply is sent in return. The protocol requires that the request must be idempotent since either the request or the reply may legitimately be lost (see also BSP).

Static Service

A service is static if its address changes only very occasionally (or not at all). The address of such a service can reliably be found by using the nameserver (cf. dynamic service).

T-authenticator

An authenticator service for the type T. Such a service specifically deals with objects of type T (for example, users, services, addresses and so on) and possesses a method to verify the identity such objects.

TPUID

In an AOT tuple the TPUID is the representation of that tuple. It can be used to either delete or maintain that tuple and distinguishes the owner of the tuple from users of it.

TUID

Temporary Unique Identifier

In an AOT tuple the TUID is the representation that that tuple implements. It is the TUID of the tuple that is parsed around the network to unforgeably denote the thing being represented by that tuple. Otherwise (in the sense that a TPUID is a TUID) a TPUID is a particular implementation of a representation name (cf. PUID).

Two-part Name

A name consisting of both an authentity and a PUID as the specification of the object to which it refers (cf. multi-part and multi-level name).

UID

Unique Identifier

The object used for the implementation of both a PUID and a TUID. UIDs are created only once and never repeated.

UID set

The generic name given for the main components of an AOT tuple, a TPUID, TUID, PUID and authentity. It is a loose term which can be used to denote a subset of these components (such as just the TUID or TUID and TPUID) when appropriate.

User

A service's user is some person that makes use of that service (cf. client). Although the gender of the word "user" is taken to be masculine the actual gender of a user is not assumed.

Userauth

User Authenticator

A server which maintains lists of users and their passwords. It provides representations under its own authentity, 'user', for user's who can quote their password correctly.

V

In x V y (t V p(A\N) or d V i(t,A\N) for example), V means "proves" or "can be used to verify that". It implies that a verifiable relationship exists between x and the proposition y.

VERIFY

An entry to an AOT service which verifies that a given TUID is a valid representation of a given object (specified with a PUID and authentity). (cf. IDENTIFY).

Virtue

A name of an object the representation for which allows a privilege to be claimed from the Privilege Manager.

REFERENCES

[Davies81]
D. W. Davies,
"Protection", Chapter 9, "Distributed Systems - Architecture and Implementation: An Advanced Course", Edited by B. W. Lampson, M. Paul and H. J. Siegert, pp. 221-245, Lecture Notes in Computer Science No. 105, Edited by G. Goos and J. H. Hartmanis, 1981

[Denning79]
D. E. Denning and P. J. Denning,
"Data Security", Computing Surveys Vol. 11 No. 3 pp. 227-249, September 1979

[Dennis66]
J. B. Dennis and E. C. van Horn,
"Programming semantics for multiprogrammed computers", Communications of the ACM Vol. 9 No. 3 pp. 143-155, March 1966

[Dion81]
J. Dion,
"Reliable Storage in a Local Network", University of Cambridge Computer Laboratory, Ph.D. Thesis, February 1980

[Evans74]
A. Evans Jr., W. Kantrowitz, E. Weiss,
"A User Authentication Scheme not Requiring Secrecy in the Computer", Communications of the ACM Vol. 17 No. 8 pp. 437-442, August 1974

[Fabry74]
R. S. Fabry,
"Capability Based Addressing", Communications of the ACM Vol. 17 No. 7 pp. 403-412, July 1974

[Gibbons80a]
J. J. Gibbons,
"The Design of Interfaces for The Cambridge Ring", Ph.D. Thesis, University of Cambridge Computer Laboratory, September 1980

[Gibbons80b]
J. J. Gibbons,
"SSP - A Single-shot Protocol for the Ring", Internal document, University of Cambridge Computer Laboratory, September 1980

[Girling81]
C. G. Girling,
"The Use of UID Sets", Internal document, University of Cambridge Computer Laboratory, September 1981

[Girling82]
C. G. Girling,
"Object Representation on a Heterogeneous Network", Operating Systems Review Vol. 16 No. 4 pp. 49-59, October 1982

[Graham72]
G. S. Graham and P. J. Denning,
"Protection - Principles and Practice", Proceedings of the AFIPS Spring Joint Computer Conference Vol. 40 pp. 417-430, AFIPS press, 1972

[Hopper79]
A. Hopper,
"Local Area Computer Communication Networks", Ph.D. Thesis, University of Cambridge Computer Laboratory, April 1978

[Johnson79]
M. A. Johnson,
"Ring byte stream protocol specification", Internal document, University of Cambridge Computer Laboratory, April 1980

[Johnson80a]
M. A. Johnson,
"MSF clock and Logger", Internal document, University of Cambridge Computer Laboratory, January 1982

[Johnson80b]
M. A. Johnson,
"Ring Authentication on CAP", Internal document, University of Cambridge Computer Laboratory, December 1980

[JNT80]
The JTP Working Party of the Data Communications Protocol Unit,
"A Network Independent Job Transfer and Manipulation Protocol", April 1980

[JNT82]
Joint Network Team of the Computer Board and Research Councils,
"Cambridge Ring 82 Protocol Specifications", November 1982

[Jones78a]
A. K. Jones,
"The Object Model", "Operating Systems: and Advanced Course" pp. 8-16, Edited by R. Bayer, R. M. Graham and G. Siegmuller, Lecture notes in Computer Science No. 60, Edited by G. Goos and J. H. Hartmanis, 1980

[Jones78b]
A. K. Jones,
"Protection Mechanisms and the Enforcement of Security Policies", "Operating Systems: and Advanced Course" Chapter 3.C pp. 228-251, Edited by R. Bayer, R. M. Graham and G. Siegmuller, Lecture Notes in Computer Science No. 60, Edited by G. Goos and J. H. Hartmanis, 1980

[Kahn72]
R. E. Kahn,
"Resource Sharing Computer Communications Networks", Proceedings of the IEEE Vol. 60 No. 11 pp. 1397-1407, November 1972

[Karger77]
P. A. Karger,
"Non-discretionary Access Control for Decentralized Computing Systems", Report numbers: MIT/LCS/TR-179 and ESD-TR-77-142, S.M. Thesis, Massachusetts Institute of Technology, May 1977

[Lampson69]
B. W. Lampson,
"Dynamic Protection Structures", Proceedings of the AFIPS Fall Joint Computer Conference Vol. 35 pp. 27-38, AFIPS Press, 1969

[Lampson71]
B. W. Lampson,
"Protection", Proceedings of the Fifth Symposium on Information Sciences and Systems pp. 437-443, at Princeton University, 1971, Reprinted in Operating Systems Review Vol. 8 No. 1 pp. 18-24, 1974

[Knight82]
B. J. Knight,
"Portable System Software for Personal Computers on a Network", Ph.D. Thesis, University of Cambridge Computer Laboratory, February 1982

[Morris79]
R. Morris and K. Thompson,
"Password Security: a Case History", Communications of the ACM Vol. 22 No. 11 pp. 594-597, November 1979

[Needham78a]
R. M. Needham,
"User-Server Distributed Computing", "Distributed Computing Systems", Proceedings of the joint IBM / University of Newcastle upon Tyne Seminar pp. 71-78, Edited by B. Shaw, University of Newcastle upon Tyne, 5 - 8 September 1978

[Needham78b]
R. M. Needham and M. D. Schroeder,
"Using Encryption for Authentication in Large Networks of Computers", Communications of the ACM Vol. 21 No. 12 pp. 993-999, December 1978

[Needham79]
R. M. Needham,
"Adding Capabilities Access to Conventional File Services", Operating Systems Review Vol. 13 No. 1 pp. 3-4, January 1979

[Needham79]
R. M. Needham and A. J. Herbert,
"The Cambridge Distributed Computing System", International Computer Science Series, Edited by D. McGettrick and J. van Leeuwen, Addison-Wesley Publishing Company, 1982

[Newman82]
W. B. Newman,
"Design of an Encryption System for Project UNIVERSE", Proceedings of the 6th International Conference on Computer Communication pp. 384-389, Edited by M. B. Williams, North-Holland Publishing Company, September 1982

[Ody79]
N. J. Ody,
"Initial loading and debugging or the Type 1 Z80 system", Internal document, University of Cambridge Computer Laboratory, May 1979

[Ody80]
N. J. Ody,
"The implementation of a Terminal Concentrator on a Z80", University of Cambridge Computer Laboratory, Internal document, January 1980

[Peterson79]
J. L. Peterson,
"Notes on a Workshop on Distributed Computing", Operating Systems Review Vol. 13 No. 3 pp. 18-27, July 1979

[Popek79]
J. G. Popek and C. S. Kline,
"Encryption and Secure Computer Networks", Computing Surveys Vol. 11 No. 4 pp. 331-356, December 1979

[Purdy74]
G. B. Purdy,
"A High Security Log-in Procedure", Communications of the ACM Vol. 17 No. 8 pp. 442-445, August 1974

[Redell74]
D. R. Redell and R. S. Fabry,
"Selective Revocation of Capabilities", IRIA International Workshop on Protection in Operating Systems pp. 197-210, France, 1974

[Saltzer75]
J. H. Saltzer and M. D. Shroeder,
"The Protection of Information in Computer Systems", Proceedings of the IEEE Vol. 69 No. 9 pp. 1278-1308, September 1975

[Saltzer78]
J. H. Saltzer,
"On Digital Signatures", ACM Operating Systems Review Vol. 12 No. 2 pp. 12-14, April 1978

[Walker79]
R. D. H. Walker,
"Basic Ring Transport Protocol", Internal document, University of Cambridge Computer Laboratory, October 1978

[Weissman69]
C. Weissman,
"Security Controls in the ADEPT-50 Time-sharing System", Proceedings of the AFIPS Fall Joint Computer Conference Vol. 35 pp. 119-133, AFIPS Press, 1969

[Wilkes68]
M. V. Wilkes,
"Time Sharing Computer Systems", Macdonald and Jane's / American Elsevier Computer Monographs No. 5, Edited by S. Gill and J. J. Florentin, 1968

[Wilkes75]
M. V. Wilkes,
"Communication Using a Digital Ring", Proceedings of the PACNET Conference pp. 47-55, Sendai, Japan, 1975

[Wilkes79]
M. V. Wilkes and D. J. Wheeler,
"The Cambridge Digital Communication Ring", Local Area Communications Network Symposium pp. 47-61, Boston, (Sponsored by MITRE Company and the National Bureau of Standards), May 1979

[Wilkes80]
M. V. Wilkes and R. M. Needham,
"The Cambridge Model Distributed System", Operating Systems Review Vol. 14 No. 1 pp. 21-29, January 1980

[Yellowbook80]
Post Office PSS User Forum,
"A Network Independent Transport Service", Prepared by Study Group Three, February 1980

APPENDIX 1

AUTHENTICATED BSP OPEN BLOCK

The format of an open block containing an AUTHBSP open, therefore, might be as follows (note that other valid TSBSP messages could precede or succeed the authentication message):

block offset 0: m.s. byte - the bit pattern 01101010
l.s. byte - flags: with at least bit d1 set. In general this will be the bit pattern 00000010.
1: port number to be used for the reply
2: the function number part of the ring service address
3: the number of BSP parameter words (2)
4: maximum block size prepared to receive
5: maximum block size that may be sent
6: start of user data:
m.s. byte - message type (64)
l.s. byte - number of authentication octets (24) + 128 to denote last fragment of parameter
7-10: TUID (proving identity of name)
11-14: PUID (representing name of sender)
15-18: AUTY (the name of the verifying authority)
19: m.s.byte - zero

Implementations of BSP which wish to provide an AuthBSP service recognize the above TSBSP authentication message in open blocks do not necessarily provide full versions of TSBSP.