An Open Architecture for Secure Interworking Services

Richard Hayton

Fitzwilliam College
University of Cambridge

A dissertation submitted for the degree of
Doctor of Philosophy

Abstract

An emerging requirement is for applications and distributed services to cooperate or *inter-operate*. Mechanisms have been devised to hide the heterogeneity of the host operating systems and abstract the issues of distribution and object location. However, in order for systems to inter-operate *securely* there must also be mechanisms to hide differences in security policy, or at least negotiate between them.

This would suggest that a uniform model of access control is required. Such a model must be extremely flexible with respect to the specification of policy, as different applications have radically different needs. In a widely distributed environment this situation is exacerbated by the differing requirements of different organisations, and in an *open* environment there is a need to interwork with organisations using alternative security mechanisms.

Other proposals for the interworking of security mechanisms have concentrated on the enforcement of access policy, and neglected the concerns of freedom of expression of this policy. For example it is common to associate each request with a user identity, and to use this as the only parameter when performing access control. This work describes an architectural approach to security. By reconsidering the role of the client and the server, we may reformulate access control issues in terms of client naming.

We think of a client as obtaining a name issued by a service; either based on credentials already held by the client, or by delegation from another client. A grammar has been devised that allows the conditions under which a client may assume a name to be specified, and the conditions under which use of the name will be revoked. This allows complex security policies to be specified that define how clients of a service may interact with each other (through election, delegation and revocation), how clients interact with a service (by invoking operations or receiving events) and how clients and services may inter-operate. (For example, a client of a Login service may become a client of a file service.)

This approach allows great flexibility when integrating a number of services, and reduces the mismatch of policies common in heterogeneous systems. A flexible security definition is meaningless if not backed by a robust and efficient implementation. In this thesis we present a systems architecture that can be implemented efficiently, but that allows individual services to 'fine tune' the trade-offs between security, efficiency and freedom of policy expression. The architecture is inherently distributed and scalable, and includes mechanisms for rapid and selective revocation of privileges which may cascade between services and organisations.

To my wife, Maria

What's in a name? that which we call a rose
By any other name would smell as sweet.

Romeo and Juliet II.ii.43

Preface

I would like to thank my supervisor, Ken Moody, for his invaluable support, encouragement and constructive criticism. I would also like to thank Jean Bacon, and other members of the Opera group for their valuable advice and helpful discussions. I appreciate the assistance provided by Ken Moody, Jean Bacon, John Bates, Ralph Becket, Sai Lai Lo, Oliver Seidel and Mark Spiteri who suggested improvements to the dissertation.

I would like to thank my sister, Karen, for drawing the palm tree on the front cover, and my wife and parents, for their moral and financial support.

This work was supported by a studentship from the Engineering and Physical Sciences Research Council.

Except where otherwise stated in the text, this dissertation is the result of my own work and is not the outcome of work done in collaboration.

This dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university.

No part of this dissertation has already been, or is being currently submitted for any such degree, diploma or other qualification.

This dissertation does not exceed sixty thousand words, including tables, footnotes and bibliography.

Contents

List of Figures

Glossary

Roles

certificate A signed statement issued by a particular service. Also used as an abbreviation for *role membership certificate*.

credential An assertion of simple fact (for example 'Fred is logged on') or an assertion of a boolean combination of these facts.

credential record A record representing knowledge about the value of a *credential*. Credential records are linked so that a change in the value of one credential may effect others.

domain See *protection domain*.

election A generalised form of delegation, whereby a client who is a member of one *role* uses this *credential* in order to allow another client to enter a (possibly different) role.

entry conditions Conditions that must be satisfied in order for a client to be granted entry to a *role*.

membership rules *Entry conditions* that must remain true during the lifetime of a *certificate*. If such an entry condition becomes false, the certificate is revoked.

protection domain The smallest unit of naming for an Oasis client, generally equivalent to a process. In architectures where a process's privileges change when they invoke protected subroutines, the process is in a different protection domain for each set of privileges.

RDL Acronym for *role definition language*.

RMC Acronym for *role membership certificate*.

role A name assumed by a client, representing the authority they are acting under. In Oasis each process is associated with one or more roles. In other models each user is associated with a role.

role definition language A language used to specify the relationships between *roles*, and the *entry conditions* and *membership rules* for each.

role membership certificate A process specific capability entitling that process to act under the authority of a *role*.

The MSSA

bypassing The act of optimising an access path through a series of *value adding custodes*, by missing out one or more of them.

container A logical grouping of files within a *custode*, used for management and accounting purposes.

custode A storage server that provides a particular storage interface.

Shared acls Access control lists that are explicitly shared between more than one file.

value adding custode A *custode* that is implemented by abstracting the interface of an existing custode, and then providing some additional functionality.

Events

active database A database that generates events when its state changes.

admission control Access control checks during client *registration*.

base events Events that are not *composite*.

composite event An event expression in terms of a number of *base events*. For example `A` followed by `B`. A composite event is signalled when the expression is satisfied.

event A named, parametrised occurrence signalled by an event server to clients who have indicated that they wish to receive it.

event horizon time stamp A lower bound on the time stamps of events yet to be signalled by a server.

event template An *event* specification, possibly including wild card parameters.

generic event object A representation of an *event* or *event template* in a type and machine independent form.

global view Complete knowledge about all events generated by a system up to some point in time. The detection of composite events is considerably simplified, if processing is delayed until a global view is achieved.

heartbeat protocol A protocol whereby an event server periodically asserts its ability to transmit events. This is used to detect delays or failures.

independent evaluation The process of evaluating two or more expressions simultaneously, so that evaluation of one does not effect evaluation of the other. In particular network delays effecting one evaluation do not necessarily affect another.

interesting events Events that a client wishes to be informed of, or has indicated they wish to receive.

pre-registration An indication by a client that it may later wish to *retrospectively register* interest in an event occurrence.

registration The process whereby a client establishes a session with an event service, and indicates which event occurrences it wishes to be notified of.

retrospective registration Registration in event occurrences starting at some point in the past. Event occurrences between the registration time and the current time are signalled immediately.

Chapter 1

Introduction

1.1 Models for Access Control

The world is full of access control systems. We have keys to doors, guest lists at clubs, passports and driving licences. Early computerised access control systems were designed to mimic one or other of these “real world” systems. Capabilities are like keys. If you have a capability for an object, you may use it. Identity based capabilities are a refinement of these, and enable access by a particular client — rather like a passport. Access control lists are much more like membership lists — ‘if you’re not on the list, you can’t come in’.

An emerging and important issue for computer systems is how heterogeneous systems can interact. This is a problem the ‘real world’ has had for centuries. We have mechanisms and policies for using British driving licences in Germany, for becoming a member of a club and for providing references when renting a house. This work proposes an analogous architecture for computing systems, whereby security policies can interwork in a general and open way.

1.2 What’s in a Name?

In order to interwork between systems, we must find some common ground. Access control may be generalised in the following way. A client approaches a server with a request. The server must then decide whether or not to grant each request. To assist its decision making, the client will provide some credentials, and the server will consult some stored policy. In a capability system, the credential supplied is a capability, and the policy is ‘perform the request if the capability is valid’. In an access control list scheme,the credential supplied is the identity of the client, and the policy is determined by the contents of the ACL. The essence of these mechanisms is the same; the client supplies a *name* and the server consults a policy.

We can generalise this, and use it as a basis for interworking. The policy stored within a server need only concern itself with the rights associated with names. How a particular process obtains a name is a separate issue, and it is this issue that this work concentrates on. Moving the problem to the domain of naming has many advantages. Reasoning is often more natural in this domain. For example hiring and firing, club memberships and being elected are all concepts we are familiar with. We may also define policy about how names are acquired in a way that is *independent* of the services which ultimately make use of the names. For example the enforcement of the policy statement *users must be members of the university* should not affect the implementation of a file service for those users to use.

The ability to control access by controlling naming, is the proverbial extra level of indirection that allows us to solve all computing problems.[1] This can lead to a simpler, and more efficient, implementation than would otherwise be possible. The alternative is to use the same naming mechanism for all clients of all services. This is necessarily heavyweight. For example if all clients are represented by a user identity, then this identity must be validated in all circumstances. This would force a homogeneity between secure console logins, and insecure remote logins that may be unwarranted.

1.3 Research Motivation

In a widely distributed environment there will be many different organisations. If users and services in these organisations are to interwork with each other, there must be standard mechanisms for the specification and enforcement of access control policies. Traditional approaches to wide area security have enforced common semantics on all clients and services; for example by insisting that access policy is defined purely in terms of user identity, and that all users have a name in a global name space. This approach severely restricts the autonomy of the organisations, and their ability to control their own security policy.

1.4 Research Statement

This work proposes a new, open, security architecture: Oasis. Oasis is different from other proposals, in that the emphasis is the specification of policies for client naming. The aim is to produce a clean and simple architecture that allows all policy decisions to be made explicit, and hence aid reasoning about the secure interworking of heterogeneous systems. As the issue of naming is separated from the functional requirements of the different services, it is believed that a general and efficient mechanism can be devised for the enforcement of these policies.

1.5 Outline of Dissertation

Chapter 2 considers the issues of client naming. It considers the arguments for and against a global name space for all clients, and proposes a novel role based model designed to capture the advantages of both approaches.

Chapter 3 considers policy definition itself. A role definition language is derived to allow both static access control rules and interworking to be defined in a clear and unambiguous way.

Chapter 4 outlines significant details of the system architecture. In particular the mechanisms for allocating, checking and revoking capabilities are considered.

[1]This remark has been attributed to Prof. David Wheeler.

Chapter 5 gives a detailed case study. The Oasis architecture has been applied to a complex distributed storage system, the MSSA[Lo94]. In the context of this architecture, the issues of access control lists and meta-access control are considered.

Chapter 6 describes a new paradigm for distributed programming to which Oasis has been applied: Event Management. The issues of event specification, registration and notification are discussed, together with the design of a distributed composite event detection scheme. An active badge system is given as a case study of event programming techniques.

Chapter 7 considers the issues of access control specification and implementation for distributed event management. The active badge example of the previous chapter is used to illustrate the issues, and some solutions.

Chapter 8 summarises the research results and suggests further work.

Chapter 2

Naming

2.1 Introduction

In this chapter we discuss naming issues for distributed services, and for clients of these services. In particular we concentrate on *open*, *distributed* systems, where an 'end client' may communicate with a service provider via several intermediate services, and where the client and service providers may exist in different administrative domains. In the following sections we derive the need for multiple inter-working naming schemes defined on a per service basis. In sections 2.5 and 2.6 we consider related work and emerging areas that a new security model must be able to meet. Sections 2.7 to 2.10 define the Oasis two level naming scheme and section 2.11 summarises the issues raised in this chapter.

2.2 Open Systems

When security systems were first designed for computer systems, it was in the context of single multi-user systems. The designers considered what was to be protected (generally files) and who the clients were to be (processes). An appropriate name space was designed that allowed authorisation policy for access to these resources to be specified in terms of the 'significant attributes' of the clients. In most cases it was the identity of the user responsible for the process that was chosen.

Modern computing systems are radically different. The diversity and proliferation of services can lead to considerable problems when attempts are made to express authorisation policy purely in terms of user identity. In addition, the trust relationship between services has become complex. In single multi-user systems, all protected services were provided as part of the operating system, and could be trusted to correctly interact without the need for formal authorisation. In today's systems, services are both distributed and distrustful of each other. A request from one service to another must be treated as a client request like any other, and must be authenticated and authorised. Typically a server-server request will be made on behalf of an 'end' client. It may be the intermediate server identity, the end client identity or some combination of both that is most appropriate for use in the authorisation test.

This problem is further compounded by the move to 'Open' systems, where services and clients interact over organisational boundaries. Such systems would tend to imply a global name space is required for clients, in order that clients in one domain may be named within another. As will be illustrated, such a scheme would not only be complex, but is not generally necessary.

We believe that a change of focus is required, from clients to services. If we reconsider the basis of client-server interaction, it is clear that the server must be able to determine the 'significant attributes' of the client in order to determine if the client is authorised to make the request.[1] The user identity associated with a client is only one attribute that might be considered significant. We argue that in a distributed environment there must be many inter-linked client name spaces. Indeed, in general, each Oasis service is responsible both for naming and authenticating its clients.

2.3 Naming Clients

A service must be able to identify its clients, and distinguish between them. A simple, and widely used approach is to use some attribute of client identity that is already securely available. Use of a single naming scheme has clear advantages. Most important is simplicity. No system is truly secure if those with the authority to control policy do not understand the full implication of their actions, and *all* users control policy to some degree.[2]

As systems increase in scale and complexity, so do the possible dependencies and contradictions implicit in the policy definitions for different services. An emerging area is automated reasoning about policy definitions in order to discover loopholes or contradictions[LYS95]. Such reasoning (automated or otherwise) is considerably simplified if the same mechanisms are used throughout.

From the user's (or organisation's) point of view, it is clear that a single name space for all services is advantageous, if not essential. In the rest of this section we will consider security from the *server's* point of view. As will be seen, a single name space causes considerable problems.

2.3.1 Discrimination

In systems with a fixed name space, a service may be unable to make useful distinctions between clients. For example, if a user-identity is chosen for naming, then a service cannot distinguish between different processes owned by the same user. In particular, access control cannot be based on the identity of the program text being executed, or the machine the process is executing on. Some centralised systems [MO87] make these attributes available for use by system services, such as filing systems, but extending these schemes to a heterogeneous distributed environment is difficult. In particular the semantics of a process identity vary across operating systems. In Unix, for example, processes are protected from each other, whereas in a Microsoft Windows environment one

[1] We agree with [BGS92, Vin88] that the service must make this decision, not some centralised authority.

[2] For example, a manager may give his secretary his password, if he knows of no alternative delegation mechanism.

'process' could conceivable modify another whilst it was running. In this environment it would be unwise to base access control on the program text a process was executing. The authentication services provided on different platforms also vary. In some it may be possible to authenticate the program text, in others it might not be, or might only be under certain circumstances (for example if the program resided on a particular device).

An approach that initially appears favourable, is to 'bundle' all relevant naming information together; for example user-identity, program-identity and host machine. However such a name is complex, and troublesome to authenticate. Unlike user-identity, such a name has unclear semantics, and it is unlikely that systems developed separately will use similar naming schemes for program and machine identity. In addition, future systems may require that additional attributes are included in such a name, for example the identity of the shared library currently being executed, or the nature of the interconnecting network. Such an approach is clearly unmanageable, and we therefore reject it.

2.3.2 Wide Area Issues

It is unlikely that in future we will have the luxury of a single global name space for all possible clients of all possible services, or that distinct name spaces in different domains will have the same semantics, allowing identification by a $(domain, id)$ tuple. We must then choose between closed systems, and an approach that allows inter-working between naming schemes. Two approaches are to allocate clients multiple identities, with one identity in each administrative domain, or to allow services to interact with clients using multiple name spaces. The latter approach is to be preferred, as the distinction between local and remote clients is generally a useful one, and the inconvenience of multiple logins is avoided.

2.3.3 Security Mismatch

When the clients of a service are taken from a wider domain than the local network, it is likely that the degree of trust will differ. For example, a client on a remote portable machine, who is connected by a radio link may be much more susceptible to eavesdropping than a user at a console terminal. In existing systems this is tackled by either enforcing the higher level of security by use of increasingly complex cryptographic techniques, or by accepting the lower level from all clients. When wide area access, and access from portables that are susceptible to theft is considered, the former choice may be difficult to implement, and the latter unacceptable. Instead, we must distinguish between clients with different levels of trust, and allow each application to act accordingly.

2.4 Server-centric Security

In the previous section we indicated that a service must be able to distinguish between requests from different clients and that the 'significant client attributes' used to make this distinction is a server specific issue. For example one service may need to distinguish between requests from different processes representing the same user, another may not.

In addition there may be reasons why different *mechanisms* are appropriate for different services. For example one service may require an access control list scheme, whilst another may prefer capabilities. If the services are to interoperate, a compromise is necessary.

There are many reasons why each service might wish to choose its own naming scheme for its clients and its own mechanisms for the expression and enforcement of authorisation policy. However, if services are to interact, common mechanisms must be used.

In Oasis there may be many client name spaces. A client will approach an Oasis server and request that it is granted the use of a name. It may then use this name when making requests to servers that understand it. Crucially, when a client requests a name, it may provide names from other name spaces, or issued by other servers, as credentials. This allows interaction between name spaces, and allows us to reason about interacting policies.

In addition, if a service requires access to client attributes not available in an existing name space, it may create a new name space. Typically a new name space is created by each service that has clients that might interact (perhaps through delegation), and by each service that must base access control on client attributes not available in some other name space. For example the fact that a particular client may access file `foo` is an attribute of the client known only to the file server.

A similar, although more restricted use of assuming names in order to increase (or restrict) access privileges is present in many experimental systems. In common with these systems, the term *role* is used for such a name. However unlike these schemes, in Oasis a pseudonym is assumed by a *process* not by a user. This removes the special status of the human user associated with a computing activity, and allows more general reasoning. In the following section a number of existing role-based systems are considered.

2.5 Related Work

Although the majority of existing systems rely on security architectures originally devised for centralised systems, several recent proposals have been targeted specifically at open distributed environments. ODP architectures, in particular, require the ability to *delegate* access rights, as invocations on one object typically involve nested accesses to other objects. In [BGS92] the authors argue that a centralised authority is unrealistic, and that a *server based* view of security is more appropriate. Their work proposes a system wide model based on capabilities, whereby the interface to a service consists of both an object name and a token allowing (some) client access to that object. The capability may be delegated, but a complete history of delegations is kept, and when the capability is used in a request, the server may apply whatever policy is appropriate. Their paper describes a scalable and open *mechanism* for enforcing security, but the issues of policy expression are not discussed. The mechanism itself has the traditional capability features of flexible delegation, but poor revocation.

Vinter [Vin88] proposes an 'extended discretionary access control scheme' for a closed, but distributed operating system. In this scheme roles are used for naming clients and a declarative policy expression mechanism is used. However the declarative policy statements are simply ACLs and the policy declaration does not cover client interaction. For example it is not possible to express "A manager may only delegate to his secretary".

Moffett [MST90] provides another mechanism for expressing access control in a distributed environment. In this paper, clients and objects are grouped into domains, and interaction between domains is expressed using *access rules*. This is a reasonably general mechanism, and although it is scalable, it relies on uniform naming mechanisms for all clients and servers.

Several proposals have concentrated on a formal approach to policy expression. Lampson [LABW93] proposes a calculus for expression of authorisation and client interaction. This formalism is accompanied by a complete implementation. However there is only a single naming scheme for clients, and although clients can extend their name using roles, these may only be used to *limit* authority, not to increase it. The system also suffers from 'over delegation', where more authority is delegated than is strictly required. Delegation is performed by a client allowing a proxy to use its identity. Although the identity may first be restricted, it is only possible to delegate the use of names. More general criteria, such as 'allow `X` to act as `Y` *but only when accessing object* `Z`' can only be specified by explicit inclusion in an ACL for `Z`. Lupu, Yialelis and Sloman [LYS95] concentrate on the formal modelling of organisational policies, with the aim of automated reasoning about these policies. They argue that policies must be modelled formally, in order that managers may query, negotiate and change policies. Although Oasis does not consider these issues in detail, it is designed in such a way that such a high level architecture could be implemented using it.

2.6 Emerging Areas

There are three emerging areas of research that are particularly in need of access control measures.

Open Distributed Programming There has been a great deal of recent work on the design of architectures to support open distributed programming. In particular, CORBA has emerged as a reference architecture adopted by many vendors in order to allow interworking between their products[Gro92]. The controlling body of CORBA, the OMG, recently published a white paper on the security requirements for CORBA[OMG94]. In relation to access control, this paper emphasizes the need for *flexibility* especially with respect to delegation (nested invocations), the physical location of an object (trust dependent on host) and the need to access object state in order to make security decisions. The general requirements given are for *consistency* and *scalability*. The document also stresses the point "If security is difficult to administer and use, it will not be set up correctly and users will try to bypass it ... ".

Cooperative Work This area has received renewed attention recently, as the emergence of open systems and fast networks have allowed cooperative work between people separated by large distances. It is often argued that access control requirements for cooperative work are more complex than those for the majority of other systems[GS86]. This is in part due to the dynamic nature of objects in a cooperative environment. If there is a need to protect and share objects which have rapidly changing security requirements, then it is unreasonable to expect a manager to make the changes manually. Consequently access control for cooperative work frequently requires access to the state, or attributes of the protected objects. For example, a requirement might be that access is controlled by the *last* person to make a modification to an object. A second requirement for cooperative work is that policies must be clearly defined, as they may be negotiated between a number of organisations, and are therefore harder to update. In particular an access control policy might be part of a larger policy determining the obligations and motivations of the various members[Slo94]. Recent work has suggested that a role based model is appropriate in order to aid policy specification for cooperative applications[CD94].

Event Management Access control is generally specified in terms of clients making requests to a service, which determines whether or not to grant each request. Event management is a new distributed programming paradigm that does not fit this model. In event based systems, a service notifies clients whenever 'significant events' occur. For example a network switch may generate events whenever it suffers from overload, and an ODP Trader[APM93] might generate an event whenever a new instance of a service is registered with it. When security policies are applied, a client may or may not be allowed to receive notification of particular instances of events. For example, in a Military environment, a client may only be told about new services registered with a Trader if that service has an *access class* lower than the client's[Den76]. The issues of access control for event based systems are considered in chapter 7.

2.7 An Alternative Approach to Naming

In Oasis a two-level approach to client naming is employed. Each process is given a unique identifier which is easily authenticated, and then gains higher level names called *roles* by interaction with different services. This idea was first proposed in Lampson's famous 'protection' paper [Lam71]. Our roles are approximately equivalent to Lampson's *access keys*, however in our scheme a client may obtain many high level names, and these may change during the lifetime of the client.

Typically, each service will provide its own name space, and arbitrate with processes wishing to gain these names. In the following section, the design and advantages of, low level identifiers are described, and the use of roles is discussed in section 2.9.

2.8 Client Identifiers

In Oasis a client identifier is a tuple (*host*, *id*, *boot time*), where *host* is the identity of the machine the client is executing on and *id* is an identity chosen by that machine that identifies the client. The third parameter *boot time* is to ensure that client identifiers remain unique for all time.

This scheme is extremely simple to implement on a variety of operating systems, and requires little from an authentication service. The host name must be authenticated in any system[3], and, in general, it is not possible to distinguish between clients on the same host, without the cooperation of the local operating system.

As the *id* part of the name is chosen by the client operating system, the identifier is 'future-proof'; each protection domain can be named individually, regardless of the nature of these domains. In the following section a suitable definition of *id* for a multi-user system is derived. This is the naming scheme used in our implementation, but each host is at liberty to choose its own names.

2.8.1 Hosts Supporting Multiple Clients

Protection domains on a client machine are rarely independent. Typically one domain will be responsible for the creation of another (such as when a process is forked in Unix). When a protection domain is created, the creator will typically wish to pass on some (but not all) of its credentials to the new domain. For example in Unix, the user identity relating to a process is generally copied to any sub-process. Although such 'handing on' of credentials could take place by explicit delegation, it is more efficient to provide a special mechanism for this common form of delegation.

Again, this problem can be solved by an extra level of indirection. The operating system is modified to provide *virtual client identifiers* that a protection domain may use to name itself. The intuition behind a VCI is that it is a name a client will use when performing a particular task. If a client performs several tasks and wishes to keep the credentials for these tasks separate, then it will use different VCIs. The operating system ensures that a domain may not use a VCI relating to a different domain, unless that domain explicitly delegates use of the VCI.

Whenever a protection domain obtains a credential, the credential is associated with a particular VCI, and can therefore only be used by protection domains who may name themselves using the VCI. A domain may therefore control the propagation of credentials to other domains, by controlling the propagation of VCIs. This is a lesser burden on the operating system, as VCIs are simple names that are meaningless outside the context of a particular host. There is considerable freedom in the detailed design and implementation of VCIs for a particular operating system.

For example, if a domain has credentials `A` and `B` relating to VCI `x`, and credentials `C` and `D` relating to VCI `y`; then it would be able to use any of `A`,`B`,`C` or `D`. If a child domain were created and allowed access to only VCI `x`, then it would be able to use `A` and `B`, but would be unable to use `C` and `D`, even if it 'stole' these from it's parent in some way.

A common requirement is for a login process to restrict access in this way. It may be used by many users and, for each, it would create a new VCI and acquire a suitable high level name from the login service. It would then fork a process for each user, passing on only the relevant VCI and login name.

[3]We do not need to know which machine it is, merely that a set of requests had the same origin.

2.9 Service-specific Naming — Roles

An Oasis role is a name that a service uses to distinguish classes of clients. Each name may be parametrised to identify a particular client, or list significant attributes of the client. For example a login service might provide a single role **LoggedOn** with parameters `UserId` and `Host`. A conferencing application might have roles **Chair** and **Member**. Definitions for these two examples are given in the next chapter, and used to illustrate how roles interact.

For each role it defines, a service provides a set of policy statements defining how a client may enter the role, and how roles interact. A client of the service is classified by obtaining signed bit-strings called 'Role Membership Certificates'[4] based on supplied credentials. The client is said to have 'entered' the role, and remains a 'member' of the role until membership is explicitly, or implicitly, revoked by the service. When performing an operation, a client presents the certificate to the service which interprets the certificate's parameters and decides the degree of access to allow.

A Role Membership Certificate is an idealised membership card. It is possible to examine its attributes, and to tell if a card has been forged, tampered with or revoked. An authentication mechanism is adopted that prevents certificates being used by other clients. This reduces the concerns of stolen certificates, and restricts clients to delegating access within the policies laid out by the service provider.

A crucial feature of the system is that certificates from one service may be provided as credentials when requesting a certificate from another service. This mimics real life, where a person may have to produce one card in order to obtain another (for example providing a proof of identity when applying for a credit card). It is this inter-operability, together with well-defined mechanisms for validation and revocation that is the basis for secure inter-working.

2.10 Scope Issues

Definitions of roles apply to some context, for example an instance of a Conference, or access to a set of files. To define the scope of a role, a set of definitions for role membership are grouped into a *rolefile*, and certificates for these roles are make specific to this rolefile, within the current instance of the service. For example there may be many instances of a conferencing application, each with a single rolefile defining who is the Chair, and rules for membership. Alternatively we may have a single conferencing application that supports many conferences, each with their own rolefile. In either case role membership certificates would be conference-specific.

To allow inter-working between services, a service may offer to validate role membership certificates for use in other services. Services offering this facility register a standard interface with a name server, thus allowing other services to (indirectly) validate certificates that they did not themselves issue. In the following chapter a conferencing application is used as an example, and this uses certificates from a Login service in the definitions of its roles.

[4] The architecture involves various certificates, but the term 'certificate' will be used where there is no ambiguity.

2.11 Summary

In this chapter we have reasoned that there are two conflicting requirements for client naming in a distributed environment. A single naming mechanism is required for simplicity and to enable secure interworking. However, diverse applications have different naming requirements and a single name space cannot hope to meet them all. In section 2.7 we suggested a compromise; a two level naming scheme that allows uniformity at the lower level, and diversity at the higher level. The lower level is easy to implement securely and uniquely identifies clients. The higher level represents contextual information about those clients. This is analagous to associating an object identifier and a type with each object in an object oriented programming environment.

Chapter 3

Language Definition

In the previous chapter the need for a flexible access control mechanism was highlighted. In particular the following points were made:

- Each service should be able to make appropriate distinctions between clients, by classifying them according to its own naming scheme.
- Clients of one service should be specified in terms of names given to them by other services.
- The specification of client roles, and interaction between roles should be distinct from the functional definition of the service.
- Specification should be flexible, but easily understood.

In this chapter, a *role definition language (RDL)* is proposed to meet these goals. In section 3.1 interaction between clients is considered, and the need for flexible specification of delegation and revocation is highlighted. Section 3.2 gives the definition of RDL itself, and section 3.3 considers extensions to this definition. Section 3.4 gives a number of examples, and uses these to discuss the extent to which RDL meets the above requirements.

3.1 Limitations of Existing Schemes

Traditional access control schemes have relied on either access control lists or capabilities, both for policy expression and as a mechanism for enforcing this policy. In this section we will consider the limitations of each of these schemes, with reference to their expressive power.

3.1.1 Access Control Lists

In an access control list scheme, a list of potential clients, and the level of access allowed by each, is stored with each protected object. ACLs are fundamentally limited by the choice of naming scheme for clients. As discussed in chapter 2, such a scheme must be flexible enough to allow all useful distinctions between clients, but simple enough to be verifiable and to be easily understood.

ACLs have a second limitation, in that there is no provision for specification of *client interaction*, where one client delegates access privileges to another. In a distributed environment, clients of a service may be 'software agents', printers, fax machines, and network gateways, as well as human users. The ability for a user to temporarily grant access rights to one of these 'value-adding' services is becoming increasingly important.

3.1.2 Capabilities Schemes

In *pure* capability schemes, there need be no classification of clients, and interaction between clients can take place easily by copying and/or refinement of capabilities. However, this is a mechanism not a policy. In capability schemes, there is generally no expression of policy regarding the kind and degree of delegation that may take place, and no mechanism for enforcing such a policy, if it existed. This limitation is overcome by several advanced capability schemes, such as I-Cap [Gon89]. In I-Cap, the service that issued a capability must be consulted whenever delegation takes place, and is in a position to enforce policy control. The method of policy expression is not defined in [Gon89], but it is reasonable to assume that it would take the form of an extended access control list[1]. This, unfortunately, requires a client naming scheme, and has the associated disadvantages.

3.1.3 Election

Emerging cooperative applications call for more complex client interaction. Voting and Election are common requirements; whereby one or more clients agree to give an individual special privileges. For example, clients of a conference may elect a Chair-Person. A similar process takes place in secure systems requiring 'joint delegation' whereby two or more 'signatures' are required before an action can take place. Little work has been done on delegation models, although several people have recognised the need for more general mechanisms [SL87, Yu89].

In the following, the term *election* is used to describe the process of one client granting role membership to another. If the role granted is already held by the electing client, then this is equivalent to delegation in traditional capability schemes.

3.1.4 Revocation

Revocation is the opposite to delegation. If delegation is allowed, then it is often reasonable to allow the delegator to reverse his decision. Typically, capability schemes that allow delegation also support some form of revocation. However, revocation is notoriously hard to implement efficiently and few schemes allow both *immediate* and *selective* revocation.

Although the decision to revoke is generally taken by the delegator, this may not always be the case. In some circumstances policy may dictate that the delegator may not revoke at all. For example once a voter has voted during an election, they may not change their mind. Equally, a full policy might dictate that a role membership should be revoked for reasons that are beyond the control

[1] A mixed capability and ACL scheme such as this is used as the basis of access control for the MSSA[Lo94], and is discussed in chapter 5.

of the delegator. For example a certificate might be revoked if it were based on an existing membership that was itself revoked.

A full policy definition must include details of when revocation will take place, both in terms of the delegator's request, and these other issues. In section 3.2.3 we will discuss how such choices may be specified in RDL.

3.2 RDL

RDL is a language for specifying all aspects of role management. Its three main functions are:

- To name a set of roles, and specify the parameters associated with each one.
- To give the conditions for client entry to each role, in terms of existing role membership, election and other constraints.
- To specify the conditions under which a client's membership of a role should be revoked.

These functions are considered in turn, in the following three sections.

3.2.1 Role Declaration

Role declaration statements declare the names of roles, and the type and number of their arguments. They do not, in themselves, define how the role is entered, or how it is used. They are of the form

$$\text{def } \mathbf{Rolename}(\mathit{arg}, \ \ldots\ ) \quad \mathit{arg} : \text{type}\ [\mathit{arg} : \text{type} \ldots]$$

Arguments are strongly typed and *type* may be one of 'Integer', 'String', a specification of a set type such as {`rwx`}, or the name of an object type. If an object type name is specified, then a table of parse functions is consulted, to allow the RDL parser to interpret literals of this type. Object and set types are 'simple' in the sense that there is no sub-typing or type compatability. These features were not considered important for such a simple language.

RDL provides a comprehensive type inference scheme, and only argument types that cannot be inferred by examination of other statements need to be specified explicitly in declaration statements. If all argument types can be inferred in this way, the declaration statement is redundant and may be omitted.

Roles defined in one server may make use of object types defined in another server by *importing* type definitions. For example

$$\begin{array}{l}\text{import } \mathbf{Login}.\texttt{userid} \\ \quad \text{def } \mathbf{Member}(u) \quad u : \texttt{userid}\end{array}$$

As with declaration statements, import statements may be omitted if the source of a type may be inferred. In the rest of this chapter, role declaration and import statements are only given where necessary.

3.2.2 Role Entry Statements

There are two forms of role entry statement, both of which define admissible forms of role entry. The first form is used to indicate role entry based a candidate client's credentials, and the second form is used to specify delegation, or election.

Standard Form

The standard form of a role entry statement defines entry to a role in terms of role memberships a candidate client already holds. It is of the form

$$\textbf{Role Name}([\textbf{arg}, \ldots]) \leftarrow \textbf{Role Reference}\ [\wedge\ \textbf{Role Reference} \ldots]\ [:\ \textbf{Constraint}]$$

A role reference may correspond to a role defined in the same rolefile, or to a role issued by another service. In the latter case, the role reference consists of a service name identifying the instance of the issuing service, and an optional identifier to locate a particular rolefile within the service. By explicitly naming the issuing service, the need for a global name space for roles is avoided. Each role reference may have a number of arguments, each of which may be a variable or literal of the appropriate type. These can be type-checked by a callback to the issuing service, and object identifier literals are also parsed in this way.

The constraint is an optional expression in terms of the roles' arguments that limits the applicability of the statement.

Election Form

In the election form of role entry statement, a client may only enter the role if a third party wished to elect them. An additional phrase specifies a role that a suitable elector must hold, and the constraint expression may limit the extent of election. Delegation statements are a special case of election statements, where the elector role is the same as that being defined.

$$\textbf{Role Name}([\textbf{arg}, \ldots]) \leftarrow \textbf{Role Reference}\ [\wedge\ \textbf{Role Reference} \ldots]\ \triangleleft\ \textbf{Elector's Role Reference}\ [:\ \textbf{Constraint}]$$

RDL statements as axioms

An RDL role entry statement can be considered as an axiom in a proof system, where the conditions on the right hand side of the $\leftarrow$ are premises and the role entry on the left is the conclusion. Role entry is validated by proving the premises, and inferring the validity of the conclusion.

An example rolefile for a conferencing application is given in figure 3.1, together with the axioms it represents. This consists of two role entry statements defining the roles **Chair** and **Member**. The first states that a client holding a **LoggedOn** certificate issued by the Login service for user `jmb` may enter the role **Chair**. The second statement is an election statement and specifies that a client may be elected by the **Chair** to the role **Member**, providing that they already hold a suitable **LoggedOn** certificate. The constraint 'u in `staff`' constrains the applicability of this statement to clients that represent users who are members of the group `staff`.

$$
\begin{array}{lll}
\text{import } \mathbf{Login}.\texttt{userid}^{\dagger} & & \\
\quad \text{def } \mathbf{Chair}^{\dagger} & & \\
\quad \text{def } \mathbf{Member}(u) & & u : \texttt{userid}^{\dagger} \\
\qquad \mathbf{Chair} & \leftarrow & \mathbf{Login.LoggedOn}(\texttt{jmb}, h) \\
\qquad \mathbf{Member}(u) & \leftarrow & \mathbf{Login.LoggedOn}(u, h) \lhd \mathbf{Chair} \;\; : u \text{ in } \texttt{staff}
\end{array}
$$

$$
\frac{\begin{array}{c} c \text{ owns } \mathbf{Login.LoggedOn}(\texttt{jmb}, h) \\ c \text{ requests entry to } \mathbf{Chair} \end{array}}{c \text{ owns } \mathbf{Chair}}
$$

$$
\frac{\begin{array}{c} c \text{ owns } \mathbf{Login.LoggedOn}(u, h) \\ c \lhd c' \\ c' \text{ owns } \mathbf{Chair} \\ u \text{ in } \texttt{staff} \\ c \text{ requests entry to } \mathbf{Member}(u) \end{array}}{c \text{ owns } \mathbf{Member}(u)}
$$

Figure 3.1: An Example Rolefile, and Associated Axioms

The rolefile given consists of two rules defining entry to the roles for a conferencing application. The statements marked with a dagger may be omitted as the typing information they contain can be inferred.

Multiple Rules

There may be several statements defining entry to the same role, and a client meeting the constraints in any of the statements will be granted role entry. As the application of different statements may result in role entry with different parameters or semantics, it is important to define a precedence between statements.

In addition a client may be able to enter a role indirectly — by obtaining some intermediate role. Although in theory, a client could enter each intermediate role explicitly, it is desirable for entry to intermediate roles to take place automatically when required. This allows intermediate roles to be introduced whenever this aids expression of policy without the need to modify each client application to take appropriate action. An added bonus is that such an approach is more efficient, both in terms of the number of RPCs required and the amount of cryptographic computation.

Clearly the use of intermediate roles and multiple statements referring to the same role could, in theory, lead to ambiguous rolefiles. In practice it is unlikely that a policy would be expressed in such a way as to appear ambiguous, but for completeness, the precedence between conflicting rules is given below.

> For each request, a list of role memberships is created. This initially contains the roles the requesting client already holds.
>
> Each statement in the rolefile is applied in turn, and if a membership results, this is appended to the tail of the list.

$$\begin{aligned}\mathbf{Bas}(1) &\leftarrow \mathbf{Foo}\\ \mathbf{Bas}(2) &\leftarrow \mathbf{Foo}\\ \mathbf{Bar}(1) &\leftarrow \mathbf{Bas}(2)\\ \mathbf{Bar}(2) &\leftarrow \mathbf{Foo}\end{aligned}$$

Figure 3.2: An Ambiguous Rolefile?

For a client holding **Foo** and wishing to enter **Bar** which should be applied $\mathbf{Bas}(2) \leftarrow \mathbf{Foo}$ followed by $\mathbf{Bar}(1) \leftarrow \mathbf{Bas}(2)$, or $\mathbf{Bar}(2) \leftarrow \mathbf{Foo}$? In RDL, the former definition is used.

> When applying each statement, any of the memberships in the list may be used as a credential, and the first suitable one found will be used. Ultimately, all but the requested membership is discarded, and this is returned to the client.

Consider the example in figure 3.2. Should a client holding role **Foo** and wishing to enter role **Bar** receive $\mathbf{Bar}(1)$ or $\mathbf{Bar}(2)$? Application of the above rule will result in the list of memberships

$$\mathbf{Bas}(1), \mathbf{Bas}(2), \mathbf{Bar}(1), \mathbf{Bar}(2)$$

The first suitable membership, $\mathbf{Bar}(1)$, is returned.

3.2.3 Specifying Revocation

In the above sections, entry to a role was specified in terms of a number of conditions that must be true to allow entry. We term these conditions *entry conditions*. In section 3.1.4 a number of potential reasons were given for choosing to revoke a client's membership of a role; these correspond to the negation of 'significant' entry conditions. Significant conditions are termed *Membership Rules* as their continuing validity is a condition of role membership. Membership rules are indicated by annotating entry conditions within a role entry statement with asterisks.

There are four kinds of entry condition which may also be membership rules.

- The fact that a candidate is a member of one or more roles.
- The fact that an elector wishes the client to become a member of the role.
- The fact that an elector is in a position to elect the client (a member of an appropriate role).
- The fact that a parameter to one of the specified roles, *is* or *is not* the member of some group (for example, a user being a member of staff).

The fifth kind of entry condition — that there is a particular relationship between the parameters to these roles — cannot change during the lifetime of the role membership, so need not be considered here. In traditional capability schemes, only negation of the second or third condition can lead to revocation.

$$
\begin{array}{rll}
\textbf{Constraint} \rightarrow & \textbf{Unit} \mid \neg\textbf{Constraint} \mid (\textbf{Constraint}) & \\
\rightarrow & \textbf{Constraint} \vee \textbf{Constraint} & \\
\rightarrow & \textbf{Constraint} \wedge \textbf{Constraint} & \\
\textbf{Unit} \rightarrow & \textbf{Atom} & \\
\rightarrow & \textbf{Atom}^* & \text{Part of membership Rule} \\
\textbf{Atom} \rightarrow & \textbf{Variable Comparitor Expression} & \\
\textbf{Comparitor} \rightarrow & = \mid \neq \mid < \mid \leq \mid > \mid \geq & \text{Integer Comparitors} \\
\rightarrow & = \mid \neq \mid \subset \mid \subseteq \mid \supset \mid \supseteq & \text{Set Comparitors} \\
\rightarrow & = \mid \neq \mid \textbf{in} & \text{Object Identifier Comparitors} \\
\textbf{Expression} \rightarrow & \textit{Integer Expression} & \\
\rightarrow & \textit{Set Expression} & \\
\rightarrow & \textit{Group or Object Identifier} &
\end{array}
$$

Figure 3.3: Constraint Expression

RDL is considerably more powerful, as not only can the circumstances of this form of revocation be clearly specified, but other conditions that may breach security policy may be identified and used to trigger revocation.

In terms of a proof system, revocation takes place when membership can no longer be proved using the axioms formed by the membership rules involved in role entry. For example, consider the role entry statement

$$\textbf{Member}(u) \ \leftarrow \ \textbf{Login.LoggedOn}(u,h)^* \lhd^* \textbf{Chair} \ : (u \text{ in } \texttt{staff})^*$$

If a client c_1 representing the user `dm` on host `ely` is elected to the role **Member** by a client c_2, then its continued membership is represented by the axiom

$$\frac{\begin{array}{c} c_1 \text{ owns } \textbf{Login.LoggedOn}(\texttt{dm}, \texttt{ely}) \\ c_1 \lhd c_2 \\ \texttt{dm} \text{ in } \texttt{staff} \end{array}}{c_1 \text{ owns } \textbf{Member}(\texttt{dm})}$$

If, for example, `dm` was subsequently removed from the group `staff`, it would no longer be possible to prove

$$c_1 \text{ owns } \textbf{Member}(\texttt{dm})$$

and membership of this role would be revoked. Equally, if the delegation $(c_1 \lhd c_2)$ were revoked, or the login certificate was invalidated, the membership of the role **Member** would also be revoked.

3.2.4 Constraint Expression

The constraint expression in a role entry statement is a boolean expression in terms of the parameters of the associated role membership certificates. If this expression contains one or more instances of the $*$ operator, then a membership rule is formed by substituting in the value of all the other subexpressions at the time of role entry. The full grammar for a constraint expression is given in figure 3.3.

3.3 Extensions to RDL

The language described above allows entry to a role based on restricted election, or on ACL like structures within a role definition file. Membership of a role may be revoked if the elector so decides, or if assumptions made on entry no longer hold. In this section possible extensions to RDL that allow greater expressive power are considered. These, or other extensions, are provided by applications that require them.

3.3.1 Attribute Based Access Control

Statements in RDL are in terms of attributes of the principal requesting access, and those of electors. Protected objects themselves are not defined within RDL, and there is therefore no way of allowing access based on attributes of these objects. Attribute based access control is relatively costly to implement [MS91], but is necessary for some applications.

RDL can be extended to allow access to an object's state by providing a server specific set of functions to be used in the constraint expression. These functions may be entry conditions or membership rules, although the latter requires considerable cooperation from the service itself. The example 'Shared Authorship' in section 3.4.4 makes use of two of these functions.

3.3.2 Role Based Revocation

Revocation has been considered as the negation of a delegation. In some circumstances we require revocation *without* a matching delegation. For example, consider an open meeting. We may wish to reserve the Chair's right to revoke the membership of a client, despite the fact that they were not involved in the client's original award of the 'Member' role.

There are other situations, where a client *is* elected to a role, but where a client other than the elector has the right to revoke. For example, in business, the election 'Hire' and associated revocation 'Fire' are rights associated with a Manager. However they are associated with the *role* Manager, not the particular person currently filling that position.

RDL can be extended to accommodate these situations. We add additional syntax in the form of a 'revoke' operator ($\triangleright$), which has the intuitive meaning that a membership may be negated by a member of the role indicated. The parameters of the delegated role are used by the revoker to specify which role membership certificates are to be revoked. This is significant as a revoker may not be aware of the client's identity.

The RDL for an open meeting with this extension is as follows

$$\begin{array}{lcl} \textbf{Chair} & \leftarrow & \ldots \\ \textbf{Member}(p) & \leftarrow & \textbf{Person}(p) \triangleright^* \textbf{Chair} \end{array}$$

This indicates that a client holding the role **Person**(p) may enter the role **Member**(p), but that that membership may later be revoked by a client holding **Chair**. In terms of a proof system, the membership condition axiom is extended with an additional clause stating that revocation has not taken place. For example, the axiom indicating that a process p representing a person `Fred` may become a member is

$$\frac{\begin{array}{c} p \text{ owns } \mathbf{Person}(\texttt{Fred}) \\ p \text{ requests entry to } \mathbf{Member} \\ \neg\mathbf{Revoked}(\mathbf{Member}(\texttt{Fred})) \end{array}}{p \text{ owns } \mathbf{Member}}$$

The implementation of this extension is considered in section 4.11.

3.3.3 Expressing Access Control Lists

Access control lists are a common form of policy specification, and in order to support legacy systems, it is useful to be able to represent existing access control lists as a set of RDL rules. However, depending on the semantics of the ACL system being mimicked, this translation may not be straightforward. In particular RDL statements are *independent*. Entry to a role will result in the application of a single rule. ACL systems are often *cumulative* whereby the rights granted are based on the application of several 'rules'. For example the ACL statement '`students may have read access`' and the statement '`user rjh21 may have write access`' may be combined to give both read and write access to `rjh21` if he were a student.

Whatever the semantics of an ACL system, it can be generalised to a mapping between a client identifier and a set of access rights. In RDL, this may be represented by extending the constraints expression with a parametrised function of this form. For example the Unix acl '`rjh21=rwx staff=r-x other=r--`' may be expressed as

$$\begin{array}{lll} \mathbf{UseFile}(r) & \leftarrow & \mathbf{LoggedOn}(u) \\ & & :r = \mathbf{unixacl}(\text{``}\texttt{rjh21=rwx staff=rx other=r}\text{''}, u) \end{array}$$

This indicates that a client may obtain a role $\mathbf{UseFile}(r)$ where r is the rights allocated to them on the ACL for the protected file.

In the Unix filing system, access to a file is restricted by the access control lists on the parent directories, in addition to the ACL on the file itself[RT78]. In order to aid reasoning about the interworking of such legacy systems with Oasis services, it is useful to be able to express such a scheme in RDL. This can be done using the extensions discussed in section 3.3.1. Two functions are required, a function $\mathbf{InDir}(f, d)$ that returns `true` if f has an entry in the directory d, and a function $\mathbf{Root}(d)$ that returns `true` if the directory d is the root directory. We represent each ACL as an entry within a single rolefile of the form

$$\begin{array}{lll} \mathbf{ACL}(r, \mathit{fileid}) & \leftarrow & \mathbf{LoggedOn}(u) \\ & & :r \ = \mathbf{unixacl}(\text{``}\mathit{access\ control\ list}\text{''}, u) \end{array}$$

We then add two rules indicating how ACL entries are related to rights given to clients.

$$\begin{array}{lll} \mathbf{Rights}(r, f) & \leftarrow & \mathbf{ACL}(r, f) \qquad\qquad\qquad\qquad :\texttt{Root}(f) \\ \mathbf{Rights}(r, f) & \leftarrow & \mathbf{ACL}(r, f) \wedge \mathbf{Rights}(s, d) \ :\texttt{InDir}(f, d) \wedge \{\texttt{x}\} \subseteq s \end{array}$$

Under Oasis ACLs for new applications are specified using a standard format, accessible by the parametrised function **acl**. This is described in the context of a storage service in section 5.4.4.

3.4 Some Examples

Some simple examples of the use of roles are given in the following sections. Longer case studies are given in chapters 5 and 7.

3.4.1 High Score Table

A game with a private high score table is often given as an example application requiring access control. Only processes running the game application may write to the high score table, which is stored in a filing system, but any logged in user may read it. For this example, it is assumed that a Loader service is available that will validate that a particular client identifier represents the execution of a particular program image. This loader is likely to consist of two parts; one local to the client machine, that interfaces with the operating system and certifies loading, and a central secure service that will rule on the validity of statements made by client loaders, based on the assumed integrity of the client host. The central loader may then issue certificates to client processes. The rolefile given is that for the high score file, and is self explanatory.

$$\begin{array}{lcl}
\textbf{def UseFile}(r) & & r = \{\texttt{rwx}\} \\
\quad\textbf{UseFile}(\{\texttt{r}\}) & \leftarrow & \textbf{Login.LoggedOn}(u) \\
\quad\textbf{UseFile}(\{\texttt{rw}\}) & \leftarrow & \textbf{Loader.Running}(\text{``}\texttt{Space Invaders}\text{''})^*
\end{array}$$

3.4.2 Open Meeting

This is similar to the Conferencing example used throughout this chapter, but with more flexibility with regard to delegation. It is assumed that the meeting has a Chair, and that any member of staff may join the meeting. A requirement is that any member of the meeting may invite someone else to join. This is an example of unrestricted, recursive delegation.

$$\begin{array}{lcll}
\textbf{Chair} & \leftarrow & \textbf{Login.LoggedOn}(\texttt{rmn}, h) & \\
\textbf{Member} & \leftarrow & \textbf{Login.LoggedOn}(u) & : u \text{ in } \texttt{staff} \\
\textbf{Member} & \leftarrow & \textbf{Login.LoggedOn}(u) \triangleleft \textbf{Member} &
\end{array}$$

We could extend this example still further by allowing the Chair to eject anyone from the meeting. This may be done using the extension described in section 3.3.2. Revocation of this form is specified using the parameters of the delegated role. We could modify the role **Member** to take the user's identity as a parameter, but this might involve modifications to the application code. Instead, we add an intermediate role **Candidate**. Note that clients may still request entry to the role Member by supplying the same credentials as before; the role Candidate is entered automatically, as describe in section 3.2.2

$$\begin{array}{lcll}
\textbf{Chair} & \leftarrow & \textbf{Login.LoggedOn}(\texttt{rmn}, h) & \\
\textbf{Candidate}(u) & \leftarrow & \textbf{Login.LoggedOn}(u) \triangleright^* \textbf{Chair} & \\
\textbf{Member} & \leftarrow & \textbf{Candidate}(u) & : u \text{ in } \texttt{staff} \\
\textbf{Member} & \leftarrow & \textbf{Candidate}(u) \triangleleft \textbf{Member} &
\end{array}$$

3.4.3 Login with Passwords

A common example of role entry using multiple candidate certificates is authenticated login. Generally there will be a central password service, that is responsible for maintaining passwords, or other user authentication information. This may be used by any other service requiring user authentication, for example a login service. After a discourse between the client and the password service, the password service issues the client with a role membership certificate stating that they have been authenticated.

Internally, the password service stores a set of secrets associated with a number of keys. For example a client wishing to perform login may request a password certificate of the form **Passwd**(*userid*, `Login`). Login itself can then be performed by presenting the password certificate to the login service, which will perform any additional checks, such as on the identity of the host.

$$\begin{array}{lll}
\mathbf{Login}(u) & \leftarrow \ \mathbf{Pw.Passwd}(u, \texttt{Login}) \ \wedge \ \mathbf{Host}(h) & : h \text{ in } \texttt{hosts} \\
\mathbf{Secure}(u) & \leftarrow \ \mathbf{Pw.Passwd}(u, \texttt{Login}) \ \wedge \ \mathbf{Host}(h) & : h \text{ in } \texttt{secure} \\
\mathbf{Untrusted}(u) & \leftarrow \ \mathbf{Pw.Passwd}(u, \texttt{Login}) & \\
\mathbf{Visitor}(u) & \leftarrow &
\end{array}$$

The above rules define four different login roles. **Login** is the role usually used; **Secure** is a login on one of a set of trusted machines; **Untrusted** is login from an unknown machine; and **Visitor** is a login consisting of an unchecked client claim that they represent a particular user. The above rolefile requires a client to decide which role they wish to enter, This decision could be left to the login service by fixing the role type as a parameter. For example

$$\begin{array}{llll}
\textbf{def}\,\mathbf{Login}(l, u) & & l : \texttt{integer} & \\
\quad \mathbf{Login}(3, u) & \leftarrow & \mathbf{Pw.Passwd}(u, \texttt{Login}) & : h \text{ in } \texttt{secure} \\
\quad \mathbf{Login}(2, u) & \leftarrow & \mathbf{Pw.Passwd}(u, \texttt{Login}) & : h \text{ in } \texttt{hosts} \\
\quad \mathbf{Login}(1, u) & \leftarrow & \mathbf{Pw.Passwd}(u, \texttt{Login}) & \\
\quad \mathbf{Login}(0, u) & \leftarrow & &
\end{array}$$

This gives a client the 'maximum' permissible role, dependent on the machine they are on. If a client wishes to explicitly Login at a particular level, they may specify the l parameter in their login request. Note that in the second set of definitions, the rules for 'secure' and 'hosts' have been reversed. This is because the *first* matching rule is used.

3.4.4 Shared Authorship

For shared authorship a requirement may be to identify the author implicitly as the creator of the document, rather than explicitly by entering their `userid` in the rolefile. This would allow one rolefile to be used for many documents with different authors. Let us imagine that each document has one author, and one editor. The author may edit or annotate the document, and the editor may annotate it, and decide when it is in its final form. Once finalised, the author and editor may still make annotations, but no more edits are allowed.

Rolefile for document DOC:

$$\begin{array}{lll} \textbf{Author} & \leftarrow \textbf{Login.LoggedOn}(u) & : u \leftarrow \texttt{creator(DOC)} \\ \textbf{Editor} & \leftarrow \textbf{Login.LoggedOn}(\texttt{MrEd}) & \end{array}$$

$$\begin{array}{lll} \textbf{def Rights}(r) & r : \{\texttt{eaf}\} & \\ \textbf{Rights}(\{\texttt{ae}\}) & \leftarrow \textbf{Author} & : \neg\texttt{finalised}()^* \\ \textbf{Rights}(\{\texttt{a}\}) & \leftarrow \textbf{Author} & \\ \textbf{Rights}(\{\texttt{af}\}) & \leftarrow \textbf{Editor} & \end{array}$$

3.4.5 Playing Golf

In many golf clubs, a new player wishing to join the club requires two recommendations from different existing members. This may be modelled in RDL as the acquisition of a set of credentials, followed by entry to the role Member. Constraint expressions are used to ensure the recommendations are made by different members. Assuming the role Candidate and Member have both already been defined, and take a single argument of a 'person identifier', a suitable rolefile is as follows.

$$\begin{array}{ll} \textbf{Recommended}(p,x) & \leftarrow \textbf{Candidate}(p) \triangleleft \textbf{Member}(x) \\ \textbf{Member}(p) & \leftarrow \textbf{Recommended}(p,x) \wedge \textbf{Recommened}(p,y) \ : \ x \neq y \end{array}$$

This example demonstrates how quorum delegation can be easily specified. It is possible to arrange that any of the electors may revoke the membership simply by adding the appropriate *s. For a service to specify a quorum for revocation, the extension described in section 3.3.2 is required.

3.5 Summary

In chapter 2, we indicated that role membership certificates provided a flexible and efficient means of distinguishing between the clients of a service. In this chapter we have derived a language for specifying the conditions for role entry and exit, and described how the interaction between clients may be clearly specified. This allows us to determine when a certificate should be issued, and when it should be revoked. Policy statements may be represented as axioms in a proof system, aiding automatic reasoning about the interaction of different policies. In the following chapter we will discuss how such certificates can be validated, and how the detection of the conditions for revocation described in this chapter can be implemented efficiently.

Chapter 4

System Architecture

4.1 Introduction

The notion of roles and the role definition language described in the last chapters provide a comprehensive and flexible access control architecture. This chapter considers how such an architecture can be implemented, both in terms of the interface between different servers, and the flexibility that a particular service has in optimising its performance.

4.2 Validating Certificates

When a client performs an operation, it will supply a role membership certificate. Validation of this certificate should fail if any of the following is true:

1. The client is acting under an identifier other than its own.
2. The certificate is forged, or modified.
3. The certificate is stolen.
4. The certificate was issued by a different service, or for use in a different context.
5. The certificate is valid, but does not embody sufficient rights for the requested operation.
6. The certificate has been (or may have been) revoked, either explicitly, or implicitly (for example by a change in group membership).[1]

If any of the first three conditions hold, then this constitutes fraud by the client. If conditions 4 or 5 hold then the client is acting incorrectly; either erroneously or in an attempt to gain illicit access. Condition 6 is the only condition that a 'well behaved' client may trigger. It is advantageous to be able to distinguish between these three classes of failure. Fraudulent or erroneous accesses could

[1]Issues such as network failure may lead to a server being unaware of a revocation request. If there is a possibility that a certificate has been revoked then the server should generally act as if it has been. This issue is considered in depth in section 4.9.

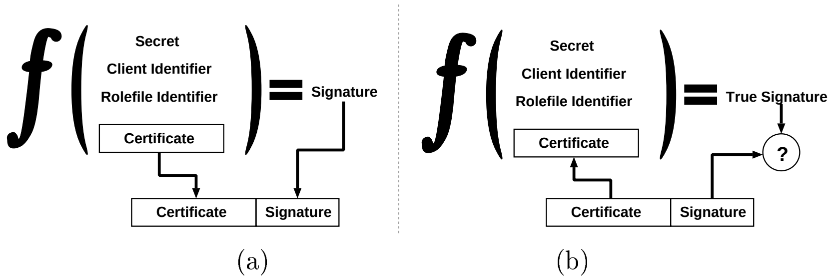


Figure 4.1: Preventing Forged Certificates

A one way function is used to protect the text of a certificate, and to associate it with a particular client-id and rolefile(a). When the certificate is later presented for verification, if the value of any of these has changed, then the recorded signature will not match the true signature (b). As 'Secret' is known only to a single instance of a service, clients are unable to discover the value of the true signature to match a modified certificate. In addition, certificates may only be validated by the instance of the service that created them, and thus use out of context is prevented.

then be recorded and used to identify miscreant users, or suspect applications. In Oasis, validation failures due to fraudulent or erroneous use are detected separately from those caused by revocation. This information can be made available for administration purposes.

There are three stages to validation; first the client identifier is validated using a suitable authentication protocol. Secondly, the integrity of the certificate, and the context in which it is used, are validated by (re)computation of a digital signature. Finally a field within the certificate is used to detect revoked certificates. After these three stages the certificate is known to be valid, and it is left to application specific code to check whether it embodies suitable access rights for the requested operation.

A typical digital signature function is shown in figure 4.1. This is based on that used in [Gon89]. Unlike other schemes, in Oasis, the *only* function of the digital signature is to check that the signature is not forged. Once the check has been performed, the integrity of the certificate may be cached, and recomputation avoided. This is particularly significant in distributed architectures where validation may involve a remote procedure call to the issuing service.

By contrast, some existing architectures make use of the fact that a recomputation of the signature will fail if the secret is changed. In the original MSSA scheme, for example, revocation took place by changing the secret[Lo94]. Although efficient, this has the disadvantages that fraudulent behaviour could not be distinguished from reasonable behaviour, and that revocation could not take place on a per-certificate basis.

Although a digital signature is the mechanism generally used in Oasis, a service is at liberty to use other forms of integrity check. For example, a service that issues only a small number of certificates may simply maintain a record of what has been issued, rather than relying on cryptography. Equally, a service requiring little security may use a cheap signature function, with small signatures; whilst a more security conscious service may use long signatures, and more expensive cryptographic techniques. This is in keeping with the aim of

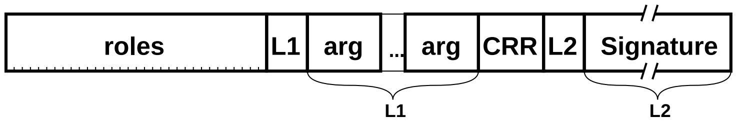


Figure 4.2: Format of a Role Membership Certificate

allowing a service to choose its own efficiency trade-offs. The format of a certificate allows for a variable length signature, although a particular service will generally issue certificates of a particular length, which allows for more efficient manipulation.

4.3 Certificate Format

A certificate must contain the name of the role it represents, together with any arguments that the role takes. In addition it must contain sufficient information to protect it against misuse, as described above.

To alleviate the need for a client to hold several certificates for related roles, compound certificates can be returned. These represent membership of more than one role. For example in the meeting example, it is likely that a client who holds the **Chair** role will be a **Member**. Both of these roles may be entered with a single request, and the client application need not distinguish between the roles when performing operations. To allow for compound certificates, the certificate format contains a set of role names. The current implementation of this architecture is limited to compounding roles that have identical arguments, although this is not a fundamental limitation. Each role is represented by a specific bit, and although the actual mapping is unimportant, it must not change during the lifetime of the service. To ensure this, the mapping is provided as configuration information when a service is initialised.

The format of a certificate is shown in figure 4.2. CRR is a credential record reference. This is an eight byte field used for revocation purposes, and is described in section 4.6. The arguments of the certificate are strongly typed by the issuing service, and must be marshalled into a host independent format to allow other services to examine their values. Object identifiers may be compared for equality in their marshalled form, which is the only admissible comparison for object identifier types. Sets are marshalled to a bit-set type, which allows equality and subset tests to be performed. Operations `gettypes` and `parsename` are provided in the server interface to allow other services to determine argument types and parse literals.

4.4 Delegation

In order to delegate membership of a role, a client requests a 'Delegation Certificate' from the appropriate server. As a side effect of this a 'Revocation Certificate' may be returned which the client can use to revoke the delegation. The Delegation certificate is then passed to the client to be elected, who accepts the delegation by using the certificate as a credential when entering the named role. In this way both parties must agree to the delegation, and the delegator

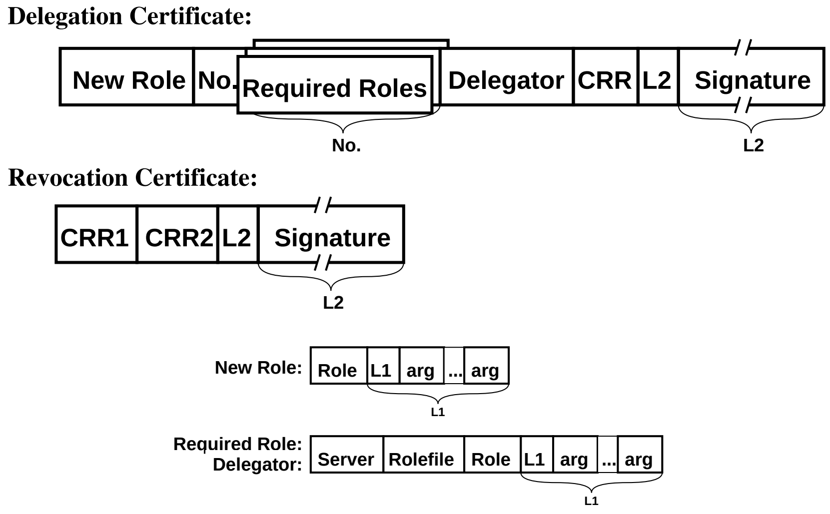


Figure 4.3: Format of Delegation and Revocation Certificates

can be assured of the ability to revoke, regardless of network failures during this process.

As clients are identified by low level identifiers, when delegating membership of a role, the candidate client must be identified by one or more roles that they hold. For example a client representing user `Bob` would delegate to a client representing user `Jim` by explicitly stating that a candidate client must hold **Login.LoggedOn**(`Jim`). This allows for delegation to clients who have yet to be given low level identifiers, and reduces the chance of a client being fooled into delegating to an imposter. A delegator is at liberty to specify any number of roles that a client must possess in order to utilise the delegation certificate. This may lead to a large number of certificates being involved in role entry. For this reason, role entry due to delegation, and role entry due to 'standard' credentials are implemented by separate RPC calls.

An additional advantage in specifying candidate clients in terms of the roles that they hold is that delegation may be specified for periods longer that the lifetime of a low level client identifier. It is often appropriate to use one digital signature function for short lived certificates, such as role membership certificates, and another more secure function for long lived certificates, such as delegation certificates, that might be prone to different forms of cryptographic attack.

Although long term delegation can be useful, short term delegation is much more common. A safety feature is that a delegator may specify a time limit on the life of the delegation, after which automatic revocation should take place.[2] This prevents the situation of un-revokable delegation due to a lost revocation certificate, and has the useful side effect of allowing the server to delete stale revocation information periodically. A delegator may also specify that revocation should take place if and when their own role is exited.[3] These features are

[2]Of course, this is only possible if the rolefile allows revocation.
[3]A client exits a role by voluntarily giving up membership, for example by logging off.

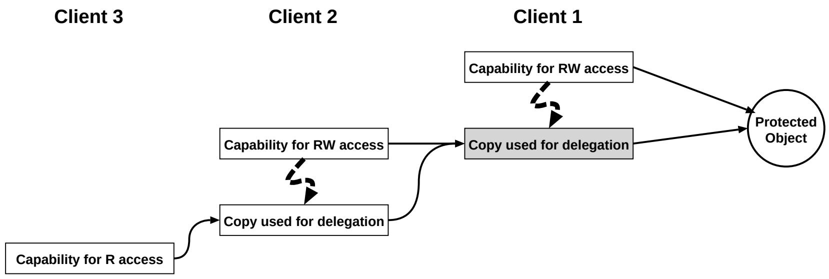


Figure 4.4: Chaining Capabilities

In this example, client 1 has delegated to client 2, who in turn has delegated a restricted form of his capability to client 3. For client 3 to use the capability all capabilities along the chain must be validated. Client 1 may revoke the delegated accesses by destroying the shaded capability. This will prevent access by both clients 2 and 3.

significant, as they help reduce the number of forgotten delegations, and hinder attacks whereby a client is prevented from accessing a server in order to issue revocation requests.

The format of delegation and revocation certificates are given in figure 4.3. Note that the 'required roles' specified in the delegation certificate are the roles the delegator requires the candidate to have. In addition to these, the candidate must supply certificates matching the candidate roles specified in the rolefile. In the revocation certificate, there are two credential record references. The first ensures that the delegator is still a member of the delegating role, and the second represents the credential to be invalidated. This is described in detail in the following sections.

When long term delegation takes place, there must be a method of revocation available which is suitably long lived. To allow this, a special delegation certificate is created that delegates the right to revoke, rather than the right to enter a role. There is a fixed policy for this: that a client may only delegate to another member of the 'elector' role.

4.5 Revocation

There are three approaches to revocation commonly used in capability schemes. Firstly, when a capability is revoked, all instances of it may be physically removed from clients. This is a reasonable approach in centralised systems (for example Multics [Org72]), but in a distributed environment where copying of capabilities cannot be prevented, this is not appropriate. The second approach is to store state about all invalid or revoked capabilities, and consult this database on each access. If revocation is rare, and capabilities are short lived, or eventually collected by some other complementary system, then this is a reasonable approach. This scheme is used in I-Cap, together with an (undefined) long term collection scheme.

The third approach, which is used by Oasis, is to store information about every valid capability. This is consulted on each access. This approach has the

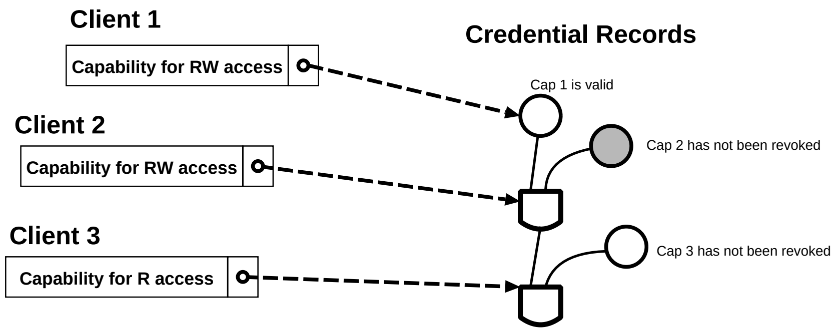


Figure 4.5: Delegation with Credential Records

Recursive delegation corresponds to a tree structure in the credential records. For example client 1 may revoke the capability delegated to client 2 by invalidating, or deleting, the shaded credential record. To allow this, client 1 is given a special 'revocation certificate' that contains a reference to the shaded record.

disadvantage that state must be stored for all capabilities, even if there is no revocation. However, if revocation is common, or capabilities are long lived, there are likely to be more revoked capabilities than valid ones.

A common implementation of this method is 'capability-chaining' whereby a delegator passes on an indirected capability, thus allowing later revocation by breaking the path[Red74]. This is illustrated in figure 4.4. Such a scheme is flexible, but is inefficient as long chains of capabilities due to recursive delegation require a large amount of stored state and many cryptographic checks.

In the design of Oasis, a key aim was to allow revocation for a number of reasons, such as change of group membership. This requires a more efficient mechanism than capability-chaining. A secondary issue is that revocation must be performed in a distributed environment, and so a solution must be scalable, and tolerant to independent server failure, or message loss. In the following section the notion of 'credential records' is introduced, as a basis for revocation. Sections 4.6 to 4.8 discuss the implementation of a revocation scheme within a single service and section 4.9 expands the discussion to a distributed environment.

4.6 Credential Records

Credential records are a methodology to allow flexible, selective revocation due to any number of 'significant events'. A credential record is a small record stored in a server that represents that server's current belief about some fact. Unlike certificate-chaining, only the belief is stored, not the actual fact. This allows credential records to be small, and to represent arbitrary facts. Records form a directed graph, such that a child represents some function of the beliefs held about its parents. In this way, only a single credential record need be consulted to confirm an arbitrary number of facts. A field is added to each certificate called a 'credential record reference'. This is a reference to a record within the issuing server that represents the validity of the certificate. The name space

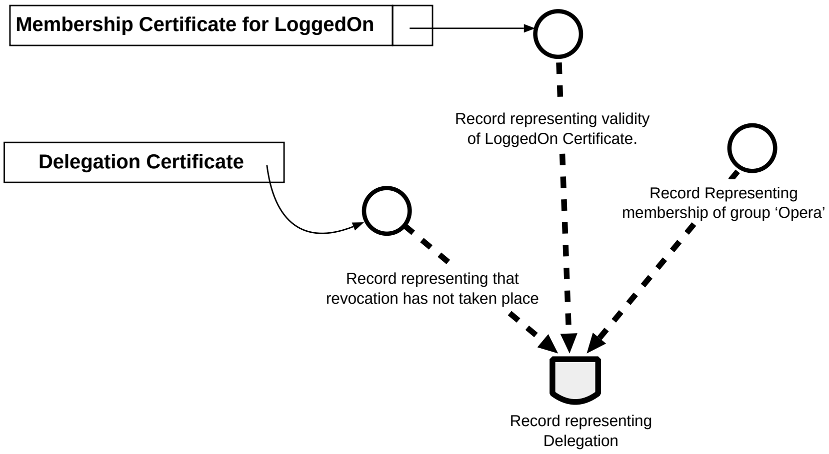


Figure 4.6: Entering a Delegated Role

When a client enters the role Member, they must supply a Login Certificate, and a delegation certificate. From the references embedded in these two certificates, and the one found by group membership lookup, the illustrated graph can be constructed, in which the shaded record represents the logical conjunction of the three membership rules. A reference to this credential record is included in the membership certificate.

for credential record references is designed so that references are never reused, and credential records representing facts that are false, and will always remain false, can be deleted. Figure 4.5 illustrates the data-structures for the example in figure 4.4.

4.7 Constructing Credential Record Graphs

Graphs in the credential records correspond directly to statements in RDL. Each membership rule involved in the definition of a role will be represented by a single credential record. Consider the definition of 'Member' from the previous chapter.

$$\textbf{Member}(u) \;\leftarrow\; \textbf{Login.LoggedOn}(u,h)^{*} \triangleleft^{*} \textbf{Chair} \;\; : u \text{ in } \texttt{Opera}^{*}$$

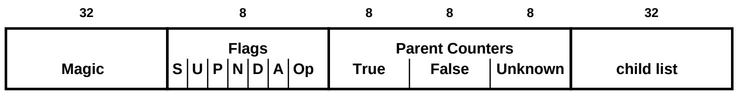
There are three membership rules for this definition:

1. The supplied logged on certificate must remain valid. The certificate contains a reference to the credential record representing this fact.[4]

2. The delegation must not be revoked. A new credential record is created to represent this fact. The delegator is given the right to delete this record.

3. The client must remain a member of the 'opera' group. Each group membership is represented by a single credential record, and membership lookup returns a reference to this as a side-effect.

[4]For the moment, we will ignore the fact that this was issued by a different service.



Flag	Description
State	Current truth value, True or False
Unknown	Value is currently unknown due to network failure
Permanent	The state will never change
Notify	Another service is using this credential
Direct Use	A certificate has been issued that embeds this credential
Auto Revoke	This credential should be revoked if a parent exits
Op	Binary operation performed on parent values

Figure 4.7: Format of a Credential Record

To create a suitable credential record to represent the truth of all three of these facts one new record must be created (for rule 2) and a second record must be created to 'and' together the records representing rules 1,2, and 3, as illustrated in figure 4.6. A small optimisation is possible in that the two new records can be combined into one fulfilling both functions. In general one new credential record is required for each (revokable) delegation, and one for each entry to a role with multiple membership rules.

In order to support role entry involving complex constraint expressions, operators other than a logical 'and' are required when combining the values of credential records. To this end, records which perform logical 'or', 'nor' and 'nand' operations on the truth values of their parents are added, and the 'not' operation is added as a distinguished parent→child reference. Although 'and' and 'not' would be sufficient to represent an arbitrary set of constraints, the other operators allow for more compact expressions, and hence fewer credential records are required.

4.8 Format of a Credential Record

The format of a credential record is shown in figure 4.7. Credential records are stored in a large table within a server. Whenever a table entry is reused, the `Magic` field is incremented, thus ensuring that (`table index`,`Magic`) forms an identifier which is unique over the life of the service. This tuple is used to form a 64 bit identifier used as the credential record reference. This method can also be used for credential records stored in persistent store, and is described in more detail in [Lo94, 6.4].

Each credential records stores a list of its children, so that when its state changes, the information is propagated to its children, which will change their value and recurse, if appropriate. Rather than storing backwards pointers from child records to parents, a more efficient scheme is used, and counters are kept of the number of parents that are 'true', 'false' or 'unknown' due to network failures. This information is all that is required to set the state of a record.

A `Permanent` flag within each record indicates if state changes are possible. Whenever a state change is not possible, for example after revocation, the record itself is redundant and can be garbage collected. A record may also be deleted if it represents a fact that is *uninteresting*, i.e. one that has not been used for issuing certificates directly, and that no longer has any child records.

Garbage collection takes place unlinking parent→child links whenever the value of a parent is made permanent. The record cannot be immediately deleted because this might leave dangling references from its parents. A periodic sweep algorithm unlinks these references, and deletes *uninteresting* and *permanent* records.

4.8.1 Credential Records for Group Membership

Credential records representing group membership can be considerably simpler than 'standard' records, as they have no ancestral dependencies, and are not used directly. However, it is essential that when group membership changes, the corresponding record can be updated. Unlike revocation, where the relevant CRR is supplied explicitly, a service managing group membership must be able to determine the identity of the related credential record whenever there is a group membership change.

It is not necessary to store a credential record representing every possible group membership, or even those that are currently valid. Instead, a hash table of 'interesting' credentials is created, indexed by (`userid`,`groupid`). An interesting credential, in this case, is one that has child records, or that is used by an external server.

4.9 Distribution Issues

In a distributed environment, certificates issued at one server may be used as credentials at another. Consequently, a credential record in one server may be required to be the parent of a record in another. This raises issues of naming, independent failure modes and robustness[Bac92]. In order to decouple the name space and failure modes of two services, *external records* are used to represent remote facts and *event notification* is used to communicate state changes between servers.

4.9.1 External Records

If a server requires a reference to a credential record on another service, it creates a local surrogate record called an *external record*. This record contains information about the identity, location and state of the record being represented, together with the standard attributes of a credential record, including an identifier within the local name space. The state of the record is maintained by event notification, as described below, and in all other respects the record is treated as a local credential record. When the local garbage collector decides the record is no longer required, the external server is informed so that it can delete corresponding state.

4.9.2 Event Notification

Asynchronous event notification is an important feature of distributed systems, and the RPC mechanism used in the current implementation of Oasis has been extended to add event management functions.[5] Oasis makes use of these functions by defining the event type **Modified**(*CRR*, *newstate*) in the interface definition file of an Oasis server. A server may then register interest in the state of a particular credential record, and will be informed if its state changes, by being sent an event with *CRR* and *newstate* set to appropriate values. In this way, revocation taking place in one server may affect certificates issued by another.

The effect of external records and event notification is illustrated by figure 4.8, which uses the example from 4.6, but highlights the distribution issues.

4.10 The Effect of Failures

Event notification between servers may be delayed indefinitely by network congestion or failure. Additionally, either of the parties may fail and restart independently. These situations must be taken into account in the design of any distributed system involving events. The approaches described here are implemented as part of a generic event library, and are equally admissible to any event based application.

Consider two parties `A` and `B`, where `A` wishes to send `B` a stream of messages. If every message `A` sends contains a sequence number, then `B` will be able to detect if any *previous* message has been lost. If in addition, `A` ensures that a message is sent at least every t seconds, then `B` will know within time t if a message has been lost or delayed.

This is the basic requirement for event handshaking. In addition, `B` must periodically inform `A` that events have been received, so that `A` may delete any associated state.

This protocol is called a *heartbeat protocol*, and a form of this is used in the event system implemented. The server responsible for signalling events is used as the initiator of the protocol and ensures that a heartbeat event is sent every t seconds. Individual events (and heartbeats) are not acknowledged for reasons of efficiency, but every i heartbeats the client replies, so that the server can detect failures and resend event instances if required.

This leads to a system with the following characteristics:

- A client can be certain of receiving an event within time t of its generation, or of detecting that notification may have failed or been delayed.
- A server can detect a client that is not responding, and after a period, can assume that it is no longer running.
- A client who processes and forwards events can treat heart-beats in a similar manner. This feature allows a service to provide guarantees about 'indirect' events from other services.

In Oasis, a missed heartbeat leads to external credential records being marked as 'unknown'. This state propagates to child records, and possibly other servers. When connection is re-established the state of each record is read and, if necessary, events are re-registered with the remote service. The period of the heartbeat and the frequency of response can be set on a per-service basis, allowing each service to choose the trade-off between failure tolerance and security.

[5] This is described in Chapter 6

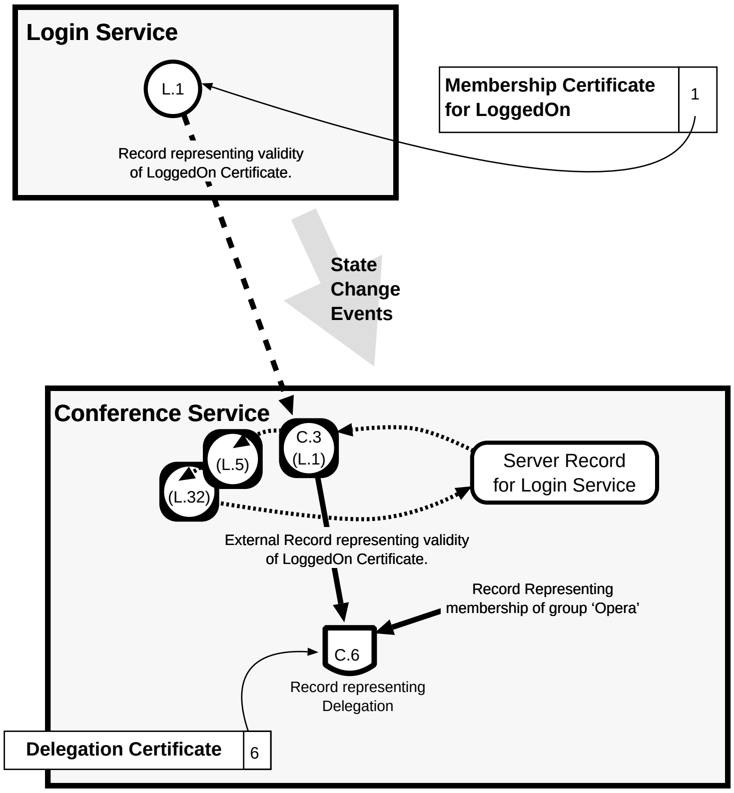


Figure 4.8: External Credential Records

In the diagram the identifiers 'L1','L5' and 'L32' are used to represent records stored in the login service, and 'C.3' and 'C.6' are used to represent records stored in the conference service. As these name spaces are managed separately, external identifiers must be mapped to internal identifiers, as is illustrated by the external record 'C.3 (L.1)', which is a local record with identifier 'C.3' that represents the record 'L.1' in the login service. Event notification is used to abstract the distribution issues from the graph-walking algorithms, and the server record stores sufficient information to allow connection to be re-established after a communications failure.

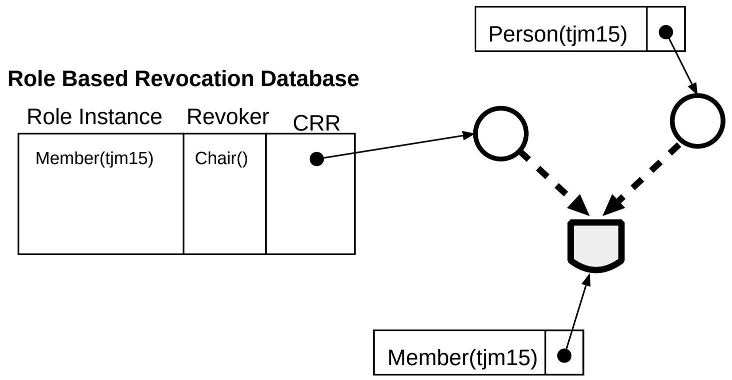


Figure 4.9: Managing Credential Records for Role Based Revocation

4.11 RDL Extensions

In section 3.3.2 a number of extensions to RDL were considered. The majority of these have a straightforward implementation, but the implementation of Role Based Revocation is more complex. To recap, Role Based Revocation is used when a client other than the delegator wishes to revoke. Indeed, there may not be a delegator at all. In these circumstances there can be no revocation certificate, and therefore no direct reference to the credential record representing the state of the delegation. Consider the example from section 3.3.2.

$$\begin{array}{lcl} \textbf{Chair} & \leftarrow & \ldots \\ \textbf{Member}(p) & \leftarrow & \textbf{Person}(p) \triangleright^* \textbf{Chair} \end{array}$$

When a client enters the role **Member** then a credential record graph with the structure shown in figure 4.9 will be created. A client wishing to revoke must supply a **Chair** certificate, and the server must locate the relevant credential record. The information required is stored in a database in the server, in records of the form

(Instance of role,revoker,credential record reference)

Once a revocation has taken place, a client must not be able to re-enter the role. A second database of revoked role instances must therefore be kept indefinitely. As a revocation of this form is 'forever', clients holding the revoker's rule are allowed to remove entries from this second database. This allows *hire, fire, re-hire* semantics.

4.12 Interworking with other Mechanisms

Role membership certificates are issued on the presentation of other role membership certificates. Clearly there must be an auxiliary mechanism, or 'bootstrapping' a client would not be possible. In general a service may issue and revoke role membership certificates for *any* reason. Role entry due to policy expressed in RDL is simply the more usual case. The loader and password services described in section 3.4.1 and 3.4.3 are both services that issue certificates based on policy not defined in RDL. Although generally used for bootstrapping, the ability to arbitrarily issue and revoke certificates is a useful mechanism, especially when inter-operating between Oasis services and those protected by legacy or alternative access control schemes.

For example, consider a system using *organisational roles*, such as **manager** and **project leader** as well as, or instead of UserIds[SCFY96]. A service could be devised that issued an equivalent Oasis role for each client holding one of these roles, and the two schemes could therefore interwork. This is facilitated by the fact that multiple name spaces are fundamental to the design of Oasis.

Perhaps strangely for an access control architecture, Oasis manages *names* not access rights. A role membership certificate is therefore a name, not a promise of access privileges. When presented to a service, the name is interpreted in order to determine the level of access to be given. This extra indirection allows legacy systems to be converted to Oasis services. For example, an NFS file server could be amended to accept Oasis role membership certificates and extract a client's user identity and group memberships from it. It could then apply its own access control measures based on this name. This fits within the Oasis model of server behaviour. In addition, as was demonstrated in section 3.3.3, this check can be represented as a set of RDL statements thus simplifying reasoning about interworking between the two systems.

4.13 Auditing and Accounting

A disadvantage of capability schemes is that it is not possible to list all the clients who currently have access to an object, or determine how that access was authorised. In Oasis, each interaction between a client and a service, or a client and another client, must take place with the service's knowledge and consent. For example if client A elects client B as a member of a meeting, then the server managing the meeting will be involved in the delegation process, and in any subsequent revocation. The server may record these requests and this information can be used to answer queries about current clients or when performing other auditing tasks.

A similar approach may be taken to accounting. Each role membership certificate can trivially be extended to include the identity of the account that should be charged for any resources used. Indeed, the notions of roles and accounting go hand in hand. This work simply notes that sufficient 'hooks' are in place to allow auditing and accounting information to be gathered, and does not consider these issues in detail. Further discussion of these management issues can be found in [Neu93, OMG94, 3.2.3].

4.14 Conclusions

In this chapter we have presented a mechanism for enforcement of access control and naming policies written in RDL. In particular a scalable mechanism to allow *rapid* and *selective* revocation of capabilities was presented. Reasoning about policy interaction and possible attacks is aided by a direct implementation, and by the fact that strong guarantees can be made by an issuing service about the length of any undetected failure. It is difficult to quantitatively evaluate the implementation as there are few architectures offering similar semantics. It is certainly clear that if there is little or no revocation, then the background activity is likely to be less than that found in other schemes where capabilities must be continually refreshed (for example [LABW94]). This is primarily due to the event driven nature of credential updates and the increased scope for caching of cryptographic checks.

Chapter 5

The MSSA

5.1 Introduction

This chapter presents an overview of The Multi Service Storage Architecture (MSSA). This is a complex distributed storage system which has been used as a test application in the development of Oasis.

Section 5.2 gives a brief overview of the MSSA, and the original access control mechanism proposed in [Lo94]. Section 5.3 considers the issues for access control specification in the MSSA in more detail, and discusses the extent to which the original scheme and other approaches meet these requirements. Section 5.4 proposes a model of *shared acls* as an aid to policy expression and complexity management. Section 5.5 discusses the enforcement of access control, and relates Oasis certificates to the original MSSA capability scheme. Section 5.6 considers possible optimisations and section 5.7 concludes.

5.2 A Brief Overview of the MSSA

The Multi Service Storage Architecture (MSSA), is a networked storage service that manages a number of different types of files. The architecture is open in the sense that new services may be added, and servers may be mutually distrustful. Clients of the MSSA may be users, databases or entire operating systems, and this leads to complex access control requirements.

The MSSA consists of three levels of server. *Byte Segment Custodes* are responsible for physical storage of data. They mask device specific details and provide a standard interface for use by *File Custodes.* These provide an open interface for the storage of particular kinds of data. For example a *flat file custode* stores regular files, a *structured file custode* stores structured data, and a *continuous media custode* stores continuous medium data, such as audio or video. Each file custode provides a different interface, but common schemes for naming, accounting and access control allow integration of services. In particular, each file is named with a machine oriented unique identifier, that may be examined to locate the (file) custode responsible for it. A feature of the structured file custode is that files stored on it may contain references to files on other custodes, allowing complex compound documents to be stored.

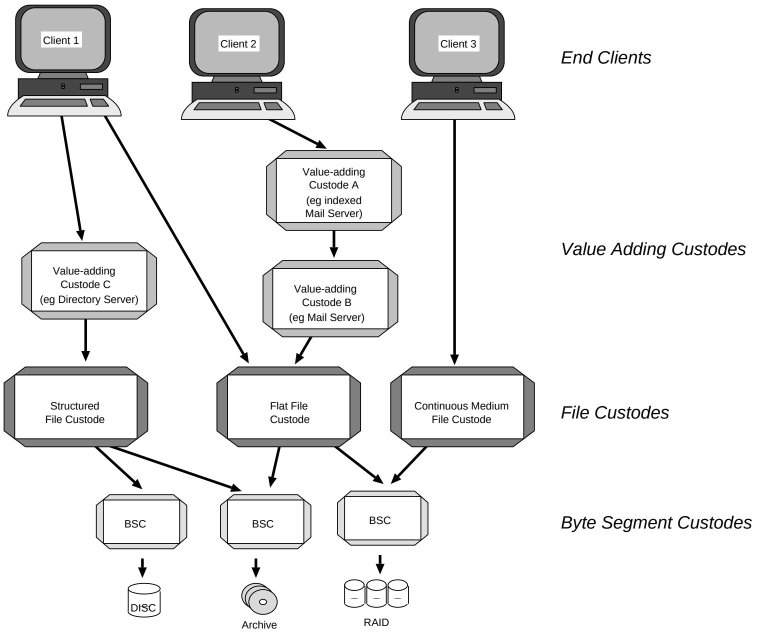


Figure 5.1: Access Paths Within the MSSA

Value Adding Custodes (VACs) form the third layer of the MSSA. These appear to clients as 'standard' file custodes, but are implemented by abstracting the interface of file custodes or other value adding custodes. Value adding custodes are a powerful feature of the MSSA, as they allow rapid integration of new services in a secure and efficient manor.

File custodes provide strongly typed interfaces, and other file or value adding custodes can provide interfaces which are sub-types of these in an object oriented sense. For example an indexed flat file custode will provide all of the operations of a file custode, and in addition will provide lookup operations based on indexing information. An overview of the MSSA is shown in figure 5.1 and a detailed description can be found in [Lo94].

5.2.1 The Original MSSA Access Control Scheme

The MSSA has complex access control needs. Naturally, all client access to a custode must be authenticated. As the MSSA is a layered architecture, and each level is distrustful of the levels above it, a client access may involve a check at each of several levels. A naive solution to this problem would be dreadfully inefficient, so an extensible capability scheme [BMLH94a] was derived to reduce the overhead. As capabilities have many disadvantages as a basis for policy control, MSSA capabilities were used merely as a fast mechanism, and an access control list scheme was added to allow policy expression.

Derivation of this scheme was undertaken jointly with Sai Lai Lo, and is presented in his thesis [Lo94]. Some of the advantages and shortcomings of this scheme led directly to the design of Oasis, and in this chapter the re-application of these ideas to the MSSA is discussed. The term MSSA will be used to refer to the original MSSA extended with the new access control scheme, and the term 'original MSSA' will be used when it is necessary to emphasise a difference.

5.3 Access Control Issues

In this section we consider a number of access control issues that are poorly addressed in the majority of storage systems. In the following section we propose a mechanism to address these issues.

5.3.1 Grouping Files

Although a storage system may consist of many hundreds of thousands of files, in general there will be considerably fewer distinct sets of access rights. Files may be grouped and given the same access rights. For example an individual may group files as 'private', 'public' or associated with some project. Grouping files in this way makes management easier, and can provide considerable performance benefits, as access control information can be cached. Existing storage systems generally provide access control grouping by overloading this function onto some other grouping mechanism.

Grouping by directory

In a traditional filing system, this function is controlled by a directory service. In a hierarchical scheme, access to a file is only possible provided that a client has sufficient rights to access all of the enclosing directories. This feature may be utilised by a user by creating suitable 'public', 'private' and 'project' directories, with associated ACLs.

However, if the files maintained within a storage service have a number of different types, it is unclear how access control information associated with a directory can be extended to cover all the possible operations on the contained objects. For example *flat files* require protection of `read` and `write` operations, whereas continuous media requires protection of `play` and `record`. Although for some file types, the rights can be cast to `generic read` and `generic write`, this is not always the case. A bank account has operations `deposit`, `withdraw` and `query balance`. These clearly do not fit 'read/write' semantics.

In some storage services, including the MSSA, it is argued that the directory service should not be an integral part of the system [BN80] - as its functionality is not required in some circumstances, for example when storing databases, or data from persistent programming languages. It is also argued that the type of directory service should be data dependent. Mail is likely to be classified and indexed, but this 'directory' system is unlikely to resemble that used for program development[DO85]. If the directory service is not an integral part of the storage system, then it cannot be wholly responsible for access control.

Grouping by structure

Emerging systems such as OLE [Bro94], make considerable use of structured documents, where one file contains references to other files, or the embedded contents of a file of a different type. Structured files are also commonly used to store databases, or other structured information. Within the MSSA, the structured file custode is responsible for managing such structured files.

It is often a requirement that the access control information associated with different parts of a structured file is the same, or that users who may access one part may also access another. Although this grouping mechanism appears suitable for access control, complications arise if a sub-object is shared between several objects. In addition a secondary mechanism is required, as there are likely to be many more structured objects than access control groups.

Explicit Grouping for Access Control

Moffett [MST90] proposes grouping of protected objects solely for provision of access control. These groups are called *domains*. Domains may be nested, and an object can be a member of more than one domain. Access rules are maintained for each domain, and indicate how clients may access (any) objects within the domain. Although domains are an appropriate mechanism for file grouping, and the nesting of domains gives powerful semantics, the system is also complex and difficult to understand. For example, if an object is a member of two domains, the access rules may contradict. Nested grouping structures were considered for MSSA access control, but were rejected on the grounds of complexity and computational cost.

The MSSA approach

In the MSSA, files are grouped into *containers* for accounting purposes. In the original scheme, a container was also used as the basis of access control. Each container had an ACL, and this was combined with an optional per-file ACL, to indicate the level of access allowed.

Although this approach removes the overloading of access control information with directory or structural information, it is inflexible and difficult to maintain. For the scheme to be effective, files that are to be grouped for access control must either exist within their own private container, or be treated individually. As containers are relatively heavyweight the second option is usually taken, and the semantic and efficiency benefits of file grouping are lost.

5.3.2 Meta-Access Control

Within a storage system, it is not only the files themselves that must be protected. Control over the access control lists is also important, and a policy for examination and modification of ACLs must be specified.

The notion of file 'owner' is most commonly used to specify control of ACLs. In Unix filing systems, for example[RT78], the file owner is the only client who may modify an ACL, but anyone with sufficient access to the enclosing directory may read it. This is restrictive for shared files, and unnecessarily high read access is a potential security risk. In general, there may be many users who may modify an ACL, and many more who may examine it. The ACL itself is an object, like any other, and therefore best protected by a second ACL. The pragmatics of ACLs for ACLs seem at first infeasible. In the following section a mechanism for *shared ACLs* is proposed as a mechanism for file grouping.

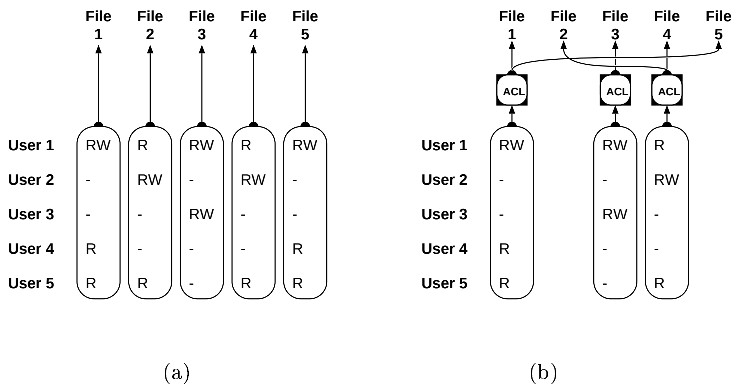


Figure 5.2: The Access Matrix

(a) The traditional ACL approach to storing an Access Matrix.
(b) In the new approach, files are logically grouped.

Section 5.4.2 describes how this technique can be extended to allow meta-access control without the risk of infinite recursion.

5.4 Shared ACLs

Access control is often expressed in terms of an Access Matrix [Lam71]. In ACL systems the columns of this matrix are stored as access control lists within each file, and there is no conceptual file grouping (Figure 5.2a). In the MSSA under Oasis, each ACL is used to protect a set of files, allowing files to be logically grouped, and access control uniformly applied to them all (Figure 5.2b).

Each ACL is given a meaningful name, and users may manipulate access control information, either by changing which ACL is used to control a file, or by modifying the ACL itself. For example, files relating to the Empire project may be controlled by an ACL called 'Empire Private'.

5.4.1 ACLs as Objects

Although the primary function of this grouping is to improve the semantics and usability of ACLs, a useful side effect is that the number of ACLs required is drastically reduced. This allows them to be efficiently treated as objects in their own right. Each ACL is stored as a file on a suitable custode, and each file the ACL protects contains an embedded reference to this file. As ACLs are themselves files, they also have ACLs to control them, and so any policy for meta-access control can be implemented. Figure 5.3 gives an example system.

5.4.2 Recursive ACL Checks

When a custode accesses an ACL to determine if a client operation is valid, then that custode must have sufficient rights to read the ACL.[1] If the file is stored

[1]Note, custodes should not (and do not) trust each other.

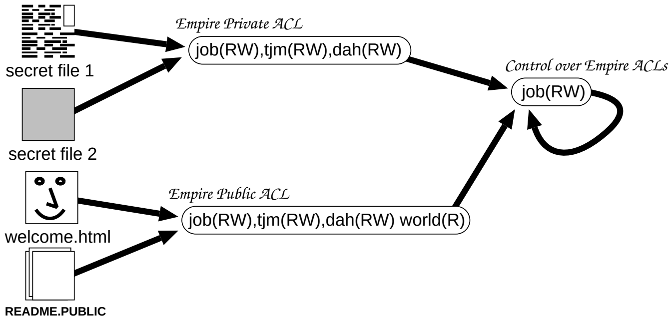


Figure 5.3: An Example Using Shared ACLs

In this diagram, seven files are protected; four regular files and three ACLs. The two central ACLs effectively define a group of users, and their associated access. User `job` has control over membership of this group, and also has control over his own position.

in the same custode, this is not an issue, but for remote ACLs a long chain of access checks may be required. This would be extremely inefficient. Indeed a cycle of ACLs could lead to infinite checks, as illustrated in figure 5.4.

A simple solution to this would be to give custodes special privileges to read ACLs. However this was rejected, as an important feature of VACs is that they need not be trusted by the underlying system, and hence can be freely used without the worry of introducing security loopholes.

To reduce the inefficiency of recursive checks, and to prevent the problems of cyclic checks, a constraint is placed on the use of ACL files as follows.

The ACL file protecting an ACL file must reside in the same custode.

This constraint restricts access checks to at most one remote call, and avoids the problems caused by cyclic checks (Figure 5.5). As there are typically hundreds of files per access control list, this overhead may be reduced still further by server caching.

5.4.3 Relating ACLs to Rolefiles

Each MSSA ACL file is represented by a single rolefile containing definitions for two roles. **UseAcl**(*rights*) controls access to the files governed by the ACL. The file identifier of the ACL corresponds to the rolefile identifier, and a **UseAcl** certificate is therefore specific to a particular ACL. Whenever a file is accessed, and a **UseAcl** certificate is suppled, the server must determine which ACL the file is protected by and then use this information when validating the certificate. The second role **UseFile**(*file*, *rights*) is file specific and is used when delegating access to a particular file.

Generally each rolefile will consist of a single ACL rule relating users to rights, together with 'standard' statements defining allowable delegation. To simplify definition of these rolefiles, a simple ACL may be given instead of the full rolefile, and this is merged with a 'policy template' during parsing.

In addition to this, all rolefiles are merged with a set of standard statements, which typically allow access by system administrators. This is preferable to a

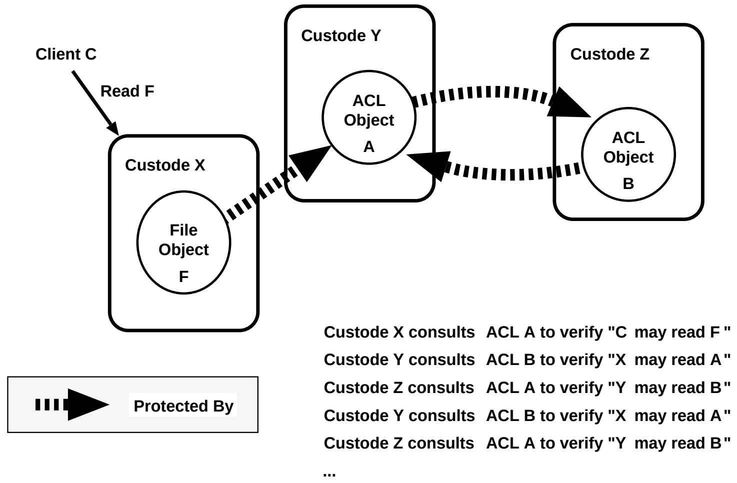


Figure 5.4: Cycle Checking of ACLs

An attempt to access file F will result in an infinite cycle of checks.

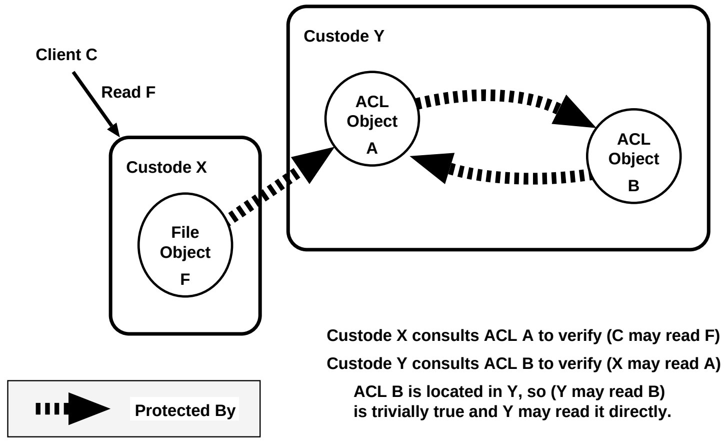


Figure 5.5: Cycle Checking of ACLs with Constrained ACL Placement

Although there is logical cycle between the ACLs, the placement of the files ensures that an access check rapidly terminates.

'root' identifier, as it allows finer grain control over administration, and removes the need for an additional mechanism.

5.4.4 ACL format

As discussed in section 3.3.3, any form of ACL could be used within an MSSA ACL/Rolefile. Indeed, different ACLs could use expressions of different formats. However, for the sake of clarity it was decided to choose a 'standard' ACL format for use in the MSSA, and this is also the preferred format for other Oasis services requiring ACLs.

The primary decision to be made was how *conflicting* ACL statements should be interpreted. For example the ACL

```
Bob(Read/Write), student(Read)
```

is ambiguous in the case that `Bob` is a student. Should he gain read and write access or just read access? The majority of existing systems choose *most closely binding* semantics, whereby only the entry directly referring to `Bob` is used, as this is more specific to the user Bob than the `student` entry. In more complicated situations, such as when there is no individual entry for a user, and a user is a member of two or more groups, different systems have different semantics.

For example, in Phoenix/MVS [Doc], access rights are ordered, and the 'highest' rights are given when multiple group entries match. In Andrew [Sat89], there are two forms of entry, those granting rights, and those denying them. When more than one entry matches, the union of all 'negative' rights is subtracted from the union of all 'positive' rights. A fuller discussion of different policies is given in [Lun88].

A common failing of early ACL systems was that it was not possible to restrict the access available to a class of user, without also defining what access was allowed. For example the rule 'Students may not have write access' is clearly different from 'Students may have (only) read access'. Such difficulties are overcome by the addition of *negative* or *restrictive* rules. These rules limit the actions of other rules, but do not in themselves define allowable access. Such a scheme is used in NT [Pow93], and was developed independently for use in the MSSA.

In this scheme the entries within an ACL are ordered, and the rights given to a client are formed by examining each entry in turn and applying any part that does not conflict with an earlier entry. More formally,

> Two sets are created, the rights to be granted, **G** (initially empty), and the set of possible rights, **P** (initially full). Each entry that matches the client is consulted in turn. Assuming this involves rights **R**:
>
> - A negative entry will restrict the possible rights granted
>
> $$\mathbf{P} \Leftarrow \mathbf{P} - \mathbf{R}$$
>
> For example if **P** is $\{\texttt{read}, \texttt{write}\}$ and a negative entry specifies write access is to be disallowed, then **P** will be reduced to $\{\texttt{read}\}$.

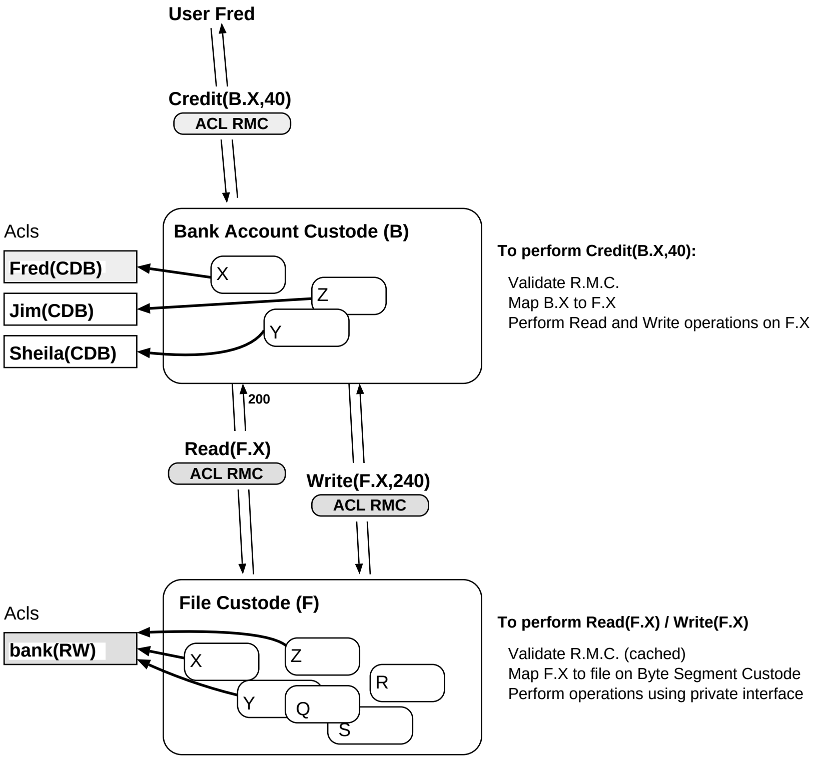


Figure 5.6: Access Paths for a VAC Operation

A client operation such as 'Credit' on a file in a value adding custode will be validated by ensuring that the supplied role membership certificate contains sufficient rights, and relates to the correct ACL file (shaded). For the Value adding custode to perform corresponding actions on the file at the server below, it will use a certificate relating to the ACL for all of its files.

- A positive entry will grant the rights indicated, providing this does not contradict an earlier negative entry.

$$\mathbf{G} \Leftarrow \mathbf{G} \cup (\mathbf{P} \cap \mathbf{R})$$

This process continues until all matching entries have been consulted. The client is then granted the rights indicated in $\mathbf{G}$.

This scheme is considerably more expressive than systems involving a fixed priority between entries of different types. It is also considerably clearer as there are no 'difficult cases'.

5.5 Enforcing Access Control

Although ACLs are used to represent access policy, actual file accesses take place using role membership certificates issued by the custode controlling the file. In the case of files managed by value adding custodes, the data is stored on lower level custodes, and access to a file will involve one or more corresponding accesses to the level below. This is illustrated in figure 5.6. As the files controlled by each VAC are generally all protected by the same access control list, each VAC need store only one role membership certificate for use at the level below, and it is likely that the validation of this will be cached.

In the original MSSA scheme, ACLs were not shared, and in general, the value adding client would have required one capability per file. Storage of these capabilities was not feasible, and this lead to the derivation of a multi-level capability scheme[BMLH94b]. In the new scheme, such complications are unnecessary.

5.5.1 Effect of Compromise

When a certificate is issued, it is protected by a signature derived from a secret stored in the issuing server. In theory, only a single secret need be stored in each server, and a signature check based on this would be sufficient to guarantee the integrity of all certificates. However reliance on a single secret is dangerous, as discovery of this would allow unlimited access to all files.

The approach taken in the MSSA (and some other Oasis servers) is to maintain a *rolling table* of secrets. Periodically, a new secret is generated, and certificates are issued using it. Certificates issued using the older secrets may still be validated, but as these certificates eventually time out, the secrets have a limited lifetime. In order to compromise the system, an attacker must not only discover the value of the secret, but must do so during its lifetime. The probability of a secret being discovered is linked to the duration of its use, and the number of certificates issued. By tuning the rate of secret generation based on these parameters, the integrity of the system is ensured.

5.5.2 Volatile ACLs

In all protected systems, access policy may change and it is important that access tokens, such as certificates, that relate to outdated policy are revoked or timed out. File systems in particular tend to have volatile policies, as access control lists may be frequently modified.

In the original MSSA, each file custode maintained one secret for each file it controlled. When an ACL was modified, the secret associated with the corresponding file could be changed, and capabilities issued using the ACL would be revoked. In the new scheme, ACLs are shared, and a per-file secret would therefore be of no help when performing revocation due to policy change. Instead we need per-acl state. In keeping with the Oasis philosophy, this is accomplished using credential records. A credential record is associated with each access control list in use. This record represents the validity of certificates issued based on the contents of the ACL. When an ACL is changed, the credential record is deleted, and another created to represent the new ACL state. Certificates issued using the old version of the ACL will therefore be revoked using the standard mechanisms described in chapter 4.

In both schemes, a certificate may be revoked when it represents a valid role membership. Client applications must therefore be able to cope with such 'non-fatal' revocation, and request replacement certificates transparently. It should be noted that a delegated client need only re-apply to the server, not to the

```
FFC: INTERFACE =

IS COMPATIBLE WITH FileCustode;

BEGIN

read :  OPERATION [cert : Certificate;
                   ssid : SSID;
                   start: Cardinal;
                   len  : Cardinal    ]
        RETURNS   [data : Bytes       ]
        RAISES AccessDenied;

write:  OPERATION [cert : Certificate;
                   ssid : SSID;
                   start: Cardinal;
                   len  : Cardinal;
                   data : Bytes       ]
        RETURNS []
        RAISES AccessDenied;

Other FFC Operations
END.
```

Inherits standard operations such as creation, deletion and attribute checks.

```
IFFC: INTERFACE =

IS COMPATIBLE WITH FFC;

BEGIN

lookup: OPERATION [ query: String;
                    cert : SEQUENCE OF
                           Certificate
                  ]
        RETURNS   [ hits : SEQUENCE OF
                           SSID
                  ];
END.
```

Inherits all operations from FFC

Flat File Custode Interface   Indexed Flat File Custode Interface

Figure 5.7: Sub-typing of Interfaces

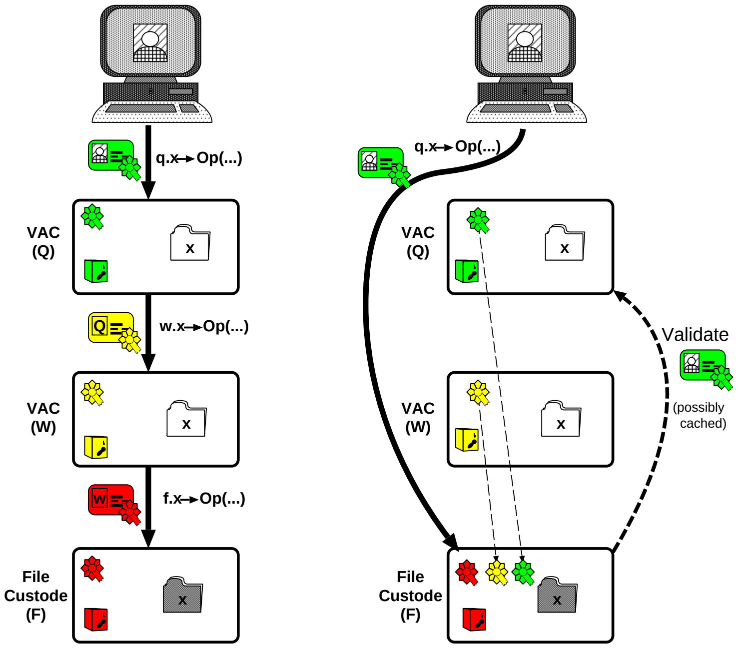
The operations for the indexed file custode are a superset of those for the flat file custode. In particular the IFFC Read operation is implemented by passing the request to the FFC without modification. The IFFC need not implement this operation at all, but instead directs the client to call the FFC directly. This leads to more efficient access, and a considerably simplified IFFC.

electing client(s), as delegation certificates themselves will remain valid. This is significant, as the elector may no longer be present. This is an improvement over the original MSSA delegation scheme, where there was no mechanism for refreshing delegated capabilities.

5.6 Optimising Access

During the design of value adding clients, it was noted that many of these perform a specialisation of the custode below them. For example an indexed flat file custode provides search operations in addition to read and write. It was also noted that many value adding custodes only modify some of the supported operations, whilst others are passed through to the level below without modification. For example, read operations on the indexed flat file custode are passed directly to the custode below.

It was proposed that such unmodified accesses could be *bypassed* around the value adding custode, as it takes no functional part in the operation. This is facilitated by the organisation of custodes using sub-typing similar to that in object oriented programming languages[BMLH94c]. Figure 5.7 gives the interface to the Flat File and Indexed Flat File custode as an illustration. In the original MSSA multi-level capability scheme, at all levels other than the lowest, access checks consisted purely of computation, and not access to stored state. These computations could be safely moved to a different custode without com-



(a) (b)

Figure 5.8: Bypassing One or More Custodes

If bypassing does not take place, each custode checks the credential supplied by the one above it using its stored secret and credential record (a). If the operation may be bypassed, the client may call the 'bottom' custode directly. A callback is made to the top level to check the supplied certificate. If a credential changes and this leads to the client's certificate, or the bypassing route, becoming invalid; then the 'bottom' custode will be informed by event notification. Note that the callback validates a cryptographic check and this may be cached.

promising security. The custode could therefore by bypassed and a significant optimisation could be made for some operations. Indeed, for the majority of custodes, operations such as the examination of standard attributes need not be specialised and can be bypassed in this way. Experience with existing filing systems suggests that these operations make up a large percentage of the total, and this optimisation is therefore significant[HKG+88].

In the new scheme such an optimisation is not quite so straight forward. The mechanisms for the revocation of capabilities, and the mechanisms for certificate checking rely on the issuer performing the check. However, bypassing can still take place, with the proviso that a callback is made to the top level of the 'stack of custodes' to validate the capability. This is never *less* efficient than a straightforward call down the stack, and in the majority of cases, where caching of credential checks has taken place, this is considerably more efficient. This is illustrated in figure 5.8.

5.7 Summary

The MSSA is a complex distributed storage service, with correspondingly complex security requirements. As such, it is a useful test bench for the abilities of Oasis. The original design of the MSSA access control system was tuned for flexibility and efficiency, and was itself an impressive scheme. What then has Oasis added? Semantically, it has clarified many of the 'meta' issues, relating to how capabilities are gained and lost, especially in the light of changes to policy (e.g. modification to access control lists). In particular the issues about who may change this policy, and how changes effect existing clients have been clarified.

More importantly, Oasis has allowed reasoning about the interworking of the MSSA access control scheme with other applications and services. For example it is now possible to indicate explicitly that the members of a meeting are the only people who may read the file used to store the minutes. This encourages the use of high level naming schemes and reduces the likelihood of forgotten updates. The alternative would be to explicitly list the members of a meeting on the ACL for the minutes file. If someone was ejected from the meeting the ACL would have to be manually updated. This not only poses a security risk, but complicates auditing, as the *reason* why a client is allowed access to a file would not be recorded.

In term of efficiency, the new scheme is certainly no less efficient, and given the event driven nature of updates, rather than continual refreshing of capabilities, it is believed that there will be a small performance benefit. In addition to semantic benefits, the use of shared ACLs reduces the storage overhead required for managing access control policies, and reduces the total number of capabilities in use. This in turn enables more effective capability caching.

Chapter 6

Events

6.1 Introduction

Many classes of service provide information in the form of asynchronous callbacks. In this chapter we will consider the class of callbacks that represent information about some *event* that has occurred. We will identify special characteristics of such callbacks which allow them to be treated in a uniform way, thus simplifying the design of secure, robust distributed systems.

Events have already been touched on in chapter 4, where they were used for credential record coherency control. In that application, the robustness of event notification was of paramount importance. Later in this chapter an *Active Badge System* will be used to illustrate other advantages of our event system, such as patten matching and composite event notification. Section 6.2 gives an overview of our event architecture, emphasizing the approach to distribution and naming. The Active Badge System [HHB93] was chosen as a suitable testbed for event programming techniques. Section 6.3 gives an overview of this, and presents an alternative, event-based approach. This is used in later sections as a motivating example. Section 6.4 describes how event monitoring applications, such as those required by the badge system, may be handled by a composite event service. A specification language is derived and examples given in the following sections. Section 6.12 summarises the chapter, and draws some conclusions.

6.2 An Event Architecture

6.2.1 Event Classification

In our architecture, events are specified using an extended RPC interface definition language (IDL). Like RPCs, events are of distinct types, and may have a number of parameters of simple or complex types. This combination of RPC and Event specification simplifies naming issues, and allows existing Trading mechanisms[APM93] to be used for the location of event servers of a particular type. In addition, it is common for services to combine both RPC and event interfaces, which simplifies the passing of parameters between the two domains. For example the interface to a print server is shown below.

```
Printer: INTERFACE =
BEGIN
 Print : OPERATION [ file : STRING ]
                   RETURNS
                   [ jobno : CARDINAL ];

 Finished: EVENT   [ jobno : CARDINAL ];
END.
```

This defines a service interface in terms of (typed) operations that may be performed on an instance of a service conforming to this interface, and gives the types of events that may be generated by an instance of a service conforming to the interface. In addition, as the interface definition contains events, the interface will automatically inherit standard event operation, such as **Register** and **Deregister**. These will be defined in the following section.

The IDL file is preprocessed to produce client and server *stubs* for the operations, and to create constructors and destructors for the events. The event constructor creates a *generic event object* representing an instance of the event of the given type. This may then be signalled or manipulated using a set of standard utilities that need not be aware of the concrete type of the event. The destructor unmarshalls an instance of an event of a given type, and returns its original arguments. The constructor and destructor for the 'Finished' event above are as follows

```
Event *Printer_Finished(int jobno)
```

and

```
Decode_Printer_Finished(Event *e,int *jobno)
```

6.2.2 Registration and Notification

In order for a client to be notified of the occurrence of an event, it must have first *registered interest* in the event with the issuing service. The alternative scheme is for all events to be broadcast to all possibly interested parties. Although suitable for a centralised system, such an approach clearly does not scale to a distributed environment. Registration has other benefits. Monitoring need not take place continuously — but only when a client is interested, and for many events (such as the example above) registration can take place as a side effect of some other operation.

Registration takes place in two stages. Firstly, a client registers with the service establishing a session, and supplying a delivery address for event notification. During this process, the client may provide various credentials, and *admission control* based on these will take place. This will be discussed in detail in chapter 7.

Once a session has been established, the client may register interest in event occurrences of different types. In order to reduce the number of *uninteresting* events that the client is notified about, the client supplies an *acceptance expression* to filter interesting events from uninteresting ones. The format of this expression is significant. A complex acceptance expression might reduce the number of uninteresting expressions that are notified, but it is less likely that the event server will be able to derive useful information to enable it to restrict the amount of unnecessary monitoring. In addition, complex expressions are more difficult to create automatically, for example by a composite event service (described later).

The form of acceptance expression chosen is an *event template*. This is an instance of an event with a number of wild card parameters. (cf. query by example [Zlo77]). For the printer example, if a client were to register with a particular print service, use of the template

Finished(27)

would lead to notification when job 27 was complete, whereas registration using

Finished($*$)

would lead to notification whenever that service completed any job. If a client is interested in **Finished** events generated by *any* print server, then they must explicitly register with each server.

Templates are a simple, but powerful mechanism. In particular, they are amenable to automatic generation, which is required for composite event detection. This issue will be discussed when composite events are considered in section 6.4.

6.2.3 Library Support

As events are marshalled into *generic event objects*, a library of utilities can be created to manipulate them. In particular, event services, such as composite event servers and event multiplexers, need not understand the concrete type of the event instances they manipulate. Libraries have been created to deal with both client and server action on event notification, template matching, event trading and failure tolerance. In addition, the client and server libraries interact using a secure protocol based on Oasis roles. This will be considered in detail in the following chapter. Figure 6.1 outlines the action of a client submitting a job to a printer and awaiting the event indicating its completion. Note that the client application, server instance and stubs require knowledge about the concrete type of the event, but that the client and server libraries do not.

6.3 The Active Badge System

Active Badges are small electronic name tags that periodically broadcast their identity using a simple infra-red protocol. Sensors placed in different rooms within a building receive these signals and inform a central database. This may then be consulted to determine the current location of a badge, and hence the location of its owner.

Active badges were developed by Olivetti Research Limited, and badge systems are currently being used in a number of sites including several departments within The University of Cambridge, and sites as remote as Xerox Parc. There are several versions of active badge software in use in different sites, each optimised for different purposes. A general failing is that communication of 'foreign'

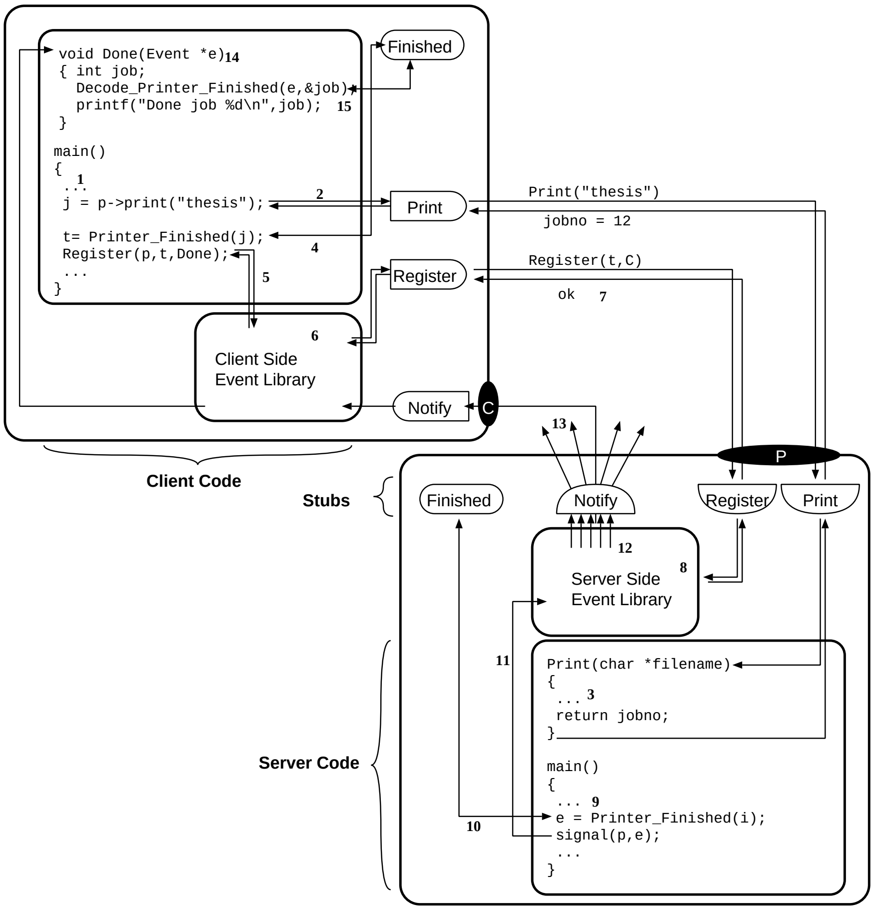


Figure 6.1: A Detailed Example

Consider a client wishing to print a file "thesis" on a particular printer, and then wait for the job to complete. First, the client locates the interface for the print server (1). The client then sends an RPC to the server (2). This is marshalled by the client stubs, sent over the network to the server, unmarshalled and eventually the server routine 'Print' is invoked (3). When this completes, the job number is returned in an analogous fashion. Next, the client creates a template for the event indicating completion of the job, by using the constructor for 'Finished' (4). The local event library is then called to register interest in events that originate from **P** and match this template. The client indicates that the procedure 'Done' should be called whenever a matching event occurs (5). The library contacts the server and establishes a session (not shown). It then registers interest in events matching the given template, and supplies a call-back interface **C** (6,7). The server side event library adds this registration to a list it maintains (8).
At some time in the future, the print server will complete the job and decide that an event should be signalled (9). The server creates a generic event to represent the completion of the job by using the constructor for 'Finished' (10). This event is signalled to the server side library (11). The library consults its database of requests and informs all interested clients (12). Note that the server library can perform this matching without knowing the concrete type of the event. When the illustrated client receives the event (13) the client library consults stored state, and passes the event to the Done method (14). The event is decoded using the destructor for 'Finished' (15) and the result is printed.

badge sightings is poor: a badge from organisation X is useless in organisation Y — despite the fact that badge identities are globally unique, and the same IR protocols are used on all sites.

As an inherently distributed, and event based, application, a 'global' badge system was chosen as a suitable example application for the development and demonstration of event based programming techniques. In addition, location information maybe extremely sensitive, and the complex access requirements of the badge system form a good testbed for the development of access control models for events.

6.3.1 A Scalable Approach

In designing a 'global' badge system, it is clear that there can be no central database of badges. Instead, each site (or organisation) must be responsible for maintaining information about its own badges. A protocol must be designed to allow this information to be exported to other sites — when a badge is seen elsewhere. Each site should have relative freedom with the design of its own badge system, and in particular must decide on the degree to which it publishes badge movements.

When the (physical) badges were designed, each was given a small amount of memory designed to store a 'pointer to home'. This may be interrogated by a sensor to determine the badge's home site. The global badge system makes use of this information, and when a previously unknown badge is sighted, its home site is informed, and in return that site returns naming information related to the badge. There is no way to detect when a badge leaves a site, but when it is seen elsewhere, this information can be used to delete unnecessary information from other sites. Figure 6.2 illustrates this protocol. The information that a badge has moved site may also be of use to other clients or servers. For example, an application watching a particular user will need to know when that user moves site.

For this reason, the movement is signalled by an event of the form

$$\mathbf{MovedSite}(badge, oldsite, newsite)$$

This information is used by remote servers, to detect when a badge has left their site, and is also available to client applications.

6.3.2 Movements Within a Site

There is considerable freedom in the design of a badge system *within* a site. In this section a badge system is described that splits the functionality of the badge system in two. *Monitoring* is performed by a process called the Master. This interfaces with the sensors, and signals badge sightings directly as events of the form $\mathbf{Seen}(badge, sensor)$. *Naming* is performed by the Namer. This is responsible for mapping badge and sensor identifiers to User and Room names, as well as performing the inter-site protocol and managing updates.

The namer must be informed of the arrival of badges from other sites. As the Master does not support this function directly, an intermediate service called the 'Sighting Cache' maintains a list of current badges, and signals when a new one is seen. Figure 6.3 illustrates the architecture.

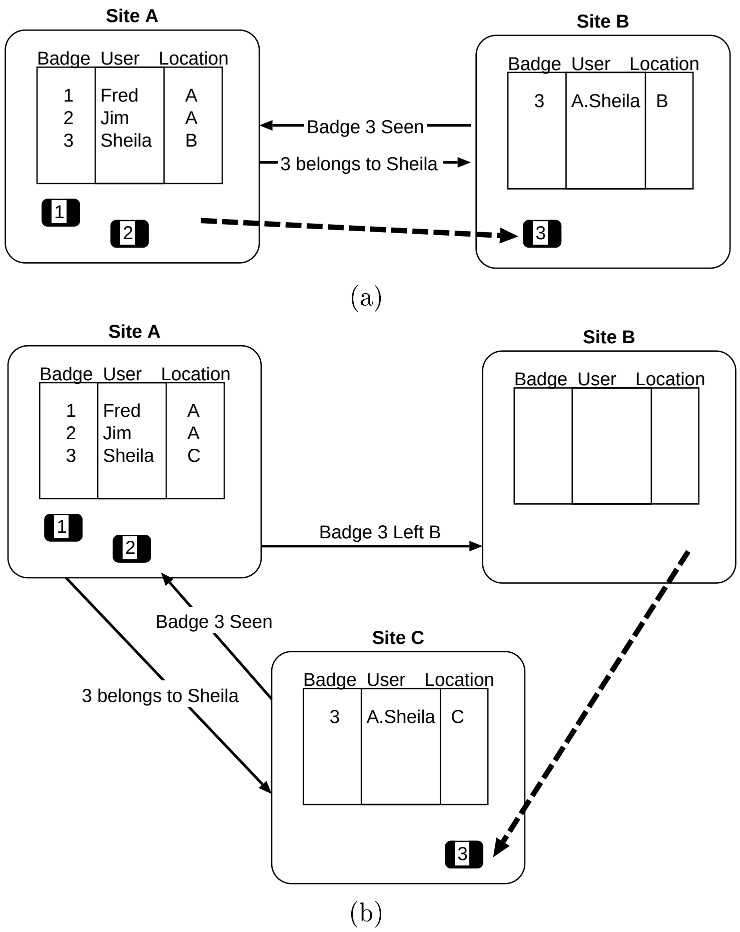


Figure 6.2: Badge Movements Between Sites

In (a) a badge based at site A is seen in site B. In (b) the same badge is seen in site C. Note how the 'home' site of each badge always knows of its location, and that the naming information at B is deleted when no longer required.

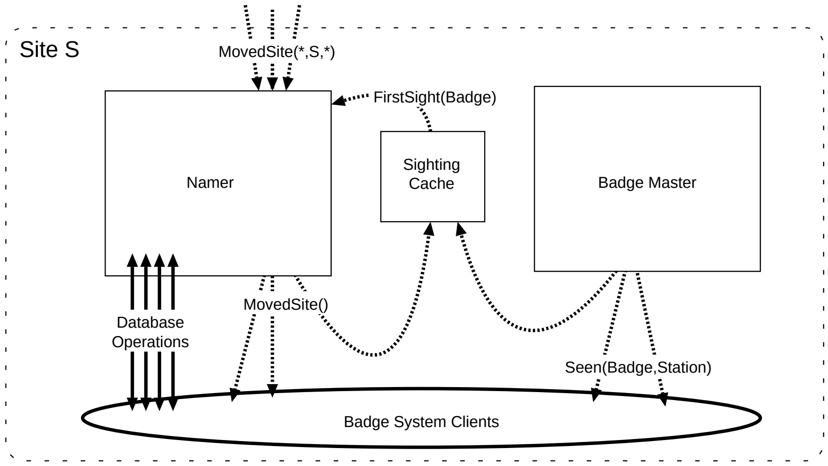


Figure 6.3: Intra-site Badge Events

6.3.3 Namer Lookups

The namer is primarily an *active* database[WC96]. It stores a number of simple relations, and in addition signals events when the database changes. Monitoring applications will generally wish to discover the badge associated with a particular user, and will then contact the Master to watch for sightings of this badge.

The Namer database is not static. It is often useful to change the badge associated with a user — for example if the batteries are flat, or the badge is lost. Long running monitoring applications need to be made aware of these changes. In keeping with the event philosophy, the solution chosen is to make the database active by signalling updates as events.

A simple monitoring application to watch user `rjh21` would then perform the following:

1. Lookup `rjh21`'s badge
   $\texttt{Select}\ b\ \texttt{from}\ \mathbf{OwnsBadge}(u, b)\ \texttt{where}\ u = \texttt{rjh21}$
2. Register Interest in changes of `rjh21`'s badge ownership
   $\texttt{Register}\ \mathbf{OwnsBadge}(\texttt{rjh21}, b)$
3. Register Interest in movements of badge b.
4. Whenever b is seen: ....
5. Whenever $\mathbf{OwnsBadge}(\texttt{rjh21}, b)$ occurs, restart from stage 3.

This sequence of operations suffers from a race condition. If the database changes between the lookup and the registration of interest in state changes, then this event will be overlooked. Although such race conditions can be avoided by careful coding, the combination of related Lookup and Register operations is so common, that it is advantageous to treat it as a special case, and allow registration and lookup to take place *atomically.* A special combined form of Lookup and Register is therefore used for such database events.

$$\mathbf{DBRegister}(\textit{event template}, \textit{callback}) \rightarrow \textit{event}\ldots$$

This will return all existing database entries matching the template, in the form of events, and in addition will register interest in updates resulting in additional matching entries being created. This feature is deceptively powerful, and examples of its use within the context of composite expressions are given in section 6.6.

6.4 Composite Events

Clients of event services typically perform monitoring functions. For example, monitoring the location of a person, or counting the number of people in a meeting. Monitoring applications are generally interested in particular sequences of events, rather than the occurrence of a single event.

Although each application could be designed in an ad-hoc manner, it is advantageous to design a general purpose composite event recogniser. There are several existing architectures for composite event recognition[GJO92, GD93, CKAK94]. However these were generally designed for centralised event systems and are unsuitable for distributed event recognition. In this section we consider the issues for distributed composite event detection, and in particular highlight the similarities and differences with more common pattern matching mechanisms. In section 6.5 we will consider the design of a composite event language.

6.4.1 Distributed Time

In a distributed environment time is the key consideration. There is, in general, no *global clock* and care must therefore be taken when comparing time stamps on two events from different sources. Distribution also leads to *variable delay.* If we are interested in determining which of the two events `A` and `B` occurred first, we cannot rely on the order in which we are notified, but must examine the events' time stamps. Equally, we cannot conclude that an event has not occurred, until we have waited to see if notification has been delayed, or determined this by other means. Together with the increased scale and complexity of a distributed environment, these features make it difficult to determine a *global view* of the system. In general we can only ensure that we have received all events up until time t after we have received events from the most delayed event source; i.e. at time $t + \Delta_{\mathrm{worst}}$.

Evaluation models that require a global view will therefore have an inherent delay of Δ_{worst}. Unfortunately the majority of existing composite event detection schemes rely on such a simplifying assumption. An important motivation in the design of a composite event detector for distributed events was to reduce this requirement.

For example, in the badge system, we may wish to detect sightings of two users Giles and Roger, in order to detect if Giles ever enters a room in which Roger is situated. This could be represented by the expression

```
Whenever Seen(Roger,x)
   Repeat
      If Seen(Giles,x) then Signal Together(x)
   Until Seen(Roger,y)
```

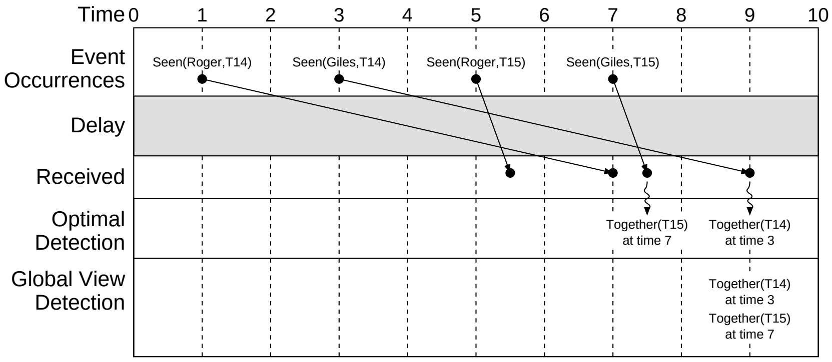


Figure 6.4: The Effect of Delay on Composite Event Detection

Consider the situation in which Roger and Giles are first seen in room T14, and then seen in room T15. If the events from the badge sensor in room T14 are delayed, we do not want this to disrupt the detection of the second meeting. A possible trace of events and detection is shown in figure 6.4. As can be seen an optimal detector processed events in the order in which they are received, and would detect the second meeting first, but a 'global view' detector, which must process event in the order in which they occur, would block and (eventually) detect the first meeting first. Note that both evaluations ultimately return the same results.

A secondary issue is that by enforcing a global view, we must create a total order on all event instances. The operators in our language require only pairwise comparison of (some) event time stamps and this lesser requirement is easier to justify in an environment where clocks are not completely synchronised.

6.4.2 Regular Expressions

Composite event detection is a form of pattern matching, and regular expressions are an obvious formalism for describing composite events. In this section we will highlight several important areas in which event detection differs from regular expression matching, and derive the features required by a composite event language.

Parameter Matching

As events in our environment are more complex than symbols in a regular language, additional expressive power is required. In particular, an instance of an event has a number of parameters. Instantiation and matching of parameters are essential parts of composite event detection. For example, in the badge application, all sightings are signalled using the **Seen**(*badge*, *room*) event. An application that monitors for two people entering the same room might use a composite event

$$\textbf{Seen}(\texttt{Badge 12}, r_1)\ldots\textbf{Seen}(\texttt{Badge 15}, r_2)$$

Where the variables r_1 and r_2 should match, to indicate that the badges are in the same room. It should be noted that the value assigned to r_1 and r_2 is not known in advance, and variables must be instantiated *during* evaluation, not simply before evaluation takes place.

The language for expressing regular expressions may be extended by adding variable instantiation, and expressions in this new form may be converted back to standard regular expressions, providing the range of each parameter is known.[1] We call the mapping between a set of variables and the associated values the *environment* for the evaluation. As evaluation takes place, variable instantiation will update the environment. The expression $\mathbf{A}(x)$ may simply be replaced with $\mathbf{A}(\mathtt{i})$ if $x = \mathtt{i}$ in the current environment. If x is uninstantiated, then $\mathbf{A}(x); \mathbf{E}$ expands to

$$(\mathbf{A}(\mathtt{x}_1); \mathbf{E}_1)|(\mathbf{A}(\mathtt{x}_2); \mathbf{E}_2)| \ldots |(\mathbf{A}(\mathtt{x}_n); \mathbf{E}_n)$$

where $\mathtt{x}_1 \ldots \mathtt{x}_n$ are the possible values of the parameter x, and $\mathtt{E}_1 \ldots \mathtt{E}_n$ are the expressions formed when the substitution $x = \mathtt{x}_i$ is made into $\mathtt{E}$.

Care must be taken with the expansion of the Kleene star. It is tempting to expand $\mathbf{A}(x)^*$ to

$$\mathrm{null}|\mathbf{A}(x)|\mathbf{A}(x); \mathbf{A}(x)|\mathbf{A}(x); \mathbf{A}(x); \mathbf{A}(x)| \ldots$$

However, this definition is restrictive. Consider $\mathbf{A}^*$. This is true for a sequence of zero or more $\mathbf{A}$'s. However, using the expansion given above, $\mathbf{A}(x)^*$ is only true for a sequence of zero or more $\mathbf{A}$'s *with the same parameter*. The expression 'a sequence of zero or more $\mathbf{A}$'s with any parameter' cannot be expressed. Moreover, this is a particularly useful expression. For example to express 'whenever a person enters a room...' we need to match a sequence of **Enters()** events with different parameters. This requirement lead to the derivation of a new operator 'Whenever' (\$) which is similar to the Kleene star but which allows a different assignment in each repetition. For example $\$\mathbf{A}(x)$ is equivalent to

$$\mathbf{A}(x)|\mathbf{A}(x'); \mathbf{A}(x)|\mathbf{A}(x''); \mathbf{A}(x'); \mathbf{A}(x)| \ldots$$

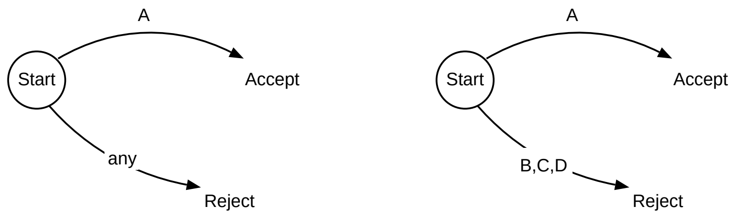
Open Environment

Regular expressions are defined over a finite 'closed' alphabet of symbols. We wish to define event expressions in an open environment where there are both an (effectively) infinite number of event types and an (effectively) infinite number of different event 'symbols' due to instantiation of variables. A regular expression is an expression which indicates which sequences of symbols from a given language are acceptable, and rejects all other sequences of symbols. For example, the regular expression $\mathbf{A}$ may be represented by the finite state machine shown in figure 6.5a. For a similar effect in an open environment we must indicate which symbols make up the language of 'interesting events' at each point in the evaluation. This is illustrated in figure 6.5b.

Although this process appears cumbersome, the semantics are very natural. For example the event expression for 'Fred is seen in a room and then Mary enters before he leaves' is

$$\mathbf{Enter}(\mathtt{Fred}, r) \text{ followed by } \mathbf{Enter}(\mathtt{Mary}, r) \text{ before } \mathbf{Leaves}(\mathtt{Fred}, r)$$

[1] This exercise is useful for formal reasoning, but does not aid implementation.



FSM for regular expression
A
(a)

FSM for event expression
A before **B**,**C** or **D**
(a)

Figure 6.5: Regular Expressions with Explicit Alphabet

In (a) the regular expression **A** succeeds only for input **A** and fails for all other inputs. In a distributed environment the language must be specified explicitly as 'any' is meaningless (b).

which is more natural than the equivalent regular expression

$$\begin{array}{r l} x = & \{\texttt{Room 1}, \texttt{Room 1}, \ldots, \texttt{Room n}\} \\ \text{alphabet} = & \{\mathbf{Enter}(\texttt{Fred}, x), \mathbf{Enter}(\texttt{Mary}, x), \mathbf{Leaves}(\texttt{Fred}, x)\} \end{array}$$

$$\mathbf{Enter}(\texttt{Fred}, r); \left(\text{any } \textit{except } \begin{array}{c} \mathbf{Leaves}(\texttt{fred}, r) \\ \mathbf{Enter}(\texttt{Mary}, r) \end{array}\right)^{*}; \mathbf{Enter}(\texttt{Mary}, r)$$

An additional advantage is that by specifying the alphabet at each point in the expression, the number of events in which interest must be registered is kept to a minimum. This point will be considered in more detail in section 6.7 where implementation is discussed.

6.5 A Composite Event Language

In this section we derive a composite event specification language, based on the modifications to regular expressions described above. This is of equal expressive power to regular expressions but has the advantages of clarity and use of parameters. In addition it has an efficient evaluation machine (described in section 6.7) and may be trivially extended to deal with operational criteria, such as network delay, that might lead to the misordering of events.

A composite expression is constructed from *base event templates* using the set of operators defined below. The parameters to these templates may be literals or variables. An evaluation Φ of a composite expression C is defined in terms of the start time of the evaluation s, and an initial set of variable bindings given in an environment E. An evaluation returns a set of tuples of (*occurrence time*, *environment*). *occurrence time* is the time at which the composite event triggers, and *environment* is a set of (*variable name*, *value*) pairs. A composite event may trigger many times, with the same or different occurrence times. The operators were constructed so that evaluation of different parts of an expression can take place *independently*. That is to say, delays affecting evaluation of one sub-expression need not affect the evaluation of different parts of the expression. This reduces the need for a global view, and helps reduce the impact of delay or network failure. This was aided by the implicit specification of the alphabet of 'interesting' symbols for each sub-expression, as discussed in section 6.4.2 above.

The evaluation function Φ is defined as follows

$\Phi(T)$ A base event.

The first time at which a base event matching the template T occurs after s.

$\Phi(T, s, E) = \{(t, E')\}$ where t is the time stamp of the first base event which matches T after time s.

A base event **B** matches a template **T** if it is of the same type as **T**, and each parameter in **T** is either a literal equal to the matching parameter in **B**, or a variable that is either undefined in the environment, or defined in the environment and has a value equal to that matching parameter in **B**. The updated environment (E') is the environment unioned with all variables in **T** and their associated values from **B**. This is similar to the base case for regular expression, with the alphabet set to symbols matching **T**. A naive implementation would be to register interest in **T** and then wait for a matching event to occur. However in a distributed environment the delay associated with registration may lead to an incorrect evaluation. This problem, and a solution, is discussed in section 6.8.1.

$\Phi(C_1\text{-}C_2)$ The 'without' operator. C_1 occurs without C_2 having occurred first.

C_1 and C_2 are independent and may be evaluated independently. If events matching both C_1 and C_2 occur, the order must be determined. If the difference in time stamps is smaller than the clock drift between the sources of the events, then this is a difficult problem. For the moment, we note that time stamp order will always give the *most probable* order, and in section 6.8.4 we return to this issue and discuss how stronger statements about event ordering may be made.

$$\Phi(C_1\text{-}C_2, s, E) = \{(t, E') : (t, E') \in \Phi(C_1, s, E) \wedge \neg(\exists t_1, E''_{s<t_1\leq t} \text{ s.t. } (t_1, E'') \in \Phi(C_2, s, E))\}$$

$\Phi(C_1; C_2)$ The 'sequence' operator, C_1 followed by C_2

As C_1 and C_2 evaluate to sets of occurrences, this is defined as the union of all occurrences of C_2, starting at the time stamp of each occurrence of C_1. Note that sequence does not imply 'immediately following', as in regular expressions. This removes the need to register interest in events other than those involved in C_1 and C_2.

$$\Phi(C_1; C_2, s, E) = \bigcup_{(t,E')\in\Phi(C_1,s,E)} \Phi(C_2, t, E')$$

$\Phi(C_1|C_2)$ The 'inclusive or' operator.

This evaluates to the union of all occurrences of C_1 and all occurrences of C_2. Note that C_1 and C_2 may be evaluated independently.

$$\Phi(C_1|C_2, s, E) = \Phi(C_1, s, E) \cup \Phi(C_2, s, E)$$

$\Phi(\$C)$ The 'whenever' operator.

This is the replacement for the Kleene star, as discussed in section 6.4.2. The intuition behind 'whenever' is that a new evaluation is started each time the previous one completes. Often it may be read as 'for each'. For example evaluation of $(\$\mathbf{Enter}(x)); C_2$ creates a separate evaluation of C_2 each time an **Enter**(`i`) event occurs, each with a (potentially) different value assigned to x. These evaluations are independent and may be evaluated separately.

$$\Phi(\$C, s, E) = \Phi(C, s, E) \cup \bigcup_{(t,E') \in \Phi(C,s,E)} \Phi(\$C, t, E)$$

$\Phi(\text{null})$ The null event

This trivial event is included as it is required in order to prove equivalence between event expressions and regular expressions.

$$\Phi(\text{null}, s, E) = \{(s, E)\}$$

Care must be taken with the combination of 'whenever' and 'null'. In all cases, the *least* solution is taken. For example 'whenever null' ($\text{null}) is defined as the least solution to

$$\Phi(\$\text{null}, s, E) = \Phi(\text{null}, s, E) \cup \bigcup_{(t,E') \in \Phi(\text{null},s,E)} \Phi(\$\text{null}, t, E)$$

i.e.

$$\Phi(\$\text{null}, s, E) = \{(s, E)\}$$

6.5.1 Side Expressions

In the language above, event templates are matched by events with matching parameters. Occasionally though, we require a more complex matching rule. For example we may require that a parameter should not be equal to a certain value, or that there is an algebraic relationship between variables.[2] These requirements are met with the use of *side expressions* analogous to those found in formal methods. For example

$$\begin{array}{rl} \mathbf{Seen}(x, y) & \{x \neq \texttt{rjh21}\} \\ \mathbf{Withdraw}(z) & \{z > 500\} \end{array}$$

Side expressions may also be used to perform variable assignment, and are useful for decreasing the complexity of expressions.

[2]This only applies for integer types

6.6 Examples

In this section we give a number of examples of composite events defined using the new language. The first four of these are from the badge system, the last was initially given as an example for an alternative composite event language designed by Gehani [GJO92].

Enters(B, R)
Leaves(B, R)

Trigger whenever someone enters/leaves a room.

In order to detect someone entering a room, they must first be seen in one room and then be seen in a *different* room.[3]

$$\left(\$\mathbf{Seen}(B, R')\right) ; \left(\mathbf{Seen}(B, R) - \mathbf{Seen}(B, R')\right)$$

In the above expression, we start a new evaluation each time a badge is seen. When the badge is next seen, another evaluation starts, and the previous one either accepts or fails depending on the room the new sighting is in. 'Whenever' is the most closely binding operator, and 'Sequence' is the least closely binding, so the parentheses given above are not required. The expression for **Leaves** is identical, except that the old location is signalled, rather than the new one.

$$\mathbf{Leaves}(B, R') = \$\mathbf{Seen}(B, R'); \mathbf{Seen}(B, R) - \mathbf{Seen}(B, R')$$

Together(A, B)

Detect when two people are in the same room.

This is similar to the example above. A first attempt is

$$\$\mathbf{Seen}(A, R); \$\mathbf{Seen}(B, R) - \mathbf{Seen}(A, R')$$

Note the second whenever. The expression can be read as "Whenever a person is seen, signal *whenever* another person is seen in the same room.". In this expression, the order in which the badges are seen is significant. If evaluated with an initial environment in which A or B is defined, this may not give the intended semantics. A more robust solution is

$$\begin{array}{ll} & (\$\mathbf{Seen}(A, R); \$\mathbf{Seen}(B, R) - \mathbf{Seen}(A, R')) \\ \mid & (\$\mathbf{Seen}(B, R); \$\mathbf{Seen}(A, R) - \mathbf{Seen}(B, R')) \end{array}$$

Trapped(P)

By combining events from the badge system with those from a fire alarm service, we can signal all people who are seen after an alarm is raised. This can be done by waiting for an alarm, and then signalling each badge sighting before the all clear. We make use of the fact that database lookup can be modelled as an event, in order to determine the name associated with each badge.

$$\$\mathbf{Alarm}(); (\$\mathbf{Seen}(B) - \mathbf{AllClear}()); \mathbf{OwnsBadge}(B, P)$$

[3] For simplicity we assume one sensor per room.

Trapped2(P) A refinement of the above example is to trigger only *once* for each person still in the building one minute after the alarm. In this example, a side expression is used to set a variable to the time after which badge sightings should be signalled. @ is a special variable indicating the occurrence time of the last event.

$$\$\mathbf{Alarm}()\{t \Leftarrow @ + 60\}; \mathbf{AbsTime}(t);$$
$$\$\mathbf{OwnsBadge}(B, P); \mathbf{Seen}(B)$$

The power of treating database lookups as events is made clear in this example. $\$\mathbf{OwnsBadge}(B, P)$ will create one evaluation for each tuple in the database matching this template; i.e. one per person. Interest will be registered in each person's badge, and the composite event will trigger the first time a badge is seen.

EndOfPoint() This is a nice example of a complex, but easily understood composite event, that was first published by Gehani [GJO92]. The aim is to detect the end of a point in the game of Squash (Racket Ball). The events of interest are $\mathbf{serve}(i)$, when player i serves; $\mathbf{hit}(i)$ when player i hits the ball; and **floor**,**wall** and **front** when the ball hits the floor, side wall, or front wall respectively.

A point starts when the ball is served. It ends when one of the following occurs:

- After the serve, the ball fails to hit the front wall first.

  "Trigger if the ball hits something other than the front wall first."

  $$(\mathbf{floor}|\mathbf{wall}|\mathbf{hit}(i)) - \mathbf{front}$$

- After the ball hits the front wall, a player fails to hit it before it bounces twice.

  "Whenever a ball hits the front wall, trigger if it hits the floor twice before being hit by a player."

  $$\$\mathbf{front}; (\mathbf{floor}; \mathbf{floor}) - \mathbf{hit}(i)$$

- After the ball is hit, it hits the floor or is hit again before it hits the front wall.

  $$\$\mathbf{hit}(i); (\mathbf{floor}|\mathbf{hit}(j)) - \mathbf{front}$$

- The players fail to take turns.

  Trigger if the server hits the ball before the other player, or if one player hits the ball twice in a row

  $$(\mathbf{hit}(s) - \mathbf{hit}(i)\{i \neq s\})$$
  $$|(\$\mathbf{hit}(i); \mathbf{hit}(i) - \mathbf{hit}(j)\{j \neq i\})$$

The full expression is the disjunction of these cases, and is started each time there is a serve.

$$\begin{aligned}\$\mathbf{serve}(s)(((&\mathbf{floor}|\mathbf{wall}|\mathbf{hit}(i)) - \mathbf{front})\\ &|(\$\mathbf{front}; ((\mathbf{floor}; \mathbf{floor})|\mathbf{front}) - \mathbf{hit}(i))\\ &|(\$\mathbf{hit}(i); (\mathbf{floor}|\mathbf{hit}(j)) - \mathbf{front})\\ &|(\mathbf{hit}(s) - \mathbf{hit}(i)\{i \neq s\})\\ &|(\$\mathbf{hit}(i); \mathbf{hit}(i) - hit(j)\{j \neq i\}))\end{aligned}$$

There is however, one problem with this expression. When a point is ended, it is likely that more than one of the above conditions will hold, and the end of the point will be therefore be signalled multiple times. A solution is to add a 'reset' event, similar to the 'All Clear' in the fire alarm example. However, we would prefer a mechanism to signal the first matching event that does not require additional infrastructure. Unfortunately this cannot be specified in the language given thus far. We require an *aggregation function* that can take a set of occurrences and map them to a single occurrence. This requirement will be considered in section 6.9.

6.7 Implementation

The composite event language was designed to have an efficient implementation using a 'push down' finite state machine. An expression is represented by a sequence of states, where each state evaluates a sub-expression using a sub-machine. Each of the operators in the language can be directly represented by a state, and there is therefore a straightforward correspondence between an expression and the machine used to evaluate it.

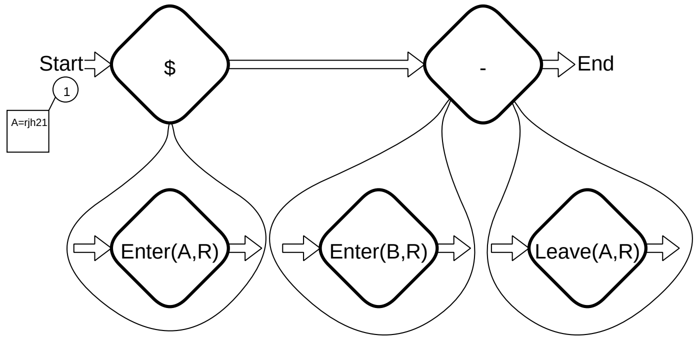
An evaluation consists of a 'bead' traversing the machine. Each bead carries with it an environment, and beads may 'split' when appropriate to give branching evaluation. Beads are evaluated independently and this removes the need for a global view of event instances. However, the relationship between beads is maintained, so that they may be destroyed when no longer required. When a bead enters a state it is 'pushed down' to the sub-machine and leaves a record behind detailing its current environment. During evaluation of the sub-expression the bead may be destroyed, or may split any number of times. When any of these beads return, the stored environment is examined to determine if the bead should proceed or if some other action should be taken. For example when a bead enters the state for $\mathbf{A} - \mathbf{B}$, it will split and one bead will be sent to each sub-machine. When a bead from $\mathbf{A}$ returns, it will only proceed if no bead from $\mathbf{B}$ has arrived (or may arrive) with an earlier time stamp.

In the rest of this section an extended example is given to illustrate how evaluation takes place. The example chosen is a simplified version of 'detect when two people are together'. The event expression used is

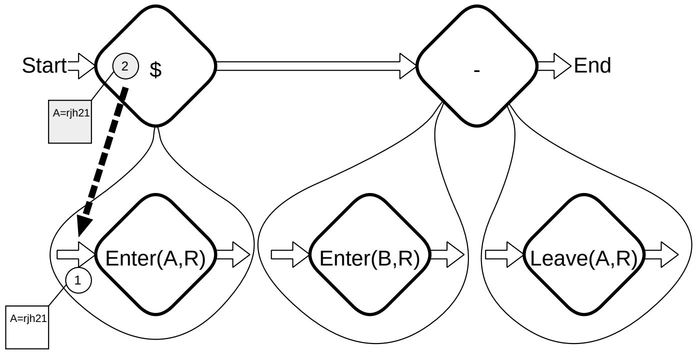
$$\$\mathbf{Enter}(A, R); \mathbf{Enter}(B, R)\text{-}\mathbf{Leaves}(A, R)$$

This may be represented by the sequence of two states, one representing the sub-machine for $\$\mathbf{Enter}(A, R)$ and the other for $\mathbf{Enter}(B, R)\text{-}\mathbf{Leaves}(A, R)$.

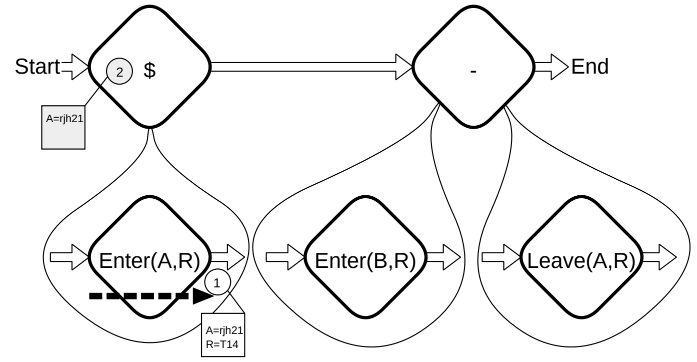
Each of these states is evaluated by a push-down to a machine for the corresponding sub-expression. Consider evaluation of this expression with the initial environment $\mathbf{A} = \texttt{rjh21}$. This will be represented by a single bead about to enter the first state, as illustrated below.



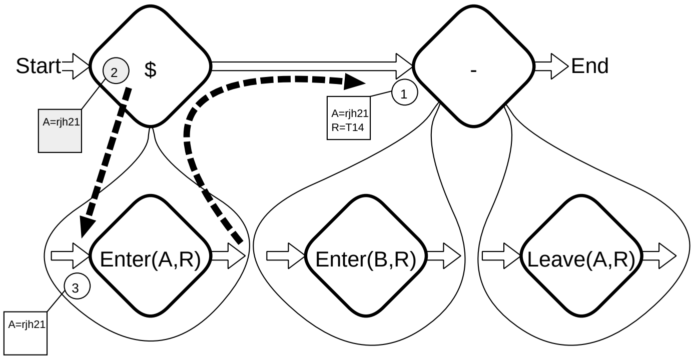
When the bead enters the first state, it will be pushed down to the sub-machine for $\mathbf{Enter}(A, R)$. In order to allow for future evaluations, a copy of the bead is left behind. As the bead has entered a state containing an event template, this template is merged with the bead's environment, and interest is registered in events matching the resulting template; i.e. $\mathbf{Enter}(\texttt{rjh21}, R)$.



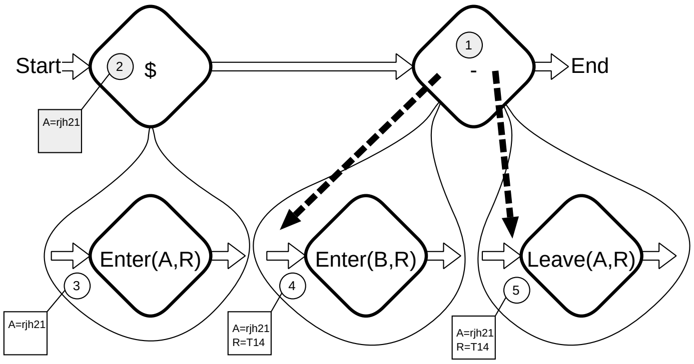
When a matching event occurs, such as $\mathbf{Enter}(\texttt{rjh21}, \texttt{T14})$ the evaluator will be notified. All beads waiting for this event will then advance and their environments will be updated. In this example there is only one bead (bead 1).



As bead 1 has now completed the sub-machine, it returns to the level above and moves to the next state. The semantics of 'whenever' are that when one evaluation completes, another should start. The stored copy of the bead is therefore copied and starts evaluation in the sub-machine, ready for the next room that `rjh21` enters.



Bead 1 has now arrived at a 'Without' state. In order to evaluate this two independent evaluations are required, so the bead is split, and one copy is sent to a sub-machine for each of the expressions. Interest is registered in $\textbf{Enter}(B, \texttt{T14})$ and $\textbf{Leaves}(\texttt{rjh21}, \texttt{T14})$.



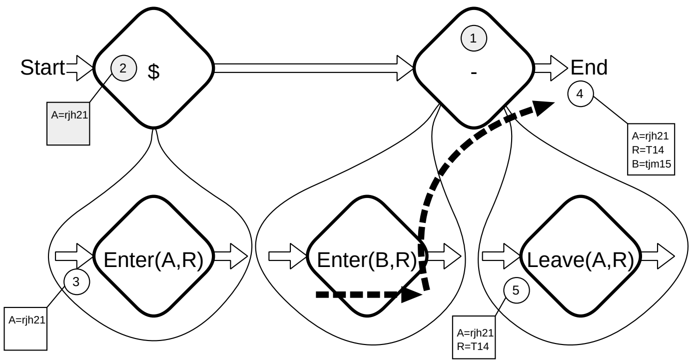
At this point three beads are active. Bead 3 is waiting for $\textbf{Enter}(\texttt{rjh21}, R)$, bead 4 is waiting for $\textbf{Enter}(\textbf{B}, \texttt{T14})$ and bead 5 is waiting for $\textbf{Leaves}(\texttt{rjh21}, \texttt{T14})$. There are also two inactive beads. Bead 1 is used to represent the relationship between beads 4 and 5, so that one can be destroyed when the other completes. Bead 2 is simply a stored environment, ready to spawn a new bead when bead 3 completes.We will consider two possibilities, firstly a second person entering `T14` and secondly `rjh21` leaving `T14`. No other events of interest can occur.[4]

Someone enters the room

If another person enters room `T14`, an event will be generated that matches the template bead 4 is waiting for. For example, if $\textbf{Enter}(\texttt{tjm15}, \texttt{T14})$ is signalled, bead 4 will advance and return to the level above and rejoin bead 1. The semantics of 'without' are that this bead may continue[5] and it will therefore reach the accepting state for the whole machine, and the composite event will trigger. Beads 1 and 5 will be deleted.

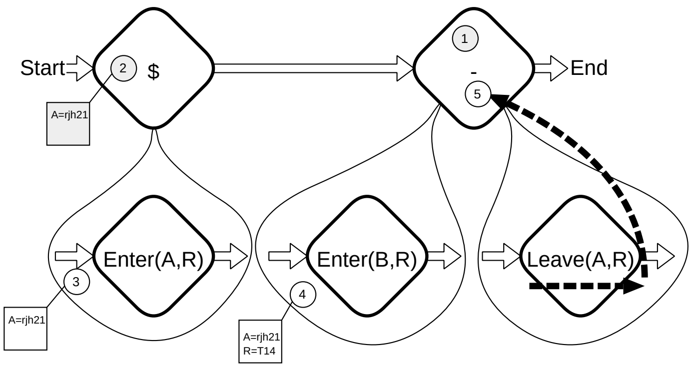
[4]Assuming `rjh21` cannot enter one room without leaving the other first.

[5]For simplicity, we assume that the evaluation of bead 5 has not been delayed. This issue is considered in section 6.8.2.

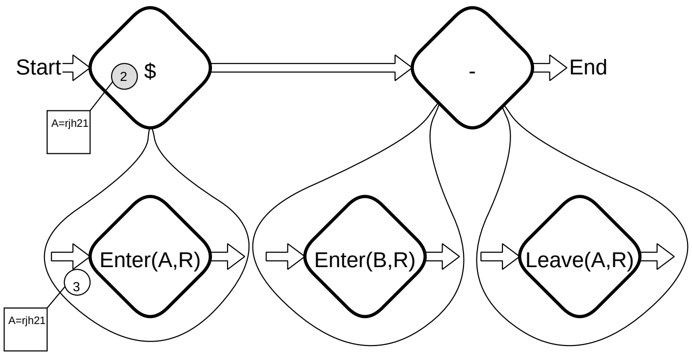


rjh21 leaves the room

If `rjh21` were to leave `T14`, then the event **Leaves**(`rjh21`, `T14`) would be detected. This would lead to the advance of bead 5 which would rejoin bead 1.



The semantics of 'without' are that once the second bead completes, evaluation has failed, and so all three beads (1,4,5) will be deleted.



Evaluation has not terminated, as bead 3 is still active, and awaiting an event signalling that `rjh21` has entered a new room. As can be seen, multiple simultaneous evaluations can take place each represented by a 'string' of beads. For example, if the same expression was used with an empty initial environment, then a bead would be generated for each person who entered a room. In this way multiple simultaneous evaluations can be represented extremely efficiently. Only events that are truly of interest are ever registered, and as beads are linked there is no need for searching or other 'expensive' operations.

6.8 Distribution Issues

In section 6.4 we indicated that distribution issues lead to the desire for a new composite event language. In the previous sections we explained how these issues can be taken into account in the design and implementation of such a language. In this section we consider other distribution issues, and how these can be tackled.

6.8.1 Registration Delay

In the implementation of the sequence $\mathbf{A}; \mathbf{B}$, interest is initially registered in events matching the template **A**. When notification of an event matching this is received, the bead advances and interest is registered in events matching **B**. Clearly there will be a delay between the occurrence of **A** and interest being registered in **B**. If any events matching **B** occur during this period, they will be missed, leading to an incorrect evaluation.

A naive solution is to initially register interest in both **A** and **B**, and to buffer information about events matching **B** until it is required. However in general there is insufficient information available to make this a reasonable approach. For example, in a simple badge application, **A** might be $\mathbf{OwnsBadge}(\texttt{rjh21}, b)$ and **B** might be $\mathbf{Seen}(b, s)$. Clearly if $\mathbf{Seen}(b, s)$ were registered before the value of b were known, then many events relating to badges that did not belong to `rjh21` would also be notified. This is clearly inefficient, especially if a large proportion of the notifications are unlikely to be relevant (as is the case in the badge system).

The solution adopted is to allow clients to use a two stage registration process. A client may *pre-register* interest in an event that they will be interested in in the future, and matching events will be recorded, and stored at the issuing server — but not immediately notified to the client. When the client is ready, it specifies a time in the past at which it wishes to *retrospectively* register. Matching events between the registration time and the current time are then notified immediately.

In the above example, a client would initially register interest in events matching $\mathbf{OwnsBadge}(\texttt{rjh21}, b)$ and pre-register interest in events matching $\mathbf{Seen}(b, s)$. When a matching **OwnsBadge** event occurs (say at time t), the value of b will be know and the client can then retrospectively register interest in events matching $\mathbf{Seen}(b, s)$ from time t. In this way, no event notifications can be lost, and correctness is ensured.

Clearly pre-registration may incur a considerable cost on the issuing service, and in general a service will only be willing to buffer events for a relatively short period, after which they will be discarded. This period should be sufficient to allow for 'reasonable' network delay in the notification of earlier events, but short enough to prevent starvation of resources. In addition, a client may repeatedly *narrow* the pre-registration as more parameters to matching events become known.

The advantages of pre-registration are three fold. Firstly, unnecessary network traffic is avoided. Secondly, as event occurrences are buffered at the source rather than the client, they may be shared by clients with similar requests. For example in the badge system, the Master buffers recent sighting information for all badges. This information is used for retrospective registration, and pre-registration incurs no additional per-client overhead. Finally, this approach allows the ultimate in flexibility. If a server is unwilling to buffer occurrences for long enough, or a client cannot afford the later registration delay, then the client can for-go pre-registration and register early.

6.8.2 Detecting Event Absence

In order to implement the 'without' operator, we must be able to detect if and event has *not* occurred. For example. evaluation of the expression $\mathbf{A} - \mathbf{B}$ involves registering interest in both **A** and **B**, and signalling if an **A** event occurs first. If an **A** event is received, the client must be certain that a **B** event has not also occurred — and been delayed — before signalling.

A naive solution is to assume a global view. For example [SR90] proposes that all event notifications should be buffered for a period greater than the anticipated maximum delay. This allows re-ordering of out of sequence events, so that **A** and **B** will always be received in the correct order. [MS95] proposes an improvement, whereby only particular events are delayed — those of particular importance or that originate on less reliable hosts. However both of these solutions necessarily introduce delay, which may be unacceptable, and neither guarantees correctness. If the 'expected delay' is exceeded then an error is likely to result.

We propose an alternative based on the heartbeats described in section 4.10. We arrange that a client is aware of the earliest event that may be produced by any server. If a client receives **A** with time stamp t_A then it must wait until it receives **B** with $t_B < t_A$ or notification that there are no more events to be received from B's server with a time stamp $< t_A$.

This information is supplied by means of an *event horizon time stamp* sent with every heart beat and event notification. This indicates a lower bound on time stamps of future notifications from that service. Use of event horizon time stamps does not preclude a service from producing events out of order, which is important for the independence of composite event activations that are re-signalled as base events. Event horizon time stamps ensure correctness in the light of failures but introduce an expected delay equal to half the heart beat interval. In the following section we will consider how this method may be combined with probabilistic correctness gained from buffering solutions in order to give a tunable trade-off for different application environments.

6.8.3 Trading Correctness

In the previous section a computation/delay trade-off was explained. If a rapid heartbeat is chosen, then there is a relatively high computation and network cost, but a low delay when evaluating $\mathbf{A} - \mathbf{B}$. Alternatively, a slow heartbeat can be used that is computationally inexpensive but that leads to longer expected delays. In some application environments a third parameter, certainty of correctness, can be traded for reduced delay.

To facilitate this, a parameter **Delay** is added to the **Before** operator.

$$C \Leftarrow C_1 - C_2\{\text{Delay} = \delta\}$$

Where δ is the maximum time that evaluation should be delayed after C_1 is signalled before $\neg C_2$ is assumed. The heartbeat rate can be reduced independently, although the two factors clearly interact. An infinitely slow heartbeat will give the same correctness semantics as the delay system presented in [MS95].

6.8.4 Clock Drift

As indicated earlier, the clocks in different computers can only be synchronised to a certain degree, and this can make it difficult to reliably order events that originated from different machines. In a centralised machine we may order E_1 and E_2 simply by comparing their time stamps. However, in a distributed environment we can only order events to within a certain probability. This probability will depend on the expected clock drift between the machines and in the difference in the actual time stamps.

We can take account of this probabilistic ordering by extending the specification of 'sequence' and 'without' to include a *minimum* probability. For example

$$C \Leftarrow C_1 - C_2\{\text{Probability} = \pi\}$$

If the time stamps for C_1 and C_2 are similar, then the probability of them being correctly ordered is low, whereas if the time stamps indicate that they took place a long time apart, then there is a high probability of correct ordering. Equally, if C_1 and C_2 originate from machines with well-synchronised clocks they may be ordered more reliably than if they do not. A client may take account of ordering probabilities in their specification. Expressions for

Signal if **A** *almost certainly* occurred before **B**
(*high minimum probability*)

and

Signal if **A** *might possibly have* occurred before **B**
(*low minimum probability*)

may both be specified. If the probability distribution of the clocks is known, these specifications may be translated into modifications in the acceptable time stamps for **A** and **B**, and no additional run time overhead need be incurred.

However there are other issues to be considered. Clock synchronisation is remarkably good in many circumstances[Mil91], and other issues may have more bearing on apparent event ordering than clock drift. For example the time taken for sensors running on a multi-processing machine to detect an event occurrence may vary considerably. This will affect the time stamp the event is given and hence the apparent ordering of events. Even if clock drift *is* the main source of errors, the clock distribution may not be known to a sufficient degree of accuracy to allow meaningful processing of a client's specifications.

The method presented above allows for flexible *specification* of the importance of correct event ordering. However further research is required in order to create implementations that can match the intended semantics. Other work

[SHM96] has tackled this issue by introducing concurrency operators into the composite event language, to allow a client may indicate that ordering is unknown. This gives equivalent expressive power to the above mechanism, if the clock variation is assumed to be rectangular.

6.9 Aggregation

In the previous sections, much has been said about the aim of reducing the dependence on a total ordering of event instances, and the need for a *global view*. The methodology used to achieve this is to split evaluation into a number of independent instances. However, there are occasions where it is necessary to take a number of evaluations and treat them together. For example in order to count the number of **Deposit** events between an **Open** event and a **Close** event we may treat the evaluation of **Deposit** events separately, but must ultimately aggregate them. In general an aggregation function will take the form

$$\Phi(f(C), s, E) = f'(\Phi(C, s, E))$$

Although an evaluation Φ conceptually returns a set of occurrences, evaluation may never terminate and a realistic implementation will return a *stream* of matching occurrences. In the following sections we discuss what facilities are required in a language for the specification of aggregation functions, and how these functions may make use of *meta events* such as notification of network delays.

Although an aggregation function will process a set of event occurrences, there may be many simultaneous and independent evaluations of an aggregation function. For example, to count the number of deposits made into each of several bank accounts, a candidate expression might be

$$\$\mathbf{Open}(x); \mathbf{COUNT}(\$\mathbf{Deposit}(x, y)\text{-}\mathbf{Close}(x))$$

where **COUNT** is a suitable aggregation function. The evaluations for each account should be independent, and indeed aggregation functions are implemented by a single push-down state in the same way as the basic operators.

6.9.1 Requirements

An aggregation function collates a number of composite event instances from one or more streams and then generates new events based on the collected information. The aggregation function should generate 'aggregate events' at the earliest possible moment. This may be when a sub-event occurs, when the stream is terminated[6], or when information about event absence is discovered. An aggregation function should be able to perform processing on any of these occurrences.

For example, in order to signal the *first* of **A** and **B** to occur, it is not sufficient to receive notification of **A**. It is also necessary to receive information that **B** has not occurred.

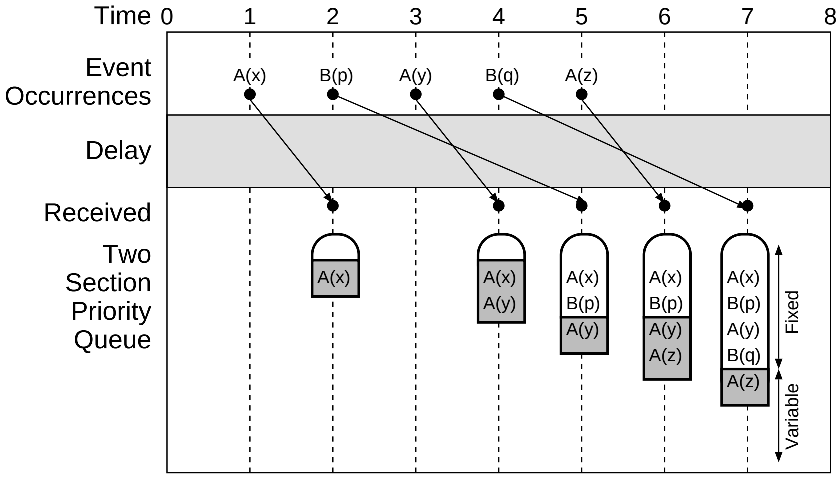


Figure 6.6: A Two Section Priority Queue

The diagram above illustrates the growth of a priority queue as event instances from the expression **A**|**B** are received. As events are received, they are added to the queue in occurrence order. In addition the fixed portion of the queue grows as the system gains knowledge about delayed events. For example at time 5 the event received indicates that no **B** events occurred prior to time 2.

6.9.2 Data Structures

The data structure chosen to represent a stream of events is essentially a priority queue, extended to allow the *meta-events* to be clearly specified. The priority queue stores (sub)events in time stamp order, and delayed events are inserted into the queue at the appropriate point. The queue has two sections. The first section is 'fixed'. The system guarantees that there will be no more insertions into this section. The second section is variable, and inserts may be made into it. Gradually, as delayed events are received and processed, and as heartbeats 'promise' the absence of events from particular servers the fixed portion will grow. The aggregation function is made aware of changes to the size of the fixed portion via meta-events, and may act accordingly. Figure 6.6 illustrates this data structure.

6.10 A Language for Aggregation Functions

Aggregation functions could be specified in any programming language capable of managing queues of structured data. Lisp is a strong contender, as it is interpreted and clearly suited to this task. However a simple language will suffice for the majority of aggregation requirements, and to this end a 'toy' language based loosely on C has been devised[ANS90]. An aggregation function

[6] Some streams are finite (e.g. **A**), some potentially infinite (e.g. $**A**) and some are potentially infinite but bounded (e.g. $**A**-**B**).

is specified by a block of the following format:

`{ int t =0;`	Local variable definitions.
`expr:` $**Deposit**($x$)-**Close**	An event expression
`event: t=t+new.x`	The action to be taken on an event occurrence.
`var:`	The action to be taken if a portion of the queue becomes fixed.
`end: hd.t=t;accept hd;`	The action to be taken when (if) the queue
`}`	is terminated.

The above example calculates the total amount deposited in a bank before the close of business. t is a local variable used to accumulate the values associated with **Deposit** events, and is initialised to zero. The sub-expression evaluated yields a stream of deposit events which is eventually terminated when **Close** occurs. The phrase `event:` ... indicates the action to be taken on each deposit (update of the accumulated total); and the statement `end:` ... indicates the action to be taken when the **Close** event occurs (assignment of the total to an event and acceptance of that event). This example is explained in detail in section 6.11.1. The `var:` phrase is empty as no action should be taken on meta-events.

If an aggregation function requires more than one stream of events, then there will be more than one `expr` statement, and matching `event`, `var` and `end` statements for each one. The action taken on a (meta)event occurrence can be to update variables in the local environment, or in the environment of one of the queued event instances; and to accept or reject one or more events.

6.10.1 Variables

The bead representing each event instance carries with it an environment consisting of a set of variable→value mappings. An aggregation function might require access to these variables. Queues of event instances are represented within the aggregation function by a linked list of records. A single pointer type is provided, and variables of this type point to an event instance record. `hd` points to the head of the queue[7], and `p.tl` points to the record after `p` (or `null` if `p` is the last record on the queue). In addition, `var` is a pointer to the first record on the variable portion of the queue and `new` is a pointer to the last item added to the queue. Pointers may be compared if they point to records on the same queue, and `p<q` if `p` is closer to the head of the queue than `q`. Any non-null pointer is `<null`. The following relations can be proved trivially.

if `hd=null`	then the queue is empty
if `hd<var`	then the first item on the queue represents the earliest event.

The aggregation function may access an event's environment via indirection of a pointer. For example `p.x=4` sets the variable `x` in the environment of the event instance pointed to by `p` to the value 4.

[7] When more than one queue is used, these are enumerated, i.e. `hd0`,`hd1`, ...

6.10.2 Constructs

A C-like syntax is chosen for statements. In addition to variable assignment two flow control constructs and two statements are provided.

`if(`$expr_1$`)` $stmt_1$ `elseif(`$expr_2$`)` $stmt_2 \ldots$ `else` $stmt_0$

Conditional execution, with the expected semantics.

`loop(pt=hd)` *stmt*

This construct evaluates *stmt* once for each record in the queue pointed to by `hd`. In each evaluation `pt` is set to point to the current record.

`accept pt`

The record pointed to by `pt` is accepted and removed from the queue.

`kill pt`

The record pointed to by `pt` is removed from the queue. If `pt` is omitted, then all current (and future) records are deleted. This is used to signal that evaluation should terminate.

6.11 Examples

Some example uses of aggregation are shown in this section.

6.11.1 Counting

A common use of aggregation is for counting. Reconsider the bank account example of section 6.10. We may wish to calculate the total amount deposited in each bank account between the opening of the account ($\mathbf{Open}(x)$) and the closing of the account ($\mathbf{Close}(x)$). A possible solution is shown below:

$$\$\mathbf{Open}(x);\{ \quad (1)$$
$$\texttt{int t=0;} \quad (2)$$
$$\texttt{expr: } \$\mathbf{Deposit}(x,a) - \mathbf{Close}(x) \quad (3)$$
$$\texttt{event: t=t+new.a} \quad (4)$$
$$\texttt{end: hd.t = t;accept hd;} \quad (5)$$
$$\}$$

The first line states that an evaluation of the aggregation function should start each time there is an $\mathbf{Open}(\texttt{i})$ event. The event expression within the aggregation function will be given the initial environment ($x = \texttt{i}$). The second line declares a local variable, t, and assigns it the initial value 0. This is used to store the total amount deposited. The third line is an expression that evaluates to a set of event instances, one for each deposit in the appropriate account. The fourth line indicates that the total should be updated by the deposited amount, each time an event instance is added to the queue.

The fifth line indicates that when the queue is terminated (i.e. when the **Close** event has occurred, and all delayed events have been received) then an event instance should be signalled with an environment updated with $t =$ total deposited. As all queued events are equally suitable, the event at the head of the queue is taken.

There are a number of improvements that can be made to the above definition. Firstly, there is a bug. If no events are signalled, and an empty queue is terminated, then `hd` = `null` and the statements for the `end` meta-event are invalid. Secondly, when the aggregate event is eventually signalled, the sub-event that is updated will have a time stamp indicating when *it* occurred. For this example, a more appropriate time stamp would be that of the **Close** event. This may be achieved by setting the time stamp variable (@) to the time of the `end` meta-event.

Finally, as only one event instance in the queue is required, the others may be deleted as they occur, thus saving resources. A final version of the expression is therefore:

```
$Open(x);{
          int t=0;
           expr: $Deposit(x,a) − Close(x)
          event: t=t+new.a;kill hd.tl;
            end: if(hd) {hd.t = t;hd.@=@;accept hd;}
         }
```

6.11.2 Maximum

Another common use of aggregation is to select one from a set of candidate event instances. In this example, we will select the event representing the longest long-jump during a competition delimited by **Start** and **End** events.

```
$Start();{
          pt q=null;
           expr: $Jump(d) − End()
          event: if(q==null)
                   {q=new;}
                 elseif(new.d>q.d)
                   {kill q;q=new;}
                 else
                   {kill new;}
            end: if(hd) {accept hd;}
         }
```

In this expression, only one event is kept on the queue; the one representing the longest jump. Unlike the previous example, when this event is eventually signalled, it retains its original time stamp, as this is more appropriate in this case.

6.11.3 First / Once

Consider an expression to detect the *first* athlete to cross a finishing line. In order to signal this, we must ensure that the event representing an earlier athlete has not been delayed. This may be done by examining the `var` pointer, and by using the `var` meta-event.

```
$Start();{
           expr: $Finish(a)
          event: if(var>hd)
                   {accept hd;kill;}
            var: if(var>hd)
                   {accept hd;kill;}
         }
```

When the first event occurs, this is immediately signalled — providing that there is a guarantee that no more events will be inserted before it. If this is not the case, then an earlier event will eventually arrive (and be signalled) or a `var` meta-event will occur indicating that the first event is no longer in the variable section of the queue. Whatever the reason, when the first event is signalled, other events are no longer of interest and are all 'killed'. In terms of implementation this will stop further evaluation of the sub-expression and de-register interest in all pertinent events. The mechanism required for this is exactly the same as that for destroying beads during evaluation of 'Without', as described in section 6.7.

In the squash example of section 6.6, 'First' could be used as a suitable aggregation function to ensure that the end of the point was only signalled once. However a function with weaker semantics will suffice. We are interested in the first *detection* of the end of the point — so that we may stop play. This is a common requirement of composite expressions representing alarms. In general, an alarm should be signalled on the first occasion that events warranting it are detected. If two sensors detect a fire, the alarm should ring as soon as a **Fire** event is received, regardless of possible delayed events from the second sensor. Both of these examples may be performed by the trivial aggregation function

```
$Start();{
          expr: expression
         event: accept hd;kill;
        }
```

6.12 Conclusions

Although many systems are inherently event driven, there has been little research into the design of general mechanisms for the notification and management of distributed events. This chapter presents an event architecture which allows implementation details to be hidden, and thus simplifies the construction of complex distributed applications.

In an open environment, it is not plausible to maintain a central database of every event occurrence, and registration is therefore the natural approach. The extension of object interface definition to include event classes is intuitive, and fits the object oriented paradigm well.

Composite event detection in a distributed environment is a complex problem. In particular the requirements are very much application specific. Speed of detection, accuracy and failure tolerance are trade-offs that differ from domain to domain. The language presented to represent composite event expressions is both more flexible, and, in the author's opinion, more intuitive than other composite event grammars. The implementation of this language is both efficient and 'tunable' to application domain requirements.

In this chapter the design of a global badge system was given as an example event driven system. This presents many challenges for an event architecture, in particular there are both a large number of *producers* of events and a large number of *consumers*. In the following chapter, the administrative problems inherent in such an application are considered; in particular how security policies for event notification can be represented, and how policies from different organisations will interact.

Chapter 7

Event Security

7.1 Introduction

Access control for event notification is a quite different problem to that for procedural systems. In this chapter we will investigate why this is the case, and explain the design of an access control system for events. Section 7.2 highlights both the differences and similarities between access control for events and operations. Section 7.3 describes how a trivial modification of RDL will allow it to be used as a policy expression mechanism for event based systems. Section 7.4 discusses implementation problems, and offers some solutions. Section 7.5 uses the Active Badge System of the previous chapter as an example to evaluate the requirements of a real system, and the extent to which these are met. Finally section 7.6 concludes.

7.2 The Problem with Events

With 'standard' procedural systems, whenever a client wishes to perform some operation, there is a dialogue between the client and the server whereby the client supplies a number of credentials. The server examines these, consults stored policy, and if the client has sufficient privileges, the operation is allowed to proceed. For each operation, the client need only supply credentials required for *that* operation. If access is denied, the client is informed and may try again with a stronger set of credentials.

With an event based system, such an approach is not feasible. When an event occurs, the server must decide which clients are interested in this event, and which of the interested clients are *allowed* to see the event. Clearly a client cannot be consulted at this point to provide credentials, and so all relevant credentials must be supplied at an earlier point.

In general, credentials will be supplied at registration, for later consultation when an event occurs. In addition the process of registration may itself be subject to access control. However, this is a procedural request like any other and may be protected using the mechanisms discussed in previous chapters. This is not considered further here.

A second problem with events is that the expected computation cost of access control checks is considerably higher than in procedural systems. In a procedural system credentials must be examined on each client call; however there is a great deal of scope for caching optimisations. Generally client calls are bursty - a client may perform many operations on one object before accessing another, and the number of clients in the current 'working set' may be quite small. In an event system, supplied credentials must cover all event instances, and so there may be more of them. In addition each event is independent, and a large number of clients may be interested in it. Caching relies on rapid re-use of information, and it is not clear how it could be applied to event access control.

A more subtle solution than caching is therefore required in order to reduce the $O(no.events \times no.clients)$ cost. In section 7.4 we will derive a suitable system that reduces the need for per-client checks on each event occurrence.

Despite these differences, event access control has a number of similarities to procedural approaches. In particular the issues raised in chapter 2 about client naming and inter-working between applications still apply. As events are a key tool in the development of cooperating applications, these issues are arguably even more significant for event based systems than for procedural ones.

The badge system is an example of a cooperating set of servers. Each server may be in a separate site, each with separate schemes for (human) client naming. A client interested in badges currently located in a number of different sites must be able to inter-work with each of these services. The issues of such a system are considered in detail in section 7.5 and used as a basis to evaluate the proposed model.

7.3 Policy Specification

Access control should be specified in terms of client roles and in terms of the parameters to events. For example, a simple policy statement from the badge system might be that a user may be told of sightings relating to their own badge. i.e.

clients with $\mathbf{Login.LoggedOn}(u, h)$ may be informed of
$\mathbf{Seen}(b, s)$ where $b = \mathbf{Badge}(u)$

This is clearly syntactically similar to the role definition

$$\mathbf{MaySee}(b, s) \leftarrow \mathbf{Login.LoggedOn}(u, h) \; : b = \mathbf{Badge}(u)$$

Indeed, modifying the semantics of RDL in this way leads to a uniform mechanism for expressing access control policy. In the extended, event form of RDL (ERDL) the left hand role specification is replaced with an event template, and the modified semantics are that a client supplying suitable credentials is allowed access to events matching the event specification. Conceptually, when an event occurs, the service first discovers which clients have registered interested in the event. For each interested client, if a matching ERDL rule is found, and the client has provided sufficient credentials to allow entry, then the client is informed of the event. If no such rule is found, then the client is not informed.

A complication is that the type scheme for events is considerably richer than that for RDL. To deal with the mismatch, each event instance is mapped to a *representative* used for access control. The representative is a model of the event, which has parameters that are significant for access control purposes. An alternative to the example above would be to map the badge and sensor identifiers in the **Seen** event to the corresponding user and room. **Seen**(*user*, *room*) could then be used in policy statements rather than **Seen**(*badge*, *sensor*). This mapping can simplify the syntax of the policy definition, and can hide implementation issues that are not relevant to the policy administrator.

As with the expression of access control lists, it is useful to be able to specify both positive and negative access control entries. For example 'students may not see staff when they are in a meeting room' is a plausible negative rule that might take precedence over other positive rules such as 'a student may see their supervisor'. We further extend ERDL by allowing negative entries to be specified. When an event occurs, each interested client is notified providing the (event,client) pair match a positive rule, without a preceding matching negative rule. For example, the above example may be specified as

$$\begin{array}{rll} -\mathbf{Seen}(u,r) & \Leftarrow^{1} & \mathbf{Login.LoggedOn}(s,h) \\ & & \quad : u \text{ in } \texttt{staff} \wedge s \text{ in } \texttt{students} \wedge r \text{ in } \texttt{Meeting} \\ \mathbf{Seen}(u,r) & \Leftarrow & \mathbf{Login.LoggedOn}(s,h) \\ & & \quad : \mathbf{Supervisor}(u,s) \end{array}$$

This has analagous semantics to the interpretation of MSSA ACLs, as discussed in section 5.4.4.

7.4 Implementation Issues

When an event occurs, the stored policy must be consulted to see if each interested client may be informed of the event. In this section, we will discuss how the access control information can be pre-processed in order to reduce the expected cost.

The ERDL relating to each event type can be parsed to produce an acceptance function ψ(*event representative*, *client*). This will be a boolean expression of three kinds of term.

- Those that involve only event parameters. For example

  $$\mathbf{Seen}(u,r) \Leftarrow \ldots : u \text{ in } \texttt{staff}$$

  Terms of this type indicate access to be allowed (or denied) dependent only on the parameters to the event, and not the parameters of the role membership certificates supplied as credentials. Computations relating to these terms can be cached and used for all clients.

- Those that involve only client parameters. For example

  $$\ldots \Leftarrow \mathbf{Login.LoggedOn}(s,h) : s \text{ in } \texttt{students}$$

  Terms of this type indicate access rights dependent solely on the parameters of supplied role membership certificates, and not on the event parameters. Computations relating to these terms may be calculated at registration, and cached for use whenever an event occurs.

[1] The $\Leftarrow$ symbol is used to denote that this is an ERDL statement.

- Those that involve both event and client parameters. For example

$$\mathbf{Seen}(u,r) \Leftarrow \mathbf{Login.LoggedOn}(s,h) \;:\mathbf{Supervisor}(u,s)$$

  These terms cannot be optimised. They must be recalculated for each (event,client) pair.

The ERDL rules are processed to separate these types of terms, and in addition order the expression, so that 'expensive' calculations are only made if acceptance/rejection cannot be based solely on 'cheap' ones.

Figure 7.1a,b gives a subset of the access control rules for one site in the badge system. These are used to illustrate the transformations that are applied. Firstly negative rules are distributed over positive rules, to remove the rule ordering. The set of rules are then converted into a single boolean expression. This is illustrated by figure 7.1c. The expression is then converted to disjunctive normal form and each clause is further divided into terms of the three types indicated above (Figure 7.1d).

Client expressions ($C_0 \ldots C_n$) need not be evaluated for each event occurrence, as they are only dependent on client credentials. They may therefore be computed at registration. If a group membership later changes to invalidate a credential, this is managed by the mechanisms described in section 4.6.

Equally, event expressions ($E_0 \ldots E_n$) need only be computed once for each event, as they do not involve client credentials.

By construction, each clause in the acceptance function ψ contains exactly one client expression, so that once $E_0 \ldots E_n$ have been evaluated, it can be expressed as

$$(C_i \vee C_j \vee C_k \ldots) \vee ((C_u \wedge f_u(e,c)) \vee (C_v \wedge f_v(e,c)) \ldots)$$

A bitmask representing the truth values for $C_0 \ldots C_n$ is stored with each client record, and therefore ψ for each client can be rapidly computed by performing the intersection of this with the bitmask representing $\{C_i \vee C_j \vee \ldots\}$. If any terms match, the client may be informed of the event. If no terms match, the intersection with $\{C_u \vee C_v \vee \ldots\}$ is performed. It is only if this succeeds that terms dependent on both the client and the event need be computed.

Access control can therefore be performed by some precomputation, followed by a series of calculations of increasing cost. Further gains can be made by combining these computations with the matching operation required to determine which clients are interested in an event. For each client who *might* be interested, cheap access control checks are preformed. It is only if these indicate that the client *may* be eligible to receive the event that the pattern matching takes place. Finally the more expensive access control checks take place if required.

These transformations, together with careful implementation, can dramatically reduce the amount of computation required for access control for events.

7.5 Badge System Requirements

In this section we will consider the access control implications for both inter and intra site communications within the badge system. We will consider three sites, "CL", "Eng" and "ORL". CL is a university computer science department, and

1. Anyone may see their own badge.
2. `rmn`'s secretary may see him.
3. Staff may see `rmn`.
4. Students may not see `rmn`.
5. Students may see staff.
6. Students may not see each other.
7. Staff may see each other.

(a)

$$
\begin{array}{rcll}
\mathbf{Seen}(u,r) & \Leftarrow & \mathbf{LoggedOn}(u,h) & & (1)\\
\mathbf{Seen}(\texttt{rmn},r) & \Leftarrow & \mathbf{LoggedOn}(\texttt{akl},h) & & (2)\\
\mathbf{Seen}(\texttt{rmn},r) & \Leftarrow & \mathbf{LoggedOn}(p,h) & : p \text{ in } \texttt{staff} & (3)\\
-\mathbf{Seen}(\texttt{rmn},r) & \Leftarrow & \mathbf{LoggedOn}(p,h) & : p \text{ in } \texttt{student} & (4)\\
\mathbf{Seen}(u,r) & \Leftarrow & \mathbf{LoggedOn}(p,h) & : u \text{ in } \texttt{staff} \wedge p \text{ in } \texttt{student} & (5)\\
-\mathbf{Seen}(u,r) & \Leftarrow & \mathbf{LoggedOn}(p,h) & : u \text{ in } \texttt{student} \wedge p \text{ in } \texttt{student} & (6)\\
\mathbf{Seen}(u,r) & \Leftarrow & \mathbf{LoggedOn}(p,h) & : u \text{ in } \texttt{staff} \wedge p \text{ in } \texttt{staff} & (7)
\end{array}
$$

(b)

$$
\begin{array}{l}
\psi(\mathbf{Seen}(u,r), \mathbf{LoggedOn}(p,h)) = \\
\quad (u = p) \\
\quad \vee (u = \texttt{rmn} \wedge p = \texttt{akl}) \\
\quad \vee (u = \texttt{rmn} \wedge p \text{ in } \texttt{staff}) \\
\quad \vee (u \text{ in } \texttt{staff} \wedge p \text{ in } \texttt{student} \wedge \neg(u = \texttt{rmn} \wedge p \text{ in } \texttt{student})) \\
\quad \vee (u \text{ in } \texttt{staff} \wedge p \text{ in } \texttt{staff} \wedge \neg(u = \texttt{rmn} \wedge p \text{ in } \texttt{student}) \wedge \\
\qquad \neg(u \text{ in } \texttt{student} \wedge p \text{ in } \texttt{student}))
\end{array}
$$

(c)

Client Expressions:

$C_0 \Leftarrow (\mathit{true})$
$C_1 \Leftarrow (p = \texttt{akl})$
$C_2 \Leftarrow (p \text{ in } \texttt{staff})$
$C_3 \Leftarrow (p \text{ in } \texttt{student})$
$C_4 \Leftarrow (\neg p \text{ in } \texttt{student} \wedge p \text{ in } \texttt{staff})$

Event Expressions:

$E_0 \Leftarrow (u = \texttt{rmn})$
$E_1 \Leftarrow (u \text{ in } \texttt{staff})$
$E_2 \Leftarrow (u \text{ in } \texttt{student})$

$$
\begin{array}{l}
\psi(\mathbf{Seen}(u,r), \mathbf{LoggedOn}(p,h)) = \\
\quad (C_0 \wedge p = u) \\
\quad \vee (C_1 \wedge E_0) \\
\quad \vee (C_2 \wedge E_0) \\
\quad \vee (C_3 \wedge \neg E_0 \wedge E_1) \\
\quad \vee (C_4 \wedge \neg E_0 \wedge E_1) \\
\quad \vee (C_4 \wedge E_1) \\
\quad \vee (C_2 \wedge \neg E_0 \wedge E_1 \wedge \neg E_2) \\
\quad \vee (C_4 \wedge E_1 \wedge \neg E_2)
\end{array}
$$

(d)

Figure 7.1: Stages in Preprocessing ERDL

An example policy from the badge system is given in (a), and specified in ERDL in (b). This is transformed to a boolean acceptance function (c), and then rewritten to eliminate common expressions, and separate event and client clauses (d).

"Eng" is the engineering department in the same university. The two departments are administered separately. ORL is a local company that has close links with CL. Indeed some members of staff at ORL are also staff members at CL.

In the next section we will consider the access control requirements of each of the three sites, and in the rest of the chapter we will consider how these may be implemented using the access control methods explained in the first part of this chapter.

7.5.1 Local Policies

In this section we will propose a (somewhat Draconian) set of access control policies that might exist between the sites, in order to illustrate some of the issues of distributed administration. Only **Seen** events are considered. Other events such as **MovedSite** will also require access control, but these are omitted for clarity.

- The engineering department only issues badges for use by staff members. It is the policy that any staff member may see any other, when in the department. Sightings of visiting badges are also available to all staff. Badge sightings are normally made available only within the department, the exception being that sightings from visiting members of other departments may be seen by members of that department - subject to that department's policy. Badges are not issued to students, and students have no access to badge sightings.

- The computer laboratory has a complex policy for local and visiting badges. Students may only see staff when they are in their own office, or in public areas within the laboratory. Staff may see each other at any time. Few students have badges, and those that do may be seen by anyone at any time. Requests from external sites are given the same access rights as students, unless they come from named individuals who are staff members at both the computer laboratory and the external site.

- The staff at ORL are allowed access to all local badge sightings at all times. Requests from external sites are always disallowed - unless they come from one of the staff members who is currently at the university.

Although the three organisations are administered separately, they are not completely separate, and there is some degree of cooperation between the sites. For example, the engineering department wishes to respect the computer laboratory's policy with respect to those members of it who are currently in the engineering department. Equally, the computer laboratory and ORL cooperate over sightings of people who hold positions in both organisations.

7.5.2 Defining Local Policy

Figure 7.2 gives ERDL statements for the policies for the three organisations. These will be considered in turn.

- Policy for ORL may be expressed simply and directly. Statement 1 indicates that all users logged into the local system may see all badge sightings. Statements 2 and 3 name users with accounts on remote systems

$$
\begin{array}{lcll}
\mathbf{Seen}(u,r) & \Leftarrow & \mathbf{Login.LoggedOn}(p,h) & (1) \\
\mathbf{Seen}(u,r) & \Leftarrow & \mathbf{CL.LoggedOn}(\mathtt{ah},h) & (2) \\
\mathbf{Seen}(u,r) & \Leftarrow & \mathbf{CL.LoggedOn}(\mathtt{djg},h) & (3)
\end{array}
$$

(a) ORL

$$
\begin{array}{lclll}
\mathbf{Seen}(u,r) & \Leftarrow & & : u \text{ in } \mathtt{staff} \wedge (r \text{ in } \mathtt{PubRoom} \vee \mathbf{Office}(u,r)) & (1) \\
\mathbf{Seen}(u,r) & \Leftarrow & \mathbf{Login.LoggedOn}(p,h) & : u \text{ in } \mathtt{staff} \wedge p \text{ in } \mathtt{staff} & (2) \\
\mathbf{Seen}(u,r) & \Leftarrow & & : u \text{ in } \mathtt{student} & (3) \\
\mathbf{Seen}(u,r) & \Leftarrow & \mathbf{ORL.LoggedOn}(\mathtt{ah},h) & : u \text{ in } \mathtt{staff} & (4) \\
\mathbf{Seen}(u,r) & \Leftarrow & \mathbf{ORL.LoggedOn}(\mathtt{djg},h) & : u \text{ in } \mathtt{staff} & (5)
\end{array}
$$

(b) Computer Laboratory

$$
\begin{array}{lclll}
\mathbf{Seen}(u,r) & \Leftarrow & \mathbf{Login.LoggedOn}(p,h) & : p \text{ in } \mathtt{staff} & (1) \\
\mathbf{Seen}(u,r) & \Leftarrow & ??? & : \mathbf{Home}(u) = \mathtt{CL} & (2)
\end{array}
$$

(c) Engineering

Figure 7.2: Policies for the Three Sites

who may also see sightings when using these accounts. In the Oasis name server, **Login** maps to the services responsible for issuing local (ORL) login certificates, and **CL** maps to the equivalent service in the computer laboratory[2].

- The computer laboratory has a more complex policy, and correspondingly complex ERDL. Statement 1 indicates that, under certain conditions, sightings of staff badges may be seen by anyone. In the original policy it was stated that this access was available to students, and to users from external sites. As there is no way to distinguish between external users (who have no useful credentials), and internal users who choose not to provide credentials, this statement is weaker that the original specification. Statements 2 and 3 are straightforward, and 4 and 5 perform the same function as the analogous ones in the ORL policy.

- Expressing policy for the engineering department is more difficult. The first statement is straightforward, but the second is more troublesome. The intention is to specify that sightings of badges belonging to members of the computer laboratory may be seen by members of the computer laboratory, subject to the computer laboratory's policy. A simple solution would be to manually insert the computer laboratory's policy specification into the engineering department's badge system. However, this requires too much cooperation between the administrators on the two sites, and an alternative, distributed, solution is preferred. This will be considered in the following section.

[2]This requires a mechanism for trading of interfaces between organisations. ODP architectures such as ANSA [APM93], and CORBA [Gro92] provide such mechanisms.

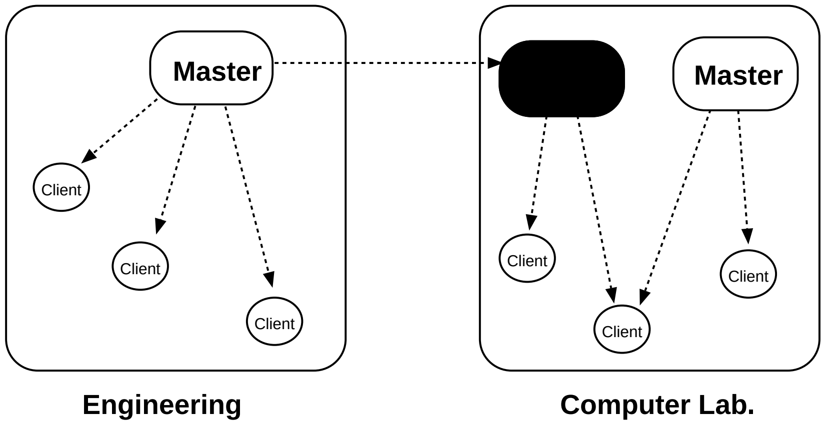


Figure 7.3: Enforcing Remote Policy using Proxys

In the diagram, badge events within engineering are only notified to clients within the computer laboratory via the proxy server. This ensures that computer laboratory policy is not breached.

7.5.3 Remote Policy

When cooperation between organisations is required as in the above example, this can be achieved by inserting a level of indirection between the event source in one site, and the clients in the other. For the above example, a proxy server is created in the computer laboratory that receives event notifications from engineering, and filters these according to the computer laboratory's policy, before echoing them to computer laboratory clients. This is illustrated by figure 7.3. Additional advantages in this scheme are that the network bandwidth between the two sites is reduced, and that administrators on both sites may collate auditing information. The ERDL for the engineering department's badge service may now be completed by adding the line

$$\mathbf{Seen}(u, r) \ \Leftarrow\ \mathbf{CLsys.BadgeProxy}(\text{`}\mathtt{Engineering}\text{'}) \ : \mathbf{Home}(u) = \mathtt{CL}$$

In this instance the Oasis service named by **CLsys** is the computer laboratory's role service responsible for naming and authenticating system services.

7.6 Conclusions

Distributed event management is a very young area of computer science. Currently, events are used to aid management or monitoring within closed systems. For example, telecommunications switches may report faults and accounting information to a management system, and active databases may use events for consistency or constraint checks. In these environments, security is not an important issue, as there is effectively only one client (the manager) who may see everything.

We belive that the event paradigm is also useful in the development of more open applications, where there are many event sources and a number of untrusted clients. In this environment event access control is essential. Additionally, the exercise of extending RDL to meet the requirements of event access control is a significant test of the ability of Oasis to extend naturally to new problem domains.

Chapter 8

Conclusions

This dissertation presents an open architecture for access control that allows fine grain specification and enforcement of arbitrary access control policies. This chapter summarises the main conclusions and suggests further work.

8.1 Summary

The fundamental problem that this work attacks is the degree of compromise present in access control systems. Trade-offs are made on the grounds of efficiency, security and readability.

Openness Existing access control architectures do not interwork well with each other. This is a difficult problem. Mechanisms often interwork badly, as they rely on conflicting assumptions. In addition, it is difficult to reason about the interaction of policies that are not formally specified, or that are specified in terms of different concepts and use different languages.

Recent work has addressed this issue by developing *large scale* security architectures, such as Kerberos, Sesame or the ODP proposals. Although these systems allow interaction between different organisations that use the same security architecture; they are not *open* with respect to interaction between organisations using different mechanisms.

Oasis tackles this problem in two ways. Firstly, policy is specified using a flexible grammar. Policies written in other grammars can be translated into this grammar to aid reasoning (chapter 3). Secondly, credential records provide a general mechanism for the enforcement of access control policies based on the notion of *beliefs* (chapter 4). Within Oasis these beliefs relate to Oasis concepts, such as the validity of a certificate. However, the concept of belief is very general, and when interworking with other architectures, beliefs may be related to other concepts such as the time of day or the integrity of a Kerberos host.

Flexibility The flexibility of an access control mechanism is limited by the granularity of client naming. An architecture which names clients by UserId, for example, cannot perform access control based on process identity. A more subtle point is that policy that is *implicit* in an implementation is inflexible. For example, in the Unix filing system, the policy relating to who may read or modify an access control list is fixed. A good test of an access control architecture is how well it extends to protect itself. Are the same mechanisms used for the specification and enforcement of *meta*-policy?

In chapter 2, we considered the issues of client naming, and derived a two level naming scheme to overcome the problems of granularity. In chapter 5 we considered the issues of meta-access control in the context of the MSSA storage architecture, and showed that Oasis does indeed extend in this way, and that meta-access control need not be unduly expensive.

In chapter 7 we considered how Oasis may be extended to protect a new class of objects — events. As computing environments are constantly evolving, it is important that security architectures may evolve to meet emerging requirements. As events are protected by Oasis and Oasis is built using events, the mechanisms used in Oasis may themselves be protected, and (for example) the notification of policy changes may be subjected to access control.

Clarity Security policies must be expressed in a clear and unambiguous way. If this is not the case, then security breaches will occur through mistakes. Early access control list schemes tended to provide either clarity or expressive power but not both. Some schemes are simple and easily understood (for example Unix ACLs) whereas others are powerful but confusing (for example Phoenix/MVS ACLs).

The author believes that policies expressed in RDL are easily understood (chapter 3). In addition, by moving the problem to the domain of naming, the majority of policy specification can be left to a small number of administrators, who allocate roles to users. Most policies under the control of 'ordinary' users are likely to be expressed as simple access control lists, and for this reason the representation of ACLs was given careful attention (chapter 5).

Efficiency The most efficient security mechanism is none at all. Efficiency is therefore always a trade-off. The expressive power available when specifying policy and speed of propagation of policy updates are likely to be compromised in the fight for faster systems. A side effect of this battle is that systems tend to exhibit artifacts that are not part of the policy definition. For example, revocation, group membership updates or policy modification may be delayed for efficiency reasons, giving system behaviour that only approximates to the stated policy. These trade-offs lead to an unsatisfactory profusion of loop-holes and quirks that are not formally specified or amenable to reasoning.

Oasis is unusual in that the implementation closely matches the specification. Where efficiency trade-offs are required, they correspond to bounded delays that are only applicable in the case of failure. For example a server may reduce load by sending only occasional heartbeats. This will result in revocation delays, but only if the network or server fails (chapter 4).

The approach to access control in this thesis may be summarised as the flexible specification of access control policy and the efficient and direct implementation of this policy.

8.2 Further Work

Further work falls into four categories; mechanisms for the expression of policies; extensions to the implementation; the application of Oasis principles to other domains; and formal reasoning.

Policy Expression There are many possible security policies that cannot be directly specified in RDL. However it is believed that the majority of these can be expressed by RDL statements extended with special operators, which might be provided by a particular service. Crucially, other Oasis services should be unaffected by such extensions.

For example *organisational roles* have received much recent interest. In these schemes there are constraints such as *no user may be both an accounts manager and a purchasing manager*. Such a statement may be specified in an extended RDL statement as

$$\mathbf{AccountManager}(u) \leftarrow \mathbf{CandidateAccountManager}(u) \\ : \neg\texttt{PurchasingManager(u)}$$

Further work is required to determine the extent to which alternative schemes can be represented in this way.

Implementation Extensions The primary extension to Oasis that is needed is the replication of Oasis servers. This is a hard problem, as it is fundamental to the implementation that the issuer of a certificate must validate it. A possible approach is to use a signature based on a secret shared between the replicas of a service for the cryptographic check, but efficient validation of credential records is more difficult. Further work is required to determine suitable mechanisms.

Other Problem Domains The mechanisms presented in this thesis relate as much to naming as to access control. As such, they ought to prove useful when considering issues of auditing, accounting or non-repudiation.

Formal Reasoning Much has been said about the advantages of RDL as a means for clear policy specification. RDL statements correspond to axioms, which should be amenable to automatic analysis. Such analysis would have many benefits. For example queries such as 'determine the circumstances in which a user from outside the organisation may access the database' would prove very useful for closing loopholes and tracing leaks. This work has not considered the details of these issues, and it is clear there is scope for further work.

Oasis is an architecture designed for the provision of access control in large distributed systems. It is designed to be flexible and efficient. A prototype system has been built and tested, however experience in applying Oasis to a 'real world' system is the only way to quantify the true cost of such a flexible scheme. The world is, and always will be, heterogeneous. Existing schemes for interworking rely on homogeneous mechanisms that are unrealistic when interaction between a large number of organisations is considered. Oasis does not make such an assumption, and whilst in an ideal world Oasis could be adopted as the universal scheme, this thesis demonstrates that secure interworking in a heterogeneous environment is possible.

Bibliography

[ANS90] ANSI. *Programming Language C.* American National Standard for Information Systems, New York, 1990.

[APM93] APML. The ansa model for trading and federation. Technical Report AR.005.00, Architecture Projects Management Ltd, 1993.

[Bac92] Jean M. Bacon. *Concurrent Systems.* Addison-Wesley Publishing Company, 1992.

[BGS92] J.A. Bull, Li Gong, and K. R. Sollins. Towards security in an open systems federation. *Lecture Notes in Computer Science*, (648), 1992. ESORICS 92.

[BMLH94a] Jean M. Bacon, Ken Moody, Sai Lai Lo, and Richard John Hayton. Access control for a modular, extensible storage service. In *IEEE SDNE, Services in Distributed Network Environments*, Prague, June 1994.

[BMLH94b] Jean M. Bacon, Ken Moody, Sai Lai Lo, and Richard John Hayton. Extensible access control through a hierarchy of servers. *ACM Operating Systems Review*, July 1994.

[BMLH94c] Jean M. Bacon, Ken Moody, Sai Lai Lo, and Richard John Hayton. Modular, extensible storage services through object interfaces. In *ACM SIGOPS European Workshop*, pages 141–146, September 1994.

[BN80] A. D. Birrell and Roger M Needham. A universal file server. In *IEEE Transactions SE*, volume 6(5), 1980.

[Bro94] Kraig Brockschmidt. *Inside OLE2.* Microsoft Press, 1994. ISBN 1-55615-618-9.

[CD94] George Coulouris and Jean Dollimore. A security model for cooperative work. Technical Report 674, Department of Computer Science, Queen Mary and Westfield College, University of London, 1994.

[CKAK94] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proceedings of the 20th VLDB Conference, Santiago, Chile*, September 1994.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[DO85] James Donahue and Willie-Sue Orr. Walnut: Storing electronic mail in a database. Technical report, Palo Alto Research Centre, 1985.

[Doc] Phoenix/MVS Online Documentation. Help file protection.

[GD93] S. Gatziu and K.R. Dittrich. Events in an active object-oriented database system. In *Proceedings of the 1st International Workshop on Rules in Database Systems, Edinburgh*, August 1993.

[GJO92] N. H. Gehani, H. V. Jagadish, and O.Shmueli. Composite event specification in active databases: Model & implementation. In *18th VLDB Conference*, Vancouver, British Columbia, Canada, 1992.

[Gon89] Li Gong. A secure identity-based capability system. In *IEEE 1989 Symposium on Security and Privacy*, pages 56–63, May 1989.

[Gro92] Object Management Group. *The Common Object Request Broker: Architecture and Specification.* John Wiley & Sons, Inc., 1992.

[GS86] I Greif and S. Sarin. Data sharing in group work. In *First Conference on Computer Supported Cooperative Work*, pages 175–183, Austin, Texas, December 1986.

[HHB93] Andy Hopper, Andy Harter, and Tom Blackie. The active badge system. In *ACM INTERCHI'93*, Amsterdam, April 1993. Olivetti Research Ltd Technical Report 93.7 (video).

[HKG+88] John H. Howard, Michael L. Kazar, Sherri G.Menees, David A. Nichols, M Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[LABW93] Butler Lampson, Martin Abadi, Michael Burrows, and Edward Wobber. A calculus for access-control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, 1993.

[LABW94] Butler Lampson, Martin Abadi, Michael Burrows, and Edward Wobber. Authentication in the taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, 1994.

[Lam71] B. W. Lampson. Protection. In *Fifth Princeton Symposium on Information Sciences and Systems*, pages 437–443, Princeton University, March 1971. Reprinted in Operating Systems Review, 8, 1, January 1974 pp.18-24.

[Lo94] Sai Lai Lo. *A Modular and Extensible Network Storage Architecture.* PhD thesis, University of Cambridge, January 1994. Technical Report No. TR 326.

[Lun88] Teresa F. Lunt. Access control policies: Some unanswered questions. Technical report, SRI International, June 1988.

[LYS95] Emil C. Lupu, Nicholas Yialelis, and Morris Sloman. A policy based role framework for access control. In *First ACM Workshop on Role-Based Access Control*, Gaithersburg, Maryland, USA, November 1995.

[Mil91] D. L. Mills. Internet time synchronization: The network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1492, October 1991.

[MO87] Masaaki Mizuno and Arthur E. Oldehoeft. An access control language for object-oriented programing systems. Technical Report TR-CS-87-12, Department of Computer Science, Kansas State University, November 1987.

[MS91] J. D. Moffett and M. S. Sloman. Content-dependent access control. *ACM SIGOPS Operating Systems Review*, 25(2):63–70, April 1991.

[MS95] Masoud Mansouri-Samani and Morris Sloman. Gem a generalised event monitoring language for distributed systems. Technical Report Doc 95/8, Impreial College, July 1995. Revised version of Report No Doc 93/49, December 93.

[MST90] Jonathan Moffett, Morris Sloman, and Kevin Twidle. Specifying discretionary access control policy for distributed systems. *Computer Communications*, 13(9):571–580, November 1990.

[Neu93] B. Clifford Neuman. Proxy-based authorization and accounting for distributed systems. In *13th International Conference on Distributed Computing Systems*, Pitsburgh, May 1993.

[OMG94] OMG Security Working Group. *OMG White Paper on Security*, April 1994.

[Org72] E. Organick. *The Multics System An Examination of Its Structure.* 1972. ISBN 0-262-15012-3.

[Pow93] Joel Powell. *Multitask Windows NT.* Waite Group Press, 1993. ISBN 1-878739-57-3.

[Red74] David D. Redell. *Naming and Protection in Extendible Operating Systems.* PhD thesis, University of California, Berkeley, CA, USA, 1974. published as Project MAC TR-140, Massachusetts Institute of Technology, Cambridge, MA, USA, November 1974.

[RT78] D. M. Ritchie and K. Thompson. The unix time-sharing system. *The Bell System Technical Journal*, 57(6)(2):1905–1930, 1978.

[Sat89] M. Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions of Computer Systems*, 7(3):247–280, August 1989.

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feistein, and Charles E. Youman. Role based access control models. *IEEE Computer*, February 1996.

[SHM96] Scarlet Schwiderski, Andrew Herbert, and Ken Moody. Monitoring composite events in distributed systems. Technical Report 387, University of Cambridge, February 1996.

[SL87] S. Stepney and S. P. Lord. A formal model of access control. *Software - Practice and Experience*, 17(9):575–593, September 1987.

[Slo94] Morris Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management, Plenum Press*, 2(4), 1994.

[SR90] Y. C. Shim and C.V. Ramamoorthy. Monitoring and control of distributed systems. In *First International Conference on Systems Integration*, pages 672–681, Morristown, NJ, 1990. IEEE Computing Press.

[Vin88] S. T. Vinter. Extending discretionary access controls. In *IEEE 1988 Symposium on Security and Privacy*, pages 39–49, April 1988.

[WC96] Jennifer Widom and Stefano Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Data Management Systems. Morgan Kaufmann Publishers, Inc., 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA", 1996.

[Yu89] Che-Fn Yu. Access control and authorization plan for customer control of network services. In *IEEE GLOBECOM '89*, pages 862–869, Dallas, Texas, November 1989.

[Zlo77] M. M. Zloof. Query-by-example: a data base language. *IBM Systems Journal*, 16(4):324–343, 1977.