Secure Naming for Distributed Computing using the Condensed Graph Model

by

Thomas Brendan Brabants Quillinan, B.Eng., M.Sc.

THESIS

Presented to the Faculty of Science
National University of Ireland,
Cork

for the Degree of

Doctor of Philosophy

July 2006

“O! be some other name: What’s in a name? that which we call a rose,
by any other name would smell as sweet.”

William Shakespeare, Romeo and Juliet Act II, Scene II.

Abstract

Distributing computations across multiple machines has become increasingly important in recent years. With the advent of the computational Grid and other distributed compute projects, such as Seti@Home and, most recently, Stardust@Home, the distributed computation area is expanding rapidly. In general, distributed computing incorporates the areas of meta-computing, Grid computing, cluster computing and Web Services. Implementing a distributed computation architecture has several basic functional requirements, such as load balancing, fault tolerance and security. Providing a security architecture for such a diverse area is an important challenge. Requirements include ensuring the integrity of results, providing an access-control mechanism for sensitive resources and computations, and authenticating the users of the system.

An important aspect of a security architecture for a distributed system is the identification and control of both the computation and the compute nodes within the system. For example, in order to control access to a computation, we must be able to identify the individual components that make up the computation. We propose that in order to control a computation, we must be able to name all the aspects of the computation. The central premise of our thesis is: *"If you can name it, you can control access to it"*.

This dissertation examines the security requirements of the WebCom distributed computing environment. WebCom is an extensible distributed computation environment that has the ability to execute arbitrarily complex compute jobs on many different types of architecture. WebCom is primarily designed to execute condensed graph applications in a distributed manner. This thesis develops the security architecture for WebCom, primarily to provide a systematic access control mechanism for condensed graph applications. We explore the condensed graph model and develop a naming system that is used to control the execution of these graphs and allows the specification of sophisticated security policies in a distributed environment. SDSI-like local naming is used to name objects in condensed graphs. We demonstrate the flexibility of this architecture with a number of case studies, including a micropayment architecture for distributed computations, an automated administration architecture for Grid and an activity based secure workflow architecture.

Dedicated to my Grandfather, Captain P. Brendan Sugrue 1917 - 2005, who passed away during the writing of this thesis. He never let me forget that family is the most important treasure that we posess. He considered his family his greatest achievement. I will forever miss his larger than life personality, his caring of others and, most of all, his sense of mischief and fun.

“Ní fheicfidh muid a leithéid arís ”

Acknowledgements

First and foremost, I want to thank my family who have believed in and supported me during my endless college career: at last the eternal student no more. To my parents whose advice, belief and love has always been readily available and is appreciated, if not always at the time. I especially want to thank my sister Niamh who spent a weekend proof-reading this totally unfamiliar work. She has also kindly provided me with somewhere (nice!) to live for the last four years. I also want to thank my other siblings, Cliona and Cillian, who have helped and entertained me over the years. I especially want to thank my mother, Áine, and brother, Cormac, who ransomed the laptop containing this work back from the thieves who stole it. I also want to thank Galway Bay FM and especially the Galway Sentinal for their help retreiving the laptop.

I also want to acknowledge the help and friendship shown to me by the members of the Centre for Unified Computing in UCC especially: Adarsh, Barry, Brian, Dave, Hongbin, James, John O'Regan, Keith, Max, Neil, Padraig, Philip and Therese. Special mention must go to Barry Mulcahy and Brian Clayton, with whom I have collaborated in the past and from whom I've learnt a great deal. Thanks also go to Barry for the last-minute proof reading. Also to Philip Healy with whom a friendly competition for completion, that I lost by two weeks, helped keep me focused throughout seemingly endless writing up period. Thanks also go to the former CUC postgradutes Daithí and Colm for their help. I want to thank the other postgraduates in computer science that have helped (and entertained) me, especially the Adrians, Jonathan, Marie, Utz and Will.

To the head of the CUC, John Morrison, who has always helped and provides a great environment within the group. I particularly appreciate his acting as my internal examiner in very difficult circumstances. I also want to thank my external examiner, Bruce Christianson, who provided a rigorous, but enjoyable, examination. His interest and enthusiasm are greatly appreciated.

Last, but never least, to my supervisor, Simon Foley, who has guided and supported me throughout my Ph.D. and whose advice was always available and cogent. I can never thank him enough for all his help. Special thanks also go to Vivien for those gorgeous brownies!

Contents

List of Figures

List of Tables

Part I

Introduction and Overview

Chapter 1

Introduction

A challenge for the design of access control mechanisms for distributed systems can be summarised as: *“if you can fully name an object, then you can properly control access to that object”*. Access control is primarily concerned with limiting the actions that authorised users of a system can perform, either directly, or indirectly through programs running on the system. This dissertation investigates the security requirements for the WebCom [123] distributed computation environment, and the condensed graphs [122] computation model on which it is based.

1.1 Distributed Computing

Distributed computing provides the ability to execute complex problems across multiple networked computers. Distributed applications range from massively parallel computations such as distributed cryptographic key cracking [4] or climate prediction [3] to complex enterprise workflows and supply chain management. The provision of a secure environment for distributed computing is a necessary part of any distributed computation system.

The basic requirement for distributed computing is the ability to link users and resources as transparently as possible. Ideally, distributed computations should be fault tolerant, the computation load should be spread across available resources as evenly as possible and the computations should be secure.

Security is an important aspect of distributed applications. When a distributed application is scheduled to execute on external resources, the stakeholders (application and resource owners) typically need security guarantees. For example, the resource owner might require that the application should not have access to local data.

Analysing the requirements of stakeholders in distributed computations implies analysing the threats to distributed computations. Threats to a distributed computation system include the illicit modification of data used in a computation; the modification of the computation itself; the unauthorised access of data by principals; the unauthorised execution of computations, and identity theft.

Addressing these threats entails the development of security policies that allow stakeholders to define their requirements and the development of a mechanism that enforces these policies. Distributed computations are, by their very nature, decentralised. For this reason, a decentralised security architecture is required for distributed applications. One such approach is Trust Management [32].

Trust management schemes [16, 29, 152] use public key certificates to specify delegation of authorisation between public keys and can be used to help decentralise authorisation policies. Trust Management is an approach to constructing and interpreting the trust relationships between public keys that are used to mediate security critical actions. Cryptographic credentials are used to specify delegation of authorisation between public keys. Trust Management has been used for a number of applications including active networks [34] and to control access to Web pages [2, 42, 44].

Trust management systems have a number of advantages compared to the traditional systems based on X.509 [41]. Policies and certificates are created and maintained separately from the application in a very natural way. The attributes used within the policies and/or certificates are application defined, and they are represented in a customisable fashion, allowing the application designer to decide what characteristics are required. Changing the attributes does not require changes to the trust management system used. By removing the traditional lookup of an identity's authority, and instead representing that authority within the certificate, applications no longer need to consider the security of where and how this authority is stored. An additional benefit of utilising a trust management system within an application is that designers and implementers of the application are required to consider trust management explicitly. This encourages good practices when considering the overall security of such applications. Trust management policies are easy to distribute across networks, helping to avoid reliance on centralised configuration of distributed applications.

Trust management provides a flexible approach to specifying and enforcing security requirement across a network of resources. Each of the stakeholders in the system, for example, the owner of compute resources and/or application, can specify their security requirements in terms of trust management policies.

1.2 WebCom

WebCom applications are specified as condensed graphs. Condensed graphs are directed acyclic graphs where the nodes are computational components and the arcs specify the sequencing constraints between nodes. WebCom is a multi-tiered parent-child based architecture. When a condensed graph is executed by WebCom, the nodes in the graph are scheduled by the WebCom parent to its children. Children can become parents themselves and schedule work to their own children.

Condensed graphs can be used as a distributed job control language to describe the scheduling of operations in an application. Nodes represent value-transforming actions and can be defined at any level of granularity, ranging from low-level machine instructions to mobile-code programs. Examples include computational primitives, Web Services [148], Corba objects [28, 81], PVM computations [125], Grid applications [123], and commercial-off-the-shelf (COTS) components [118]. Atomic operations in a condensed graph application need not address synchronisation or concurrency concerns: such details are implicitly specified by the arcs between nodes and are managed by the condensed graph execution scheduler.

WebCom has been designed as a modular architecture where individual components, such as fault tolerance [104], load balancing [144] or security, can be replaced as required by applications. WebCom's modules are connected to a central scheduler and are used to help determine where nodes are to be scheduled for execution.

WebCom handles the issues associated with distributed computations, such as communication, load balancing, fault management and security. These features are transparent to the execution of condensed graph applications. In this dissertation, we describe the security architecture of WebCom. This is an architecture that can control where the nodes in a graph are executed, and monitor the results of these executions.

1.3 Naming Distributed Computations

Creating security policies for distributed computations is a challenging prospect. Different applications have a wide range of security goals. In this dissertation, we are primarily interested in access control. Providing a means to create such access control policies requires having the ability to refer to components throughout the computation in a consistent and potentially unique way. For example, application owners may want to specify where sensitive portions of their computation are executed, or to specify an acceptable range for the result of a computation.

Our thesis is that this fundamental problem can be reduced to a *naming* problem. The central premise of our argument is *"if you can name it, then you can make authorisation decisions about it"*. If every component (or, in the case of a condensed graph application, a node) in the computation is properly named, then it can be referred to with as much precision as is required.

Object names range from simple descriptions, such as *LaserPrinter*, to globally unique references, for example, the digital object identifier [73] *http://doi.acm.org/10.1145/1111348.1111359*. This requirement is seen is all aspects of computing. For example, locating websites on the Internet requires the use of DNS names.

Naming distributed components is not a new problem, for example CORBA [81], the Spring naming system [149], the X.500 naming architecture [182] and Enterprise Java Beans (EJB) [169] each provide practical solutions towards the naming of distributed components. However, each of these solutions addresses naming as a static problem. Distributed objects in these systems have *a priori* defined names as they do not change often. In contrast, nodes in a condensed graph evolve continually during execution and therefore the names of these nodes must also evolve. A naming scheme for condensed graphs must consider this evolving nature of components in the computations.

In this dissertation, we develop a naming model for condensed graphs. This model allows us to name nodes in a condensed graph with as much precision as is required. These names are used to specify policy requirements in WebCom, and are known as *WebCom names*. WebCom names can be used by any of WebCom's modules.

1.4 Securing WebCom

Securing WebCom involves specifying and enforcing security policies for condensed graph applications. As nodes in condensed graphs are represented by WebCom names, security policies are specified in terms of these names. In this dissertation, we develop an access control model and secure authentication mechanism for WebCom. This model defines what is meant by a secure WebCom system.

Access control policies are enforced in Secure WebCom by the security manager. The security manager ensures that any resources involved in scheduling and/or executing nodes are authorised to do so. The enforcement mechanism that is used by security managers is dependent on application requirements. We provide an application programming interface (API) for WebCom that allows third party enforcement mechanisms to be implemented if required. A general purpose trust management based security manager is available. Secure communication between instances of WebCom is also supported using secure communication managers; for example, a SSL-based communication manager module.

Managing and verifying the principals using a distributed computing environment entails ensuring that there is a systematic means to determine the authenticity of the principals and the resources used in the computation. This can be provided through the use of authentication mechanisms.

The WebCom security architecture is designed to address both access control and authentication. We argue that the goal of access control for distributed computations is threefold. It can be characterised as the need to ensure that: computations will be executed only on resources that are explicitly authorised; resources will execute only computations that come from authorised servers, and results of computation execution will be accepted only from resources that are authorised. In an access control based security architecture, access to an object is authorised when the subject has been granted permission to use the object in the requested way.

The authentication problem in WebCom can be characterised as the requirement of two principals to set up a communication channel whereby each principal believes that they are communicating only with the other principal.

WebCom's security architecture addresses authorisation and authentication separately: WebCom's authorisation architecture is supported by the naming and security manager modules; authentication is supported by WebCom's communications manager. This entails using a secure authentication protocol, such as SSL/TLS [92], and providing support for a public key infrastructure (PKI), when necessary. Providing authentic and secure connections between WebComs ensures that data is sent to the correct destination, and cannot be intercepted, or modified, by a third party.

1.5 Contributions

The contributions contained within this dissertation are as follows;

1. A naming architecture for condensed graphs, that specifies the contextual detail required to properly name a distributed component.

2. An access control-based security architecture for WebCom that allows application developers to specify security constraints regarding their applications.

3. A software architecture to support names in practice.

4. A number of case studies that examine the capabilities of WebCom and explore some of the advantages of WebCom's security architecture.

Early versions of the results in this dissertation have published in peer-reviewed publications. These publications are broken down into work that the author was the primary contributor, and collaborations with others.

Primary Contributor. Primary investigative research has concentrated on the naming and security architectures for WebCom.

- S. N. Foley, T. B. Quillinan, J. P. Morrison, D. A. Power, and J. J. Kennedy. Exploiting KeyNote in WebCom: Architecture neutral glue for Trust Management. In Proceedings of the Nordic Workshop on Secure IT Systems Encouraging Co-operation, Reykjavik University, Reykjavik, Iceland, October 2000.

- S. N. Foley, T. B. Quillinan, and J. P. Morrison. Secure component distribution using WebCom. In Proceeding of the 17th International Conference on Information Security (IFIP/SEC 2002), Cairo, Egypt, May 2002.

- S. N. Foley and T. B. Quillinan. Using Trust Management to support Micropayments. In Proceedings of the Second Information Technology and Telecommunications Conference, pages 219–223, Waterford Institute of Technology, Waterford, Ireland., October 2002. TecNet.

- T. B. Quillinan and S. N. Foley. Security in WebCom: Addressing naming issues for a Web Services architecture. In Proceedings of the 2004 ACM Workshop on Secure Web Services (SWS)., Washington D.C., USA., October 2004. ACM.
- T. B. Quillinan and S. N. Foley. Synchronisation in Trust Management using push authorisation. In Proceedings of the First International Workshop on Security and Trust Management STM2005. Electronic Notes in Theoretical Computer Science, September, 2005.

Contributions as part of collaborations. Several of the case studies discussed in Chapter 8 are the result of collaborations with researchers in the Centre for Unified Computing in UCC.

- S. N. Foley, T. B. Quillinan, M. O'Connor, B. P. Mulcahy, and J. P. Morrison. A framework for heterogeneous middleware security. In Proceedings of the 13th International Heterogeneous Computing Workshop, Santa Fe, New Mexico, USA., April 2004. IPDPS.
- T. B. Quillinan, B. C. Clayton, and S. N. Foley. GridAdmin: Decentralising grid administration using Trust Management. In Proceedings of the Third International Symposium on Parallel and Distributed Computing (ISPDC04), Cork, Ireland, July 2004.
- S. N. Foley, B. P. Mulcahy, and T. B. Quillinan. Dynamic administrative coalitions with WebComDAC. In WeB2004: the Third Workshop on e-Business, Washington D.C., USA, December 2004.
- B. C. Clayton, T. B. Quillinan and S. N. Foley. Automating security configuration for the Grid. In Journal of Scientific Programming. IOS Press, Vol 13, No. 9, 2005.
- S. N. Foley, B. P. Mulcahy, T. B. Quillinan, M. O'Connor and J. P. Morrison. Supporting Heterogeneous Middleware Security Policies in WebCom. In Journal of High Speed Networks (Special issue on Security Policy Management). IOS Press, 2006. *To appear*.

1.6 Layout of Dissertation

The remainder of this dissertation is structured as follows: Part II examines the background information and current research discussed in this dissertation. In particular, Chapter 2 examines security research relevant to this dissertation; Chapter 3 investigates common naming systems; Chapter 4 describes the condensed graph model in some detail, and finally, the WebCom distributed metacomputer is examined in Chapter 5.

Part III contains the primary contribution provided by this thesis. Specifically, Chapter 6 introduces the naming architecture for the condensed graph model; this naming architecture is then applied to WebCom, and a new security model is described in Chapter 7; Chapter 8 evaluates the effectiveness of architecture through the examination of applications for Secure WebCom.

Part IV (Chapter 9) discusses the results of this Thesis and proposes some future work that may be undertaken.

Part II

Background and Review

Chapter 2

Authorisation and Authentication

In this chapter, we examine the current security research relevant to our thesis. Securing any system entails identifying and addressing the threats to that system. There are many classes of security threats, including identity theft, the misappropriation of information, illicit access to protected resources and so on. In this chapter we examine mechanisms and technologies that address specific categories of security threats, including authorisation and authentication.

This dissertation is primarily concerned with the development of a security architecture for the WebCom distributed computation environment. Providing for secure distributed computation involves controlling access to resources and authenticating the entities that are participating in the computation. In this chapter, we investigate methods to control access to security critical operations and methods to properly authenticate entities of the system.

Section 2.1 investigates access control and introduces the fundamental concepts of the access control matrix, reference monitors and the security kernel. In Section 2.2, we discuss the conventional models of access control, including the Bell LaPadula [25], Biba [27] and Clark-Wilson [46] models. As trust management is extensively used throughout this dissertation, Section 2.3 reviews several of the trust management schemes currently available, including PolicyMaker [33], KeyNote [31] and SPKI/SDSI [56, 152]. Section 2.4 investigates authentication and describes authentication protocols such as SSL/TLS [92]. Finally, we discuss other relevant security research in Section 2.5.

2.1 Access Control

Access control [77] is concerned with providing control over security critical actions that take place in a system. Providing control over actions consists of explicitly determining either the actions that are permitted by the system, or explicitly determining the actions that are not permitted by the system.

2.1.1 Access Control Matrix

Lampson [110] introduced the concept of an access control matrix, with domains forming the rows, objects forming the columns, and cells indicating the permissions. Objects are things in the system that need to be protected. Subjects are entities that have access to objects. Permissions are attributes that specify the access that subjects have to objects. Subjects can themselves be objects. The access control matrix model is not intended for practical use.

Example 2.1 A simple access control matrix is shown in Figure 2.1. This system has three objects,

	`File1`	`DirectoryB`	`InetSocket`
`Alice`	*read*		*write*
`Bob`	*read, write*	*write*	*read*

Figure 2.1: A simple access control matrix.

`File1`, `DirectoryB` and `InetSocket`, and two subjects, `Alice` and `Bob`. The cells display the access rights that subjects have to the objects. For example, Alice has *read* access to `File1`, but no access rights to `DirectoryB`. △

An access control model captures the set of allowed actions as a policy within a system. In [110], Lampson defined the term *protection* to describe *"mechanisms that control the access of a program to things in the system"*. This notion of protection was further formalised in the Harrison-Ruzzo-Ullman (HRU) access control model [84]. The HRU model provides a theoretical study of policies to control the creation and removal of access rights, subjects and objects in the Lampson matrix model. The HRU model formalised the *safety* problem, that is, an access mechanism is considered safe when there is no sequence of commands that can cause the matrix to *leak* an access right. A leak occurs when a sequence of commands exist that add an access right to a subject for an object that previously did not have that right. One of the significant results of HRU is that the safety problem is undecidable in their model. Other work, such as [15], has examined means to make the problem decidable, for example, by limiting the commands to contain a single operation or limiting the number of subjects in the system.

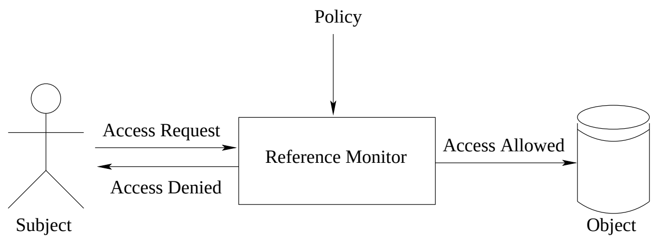


Figure 2.2: The Reference Monitor Model

A *reference monitor* represents the mechanism that implements the access control model, and is depicted in Figure 2.2. A reference monitor is defined by the Department of Defence Trusted Computer System Evaluation Criteria (TCSEC) (commonly known as the Orange Book) as:

> *An access control concept that refers to an abstract machine that mediates all accesses to objects by subjects.*

A reference monitor typically operates as follows: a security critical action is required, for example, an access request for sensitive data, the reference monitor intercepts the action and checks whether the action is authorised according to the security policy. If it is, then the action proceeds. Otherwise the security critical action is not authorised and the caller is notified of this failure. Many security systems use the reference monitor paradigm to enforce security policies. However, every access control model has a different means to specify their security policy, and therefore, a different implementation of the reference monitor. In practice, implementations of reference monitors lie in a range between the two extremes of the security/usability tradeoff: security kernels and application based reference monitors. Security kernels provide verifiable security but are more difficult to configure; application based reference monitors are easier to use and configure, but are more easily bypassed. Other implementations of the reference monitor model also exist that lie between these extremes. For example, application wrappers, such as TCP/IP wrappers [174], have kernel-level primitives that are used to confine access of the application to the system, but operate at the application layer.

Security Kernel

A security kernel [74] is an implementation of a reference monitor in the kernel of a system. This means that all actions that take place on the system are mediated upon by the security kernel. A security kernel is defined by [173] as:

> *The hardware, firmware and software elements of a trusted computing base that implement the reference monitor concept. It must mediate all access, be protected from modification and to be verifiable as correct.*

The trusted computing base (TCB) is defined by [173] as:

> *The totality of protection mechanisms within a computer system–including hardware, firmware, and software–the combination of which is responsible for enforcing a security policy.*

The advantages to the security kernel approach lie in the fact that any security architecture can be compromised when the attacker manages to infiltrate a layer below the security system. A security kernel runs at the lowest software layer, avoiding these types of attack. A reference monitor is an

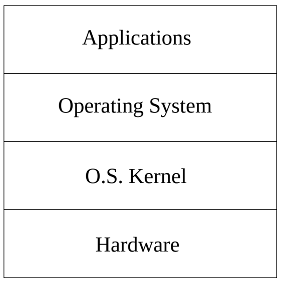


Figure 2.3: Layers of a Software System.

abstract model, the security kernel is an implementation of that model, and the trusted computing base contains the security kernel together with other protection mechanisms [77].

Figure 2.3 shows the component layers of a software system. In these systems, the hardware is on the bottom, with the operating system providing access to applications. The security kernel forms part of the operating system kernel.

Application-based Reference Monitors

Application-based reference monitors are reference monitors that operate at the application layer of software systems. They are typically embedded into a specific application, rather than operating on the entire system. The application system makes the security decision using advice from their application-based reference monitor. Examples of application-based reference monitors include those that use Trust Management [32] and the Java [170] security model [78, 79].

Such systems are typically used to enforce security policies on user actions. For example, the Java security model is used to enforce the access programs have to the system. However, the Java security model also has kernel-level primitives that are used to confine code, for example, to particular JVM domains, to support application level mechanisms.

2.2 Access Control Models

Security models characterise different kinds of security policies. There are many different access control models, such as Bell LaPadula (BLP) [25], Biba [27] or Access Control Lists (ACL) [160]. These models provide a means to define the security goals of a system. For example, BLP is concerned with ensuring confidentiality of classified information, whereas the Biba model is concerned with ensuring integrity. In general, access control models form two categories: mandatory and discretionary access control.

Mandatory access control means that the security kernel controls the access that subjects have over objects. In contrast, in discretionary access control, the owners of objects define the access that other users have to their objects.

2.2.1 Mandatory Access Control

Mandatory Access Control (MAC) policies allow subjects access to objects only when the security level of the subject is greater than the security level of the object. There are many different types of mandatory access control models, including Bell LaPadula (BLP) [25], Biba [27], Clark-Wilson [46], Chinese wall [40], role-based access control (RBAC) [159, 185] and type enforcement (TE) [36].

Bell LaPadula Model (BLP)

The Bell LaPadula (BLP) [25] model was designed to provide security guarantees for multi-user operating systems. BLP is a state machine model that addresses confidentiality concerns. BLP policies are concerned with preventing information flowing downwards from a high security level to a lower level. This can be summarised as "no read up" and "no write down." Such policies are commonly referred to as multi-level security (MLS)-type policies.

BLP defines three access control properties, two of them define mandatory access properties (ss-property and *-property), the third defines a discretionary property (ds-property).

- The Simple Security Property (*ss-property*) states that a subject at a given security level may not read an object at a higher, or disjoint, security level (no read-up).
- The Star (*) Security Property (**-property*) states that a subject at a given security level must not write to any object at a lower, or disjoint, security level (no write-down).
- The Discretionary Security Property (*ds-property*) uses an access matrix to specify discretionary access control.

As BLP is a state-machine model, security is defined in terms of the current state and any transitions from that state. An important property of BLP is defined as the *Basic Security Theorem*: *"If all state transitions in a system are secure and if the initial state of a system is secure, then every subsequent state of the system will also be secure, regardless of any input that occurs."*

Biba

The Biba security model [27] is designed to address integrity concerns in terms of access that subjects have to objects. Integrity in this respect is defined in terms of the correctness of data. As in the BLP model, the Biba model is defined in terms of state machines. However, the integrity properties defined by Biba mirror the confidentiality properties defined by BLP. Biba defines two mandatory access properties:

- The Simple Integrity property defines that a subject at a given security level may not write to an object at a higher security level (no write-up).

- The Integrity Star (*) property defines that a subject can read an object at a given security level, that subject may not write to any other object at a higher security level.

If these two properties are upheld then objects cannot be contaminated by lower level information.

Clark-Wilson

The Clark-Wilson model [46] also addresses integrity. The authors argue that one of the primary security concerns for applications is that data is not illicitly modified, and that errors and fraud do not occur. They separate integrity requirements into two areas: internal and external consistency. Internal consistency is concerned with the internal state of the system, and can be enforced by the system. External consistency is outside of the control of the system, and must be enforced by an external mechanism, for example auditing.

The basis for enforcing integrity policies is that data may only be modified by specific programs. Users have access only to these programs, not the data itself. Furthermore, users have to collaborate to perform changes to data. Therefore, multiple users must collude in order for the security system to be broken. This is known as a *separation of duties* requirement.

The Clark-Wilson model uses programs as an intermediate between subjects and objects (in this case data). Subjects are authorised to execute programs; programs are authorised to modify objects.

Chinese Wall

Chinese Wall policies [40, 58, 106] are based on the premise that once a subject accesses an object, they must not access any other objects that cause a conflict of interest. The standard example of such a conflict is when a subject working in an accounting firm accesses the financial data of one company, they should not have access to a competing company's financial data. In effect, once a subject accesses an object a "Chinese wall" is build around any conflicting objects for that subject.

The Chinese Wall model proposes a formal model to address such policies. This model has been defined as an extension to the BLP model [58] to address these specific concerns.

In the Chinese Wall model, as actions can potentially change the access rights that a subject has to every other object, access rights must be examined after every action. In contrast, in the BLP model access rights can usually be considered static.

Role Based Access Control (RBAC)

Role based access control, or RBAC [159, 185], is an access control architecture that places users into roles, and permissions are assigned to these roles. RBAC is designed to reflect real-world relations between users and permissions. Roles define the logical tasks that users can perform. Users become members of roles and roles are assigned permissions. For example, in a financial company, clerks may order items purchased. Thus, in RBAC, we define a *Clerk* role, and assign it the permission to make purchase orders. Then we assign users, who are employed as clerks into the *Clerk* role. This implicitly gives them the permissions necessary to make purchase orders.

RBAC is an efficient means to define an access control policy. As users are grouped in roles and permissions are assigned to these roles, the policy does not require the enumeration of every user and permission possible. The enforcement check is twofold: when a user attempts to perform an action, the reference monitor checks the roles the user is a member of; these roles are then checked in turn to determine whether the action is permitted by any of the user's role. RBAC is commonly used in operating systems, database management systems and middleware architectures.

Type Enforcement

Type enforcement (TE) [36] is a labelling approach to access control. TE is a table-orientated mandatory access control mechanism suited for confining applications to known behaviour. domain and type enforcement (DTE) [21] is an extension of type enforcement including the concept of domains to traditional type enforcement. Several systems use implementations of TE, such as SELinux [115] (DTE) and Flask [165] (TE) to enforce their security policies.

In type enforcement, labels are attached to both subjects and objects. Subjects are considered active entities and a domain label is attached to them. The domain label encodes the access control attributes associated with subjects. Objects are considered passive entities, and a type label is attached to them. The type label encodes the access control attributes associated with objects. The access that subjects have to objects depends on the access capability that the subject's domain has to the object's type. These capabilities are encoded in a set of tables.

There are two main tables, the domain definition table (DDT) and domain interaction table (DIT). As with an access control matrix, the DDT specifies the allowed interactions between subjects (domains) and objects (types). Domains form the rows of the table, types form the columns and the permitted access modes, for example, *read* or *write*, are stored in the cells. When a domain attempts to access a type, the DDT is consulted to determine whether the access is permitted.

The DIT table specifies the allowed interactions between subjects. In the DIT table, both the rows and columns contain subjects (domains). As with the DDT, the cells contain the access mode permitted between domains, for example, create, destroy or signal. When one domain attempts to interact with another, the DIT is consulted and an authorisation determination is made.

2.2.2 Discretionary Access Control

In discretionary access control (DAC) models, subjects can, at their discretion, modify access to objects that they own. For example, the Unix file system controls allows the owners of files to set the read, write and execute permissions for other users. Many operating systems have some type of

DAC system. The most prevalent DAC model is the access control list (ACL) model. DAC models are also used in database management systems and operating systems, as these systems use the concept of subjects owning objects and these subjects control access to their objects.

Access control lists (ACLs) [160] are a simple means to implement the access control matrix model. ACLs represent columns in the matrix. Each object in the system has an associated ACL that holds a list of subjects and the access rights that they hold for that object. When a subject attempts to access an object, the reference monitor checks the access control list associated with the object and determines whether the subject has the required permission.

2.3 Trust Management Systems

Trust Management [29, 32, 80, 152] is an approach to constructing and interpreting the trust relationships among public keys that are used to mediate security critical actions. Credentials are used to specify delegation of authority among public keys, and are used to determine whether a signed request complies with a local authorisation policy.

Blaze et al [32] defined trust management as *"a unified approach to specifying and interpreting security policies, credentials, and relationships that allow direct authorization of security-critical actions"* Trust Management is basically designed to answer the question "Do I trust principal X to do action Y?" A trust management system enables permissions to be associated with cryptographic keys. These permissions can be delegated by one key to another. Trust management systems must be able to navigate these delegation chains, linking a request to the authority required to perform the requested action.

There are two basic categories of cryptographic certificates: identity certificates that define an association between a public key and the identity of the holder of the certificate; and attribute certificates that define an association between public keys and a set of permissions. For consistency, we refer to identity certificates simply as *certificates* and attribute certificates as *credentials*. Existing systems such as X.509 and PGP allow the association of identity to public keys. Trust Management addresses the need to associate abilities to public keys. In other words, certificates answer the question: "who is the holder of this public key?"; credentials answer the question "what can I trust this public key to do?" In general, trust management systems do not necessarily need to verify the identity of the holder of credentials. These questions, while valid security problems, are not relevant to the application that is attempting to decide whether or not an action is authorised. Such problems are instead left to identity certificates systems (such as X.509).

The ability to delegate permissions between users is a central feature of trust management systems. This allows the users of a system to selectively give other users part of their authority. This reflects how authority is shared in reality. In the commercial world, a central figure does not set out the authority of every employee of a company. Instead, the CEO delegates authority for specific areas to company directors, who delegate portions of their authority to specific employees. Trust management supports this view of authorisation.

Example 2.2 Consider a university application (Figure 2.4): A University appoints a President and a Registrar. The President also appoints departmental heads (HoD), who appoint lecturers. The Registrar registers students for specific courses. If this is considered in a trust management sense, then the University delegates the authority to appoint departmental heads and lecturers to the President. The President then further delegates the authority to appoint lecturers to the departmental heads. The Registrar is delegated the authority to register students by the University.

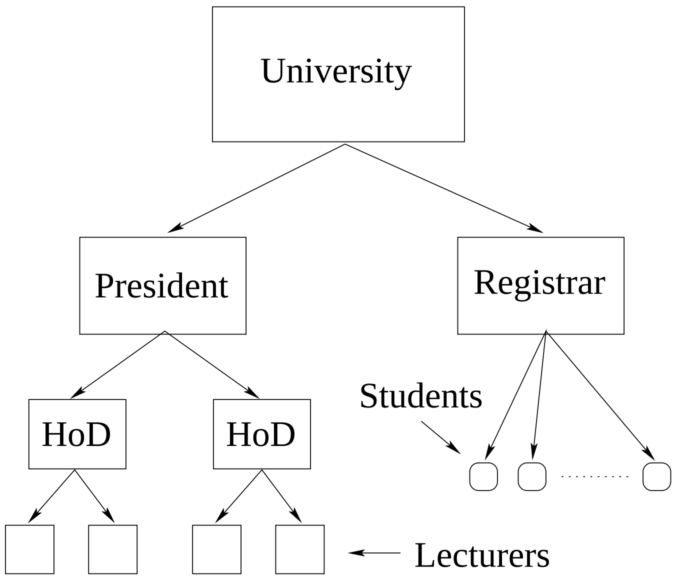


Figure 2.4: Overview of a University Delegation Tree

In this case we can say "The University *trusts* the President to appoint departmental heads and lecturers". Furthermore, the delegation of authority to the departmental heads can be stated as "The President *trusts* the departmental heads to appoint lecturers". △

In general a Trust Management system is made up of five basic components (From [29]):

- A language for describing 'actions', which are operations with security consequences that are to be controlled by the system.
- A mechanism for identifying 'principals', which are entities that can be authorised to perform actions.
- A language for specifying application 'policies', which govern the actions that principals are authorised to perform.
- A language for specifying 'credentials', which allow principals to delegate authorisation to other principals.

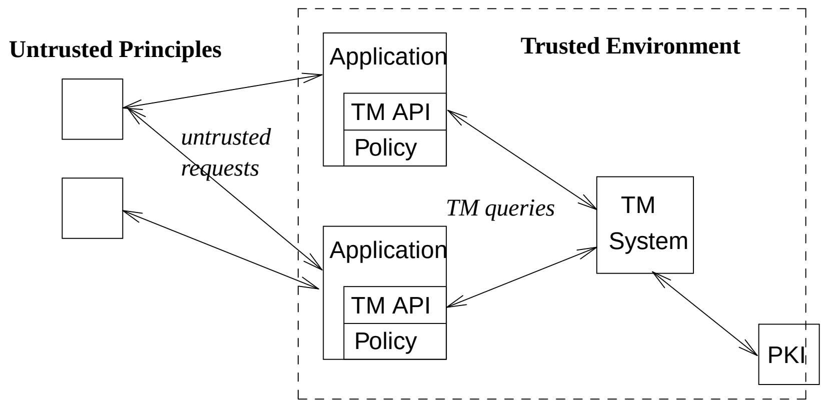


Figure 2.5: KeyNote Trust Management Architecture (From [30]).

- A ‘compliance checker’, which provides a service to applications for determining how an action requested by principals should be handled, given a policy and a set of credentials.

Trust Management systems have a number of advantages compared to the traditional identity-based systems created using X.509 [1]. Policies and certificates are created and maintained separately from the application (Figure 2.5). The terminology used within the policies and/or credentials is application defined. They are represented in a application specific fashion, allowing the application designer to decide what characteristics are required. Trust management removes the traditional access control approach of first determining the identity of the requester, and then determining the requester’s authority. Instead that authority is represented within the credential. Applications no longer have to consider the security of where this authority is stored. A benefit of utilising a trust management system within applications is that designers and implementers of these applications are required to consider authorisation issues explicitly. This encourages good practice when considering the overall security of such applications. Trust management policies are easy to distribute across networks, helping to avoid reliance on centralised configuration of distributed applications.

In Sections 2.3.1 – 2.3.3 we will examine some of the more common trust management systems that are currently available. Section 2.3.4 examines some proposed trust management systems that provide more advanced mechanisms to support specific types of applications.

2.3.1 PolicyMaker

In [32], Blaze et al identified the trust management problem as a distinct component of network security. PolicyMaker [31–33] proposed a solution towards addressing the trust management problem. Prior to the development of PolicyMaker, existing systems that supported security in network applications, such as X509 and PGP, addressed only narrow regions of the trust management spectrum, namely identification. PolicyMaker changed this in providing a generic trust management

[1]X.509 (Section 2.5.1) has also been extended to support trust management [70].

framework.

In PolicyMaker, policies, credentials and trust relationships are specified as programs (or parts of programs), and are expressed in a *safe* programming language. In this context, “safe” means resource and I/O limited, for example, AWKWARD (which is a safe version of the AWK [14] language was developed for the PolicyMaker system). Other “safe” languages include Safe-TCL [57] and Java [170]. As the policies, credentials and relationships are specified as programs, they are extremely flexible and expressive. However, when the authors of [33] analysed the proof mechanism used by PolicyMaker, they found that in its most general form it is undecidable and is NP-hard even when restricted in several natural ways.

Security policies and credentials are defined in terms of predicates, called *filters*. These filters are associated with public keys and accept or reject actions based on what abilities those public keys are trusted to perform. The basic function of the PolicyMaker system is to process *queries*. Queries (see Figure 2.6) are requests to determine whether a public key (or keys) is allowed to perform a specified action, according to the local security policy.

$key_1, key_2, \ldots, key_n$ **REQUESTS** *Action String*

Figure 2.6: Structure of a PolicyMaker query.

Action Strings are application-specific messages that describe a trusted action by one or more public keys. Applications specify themselves the structure and the content of these action strings, PolicyMaker has no knowledge of their structure. These action strings are interpreted only by the application that generates them.

PolicyMaker uses *assertions*, containing bindings between predicates (filters) and one or more public keys. This binding is called an *authority structure*. These assertions (Figure 2.7) confer authority on keys.

Source **ASSERTS** *AuthorityStruct* **WHERE** *Filter*

Figure 2.7: Structure of a PolicyMaker assertion.

The simplest filters are interpreted programs that accept or reject action strings. In Figure 2.7, *Source* indicates the source of the assertion (either the public key of the generator of the assertion or the local policy, in the case of a policy assertion). *AuthorityStruct* specifies the key or keys for whom the assertion has been created. *Filter* is the predicate that action strings must satisfy for the assertion to hold.

As was previously stated, there are two types of PolicyMaker assertions: signed assertions–more commonly called credentials–and unsigned assertions, or policies. A credential is a signed message that binds a particular authority structure to a filter. Policies are unconditionally trusted because they originate locally and are therefore not signed. On any given machine, a local root must exist from whom all trust is delegated. PolicyMaker has no concept of the semantics of action strings or signatures. The calling application must verify the signature. This allows Policymaker to exploit existing signature schemes. Signing a credential represents the delegation of authority from the signer of the credential to the holder of the public key mentioned in the credential.

Example 2.3 Consider the University example shown in Figure 2.4. The University's policy is

```
policy ASSERTS
  pgp:"0x0123456abcdefabe23428129038747b32"
    WHERE
        PREDICATE=regex:"Appoint Staff";
```

Figure 2.8: PolicyMaker policy assertion for University.

represented in the policy assertion shown in Figure 2.8. This gives the PGP key mentioned the authority to appoint staff. As this is a policy assertion, it is unsigned. This authority can be further delegated by the President. For example, the President can sign a credential for department heads authorising them to appoint lectures only, as shown in Figure 2.9.

```
pgp:"0x0123456abcdefabe23428129038747b32"
  ASSERTS
    pgp:"0xfa3463334bc934a34b34fd0232ad"
    WHERE
      PREDICATE=regex:"Appoint Staff
                      Position: Lecturer";
```

Figure 2.9: PolicyMaker assertion for Departmental Head.

This assertion specifies that the only staff the Departmental Head can appoint are lecturers. This assertion would of course be signed by the President's PGP key, in order to bind the predicate to the Departmental Head's PGP key. △

2.3.2 KeyNote

KeyNote [29, 30] is an expressive and flexible trust management scheme that provides a simple credential notation for expressing both security policies and delegation. A standard application programming interface (API) to KeyNote is used by applications to help determine if security critical actions are authorised. The formulation and management of security policies and credentials are separate from the application, making it straightforward to support trust management policies across different applications. KeyNote has been used to provide trust management for a number of applications including active networks [34] and to control access to Web pages [2].

KeyNote was developed to address the complexity issues surrounding PolicyMaker. In effect, KeyNote provides support for a limited subset of PolicyMaker's capabilities. Like PolicyMaker, KeyNote uses a single language to specify policies and credentials. However, unlike PolicyMaker, these have a structure, and are not arbitrary programs. This structure is defined in RFC 2704 [29] and is designed to be flexible and human-readable.

Example 2.4 Figure 2.10 shows an example of a basic KeyNote credential. This credential represents the example discussed before – a University delegates the authority to appoint staff to the President, actually to the President's public key.

```
KeyNote-Version: 2
Local-Constants: kUniversity="RSA:324b234a"
                 kPresident="DSA:67bc23fa2"
 Authorizer: kUniversity
 Licensees: kPresident
 Conditions: app_domain="University" &&
             actions="Appoint" &&
             (Position == "Lecturer" ||
             Position == "Dept-Head");
 Signature: ...
```

Figure 2.10: Basic Structure of a KeyNote credential

The *Conditions* field specifies the authority that the President has been granted, in this case the authority to appoint lecturers or to appoint departmental heads. △

The `Conditions` field defines a set of permissions that represent all possible attribute value combinations. Policies are practically identical syntactically to ordinary credentials, except the `Authorizer` field is not set to a key but is instead set to the keyword `POLICY`. Policy assertions are unsigned (as is true in PolicyMaker) and are implicitly trusted by the application.

KeyNote credentials can be restricted by the authoriser to prevent further delegation of the authority granted. In our original example (Figure 2.4) the Registrar registers students. In effect the registrar would delegate to the students the authority to (for example) sit exams for the subjects that they were registered to take. If a student could further delegate this authority then a situation is created where an unauthorised student is sitting an exam for an authorised student. To prevent this the Registrar will use the _ACTION_AUTHORIZERS keyword within the `Conditions` field. This tells KeyNote that no further delegation is allowed. This keyword specifies the names of principals that are directly authorising the action in the credential.

When a KeyNote query is evaluated by the compliance checker, the evaluator returns the Policy *compliance value* of the query. These compliance values range from _MIN_TRUST to _MAX_TRUST. Applications specify compliance values in the form of sets, again ranging from _MIN_TRUST to _MAX_TRUST. An example would be `{FALSE, TRUE}` where an application requires a boolean answer to a query or `{NOTAPPROVED,APPROVEDLOG,APPROVED}` for an application that requires more context. In this case `NOTAPPROVED` is _MIN_TRUST and `APPROVED` is _MAX_TRUST. An assertion compliance value results from the minimum of the compliance values of the Conditions field and the `Licensee` field. If the `Licensee` or `Conditions` field is missing, then the assertions licensee/conditions compliance value is considered to be _MAX_TRUST; however, if it is present but empty, then the licensee / conditions compliance value is considered to be _MIN_TRUST.

As principals can delegate permissions to other principals, KeyNote must walk these chains of credentials to determine if a request is authorised. The KeyNote compliance checker attempts to find a path between the requesting key and the policy. Policies can also specify thresholds, that is, multiple principals must be involved in any valid request.

The KeyNote assertions syntax is defined in RFC2704 ([29]). For example, the `Authorizer` field is the only required field; all other fields are optional. The optional fields are the `Comment`, `Conditions`, `KeyNote-Version`, `Licensees`, `Local-Constants` and `Signature`. When the `Signature` field is present, it must be the last field. No field must appear more than once, and they are all case insensitive.

2.3.3 SDSI/SPKI

SDSI [47, 112, 152] or Simple Distributed Security Infrastructure (pronounced Sudsy) and SPKI [94] or Simple Public Key Infrastructure (pronounced Spooky) began as two separate standard proposals. It was quickly realised that they shared many similarities, and the projects were combined in 1999.

SDSI was originally developed by Ron Rivest and Butler Lampson to address the complex and incomplete proposals for a public key infrastructure. Unlike X509, SDSI does not rely on a formalised global certificate hierarchy. Instead they use a “Local Naming” architecture that leverages the advantages of a PGP-like “web of trust” (described in Section 2.5.2).

Both SDSI and SPKI are key-centric systems, in that, principals are public keys and all actions are performed by these keys. SDSI does not attempt to link identities to keys, however,people can hold these keys and thus manipulate the system.

The SPKI was proposed by the Internet Engineering Task Force (IETF) in 1996 as an alternative to the X.509 version 3 PKIX public key infrastructure (PKI). The IETF SPKI working group was founded just before the publication of the original SDSI proposal [152]. However, SPKI quickly incorporated SDSI names and shares many other similarities with the SDSI proposal. In 1999 the projects were merged and are now referred to as simply SPKI.

S-expressions

S-expressions [153] were developed by Ron Rivest at MIT. S-expressions are lisp-like data structures that are used to represent complex data. They are either byte-strings (octet-strings) or lists of other S-expressions. The data in S-expressions can be represented in many formats, from simple strings to hexadecimal or base64 strings. S-expressions were designed to be a compact, human-readable efficient and transportable mechanism for storing data.

An octet string is a sequence of eight-bit octets. These can be represented in many formats including strings, quoted strings, base64, hexadecimal, and length prefixing "verbatim" encoding; these should all be interchangeable. For example, the string "abc" can be represented as `#616263#`, that is simply the hexadecimal form of the characters a (61), b (62) and c (63). The # marks surrounding the numbers are used to specify the data between is in hexadecimal form. Verbatim length prefixing encoding, represents the string with the length of the string prefixed to that string, for example, `3:abc`.

Lists of s-expressions are also s-expressions. In this case, these lists are made up of s-expressions. An s-expression is surrounded with parentheses, for example `(a (b c))` represents a s-expression that links `a` to the second s-expression `(b c)`. S-expressions are used to represent credentials in both SDSI and SPKI. S-expressions can be used to represent a variety of constructs:

- *sets* of elements;
- *ranges* of data, such as dates, time or numbers;
- *prefixes* of strings, used for comparisons, and
- *any* set constructs representing any s-expression out of a set of possible s-expressions.

These constructs can be used to store arbitrary data.

Local Names

The SDSI project introduced the idea of "*local naming*". In SDSI all principals (keys) are equal. Each key has its own name-space, as in PGP (Section 2.5.2). When a principal refers to another principal in their own name space, they define the name themselves: For example if Alice has a computer, then she calls it "Computer". Bob may also have a computer, and he may too refer to it as "Computer". As Alice and Bob are separate principals this is perfectly acceptable. However, how does Bob refer to Alice's Computer? Suppose Bob knows Alice simply as "Alice". Local naming provides the ability to use names from other namespace. Bob therefore refers to Alice's Computer as "Alice's Computer".

```
(Alice Computer)
(Bob Computer)

(Bob (Alice Computer))
```

Figure 2.11: Local Names using S-Expressions

If we take these local names and describe them using s-expressions, then we get a naming system such as the one shown in Figure 2.11. Informally we refer to these naming relationships using the "'s" construction (i.e., Bob's Alice's Computer). These namespaces can be built up to describe a hierarchy, for example, consider there is more than one Alice in a company. We could refer to her computer as Company's Department's Alice's Computer. This provides a flexible yet consistent method to name objects.

SPKI

The SPKI [54–56] is a certificate based system like KeyNote and PolicyMaker. A SPKI certificate is a signed statement consisting of five fields. The message without a signature (assertion in PolicyMaker), is called a 5-tuple. The five elements are: issuer; subject; delegation; authorisation, and validity dates.

The issuer and subject fields are mandatory, the remaining three are optional. These fields are expressed in the form of s-expressions. There are standard rules for converting s-expressions into binary format and back. These have the same purpose as the ASN.1 [99] notation. S-expressions are used instead of ASN.1 simply because they are lightweight expressions that are more suitable for lower specification hardware. The issuer field holds the key (or a hash value of the key) and a name associated with that key. This name is a SDSI local name.

The subject field also holds a SDSI local name, or a list of local names. The validity dates field contains the validity period of this certificate, in terms of a *not-before* and/or *not-after* date fields. The validity field may also include a number of "online test" expressions, that specify the certificate should be verified by checking a certificate revocation list (CRL) or an online validity list (a list of currently valid certificates). These are optional components of the field.

The authorisation field specifies the authority being delegated. This is roughly equivalent to the KeyNote `Conditions` field. Finally, the delegation field specifies whether the authorisations contained within this certificate can be delegated further.

Example 2.5 Figure 2.12 shows an example of a SPKI certificate, representing the `Appoint Lecturer` permission discussed in Example 2.3. In this example a certain key (the President's

```
(cert
 (issuer (hash sha1 |dsEFA73sahfdDF3784JDFjfsFsd=|))
 (subject (ref: UCC (ref: CSDEPT (hash sha1 |dasdk...|))))
 (propagate)
 (
  (tag (Appoint Lecturer))
 )
 (not-before "2002-11-31_17:00:00")
 (not-after "2003-11-31_16:59:59")
)
```

Figure 2.12: An example SPKI Certificate, authorising the appointing of academic staff.

key) is authorising a key in the local namespace `UCC's CSDEPT` to `Appoint Lecturer`. The `propagate` field allows this permission to be delegated further, and the validity dates specify that this certificate is valid for one year only. △

There are three types of SPKI credentials: identity, attribute and authorisation credentials.

- Identity credentials bind a public key to a name, similar to an X.509 certificate. However, unlike X.500 distinguished names, SPKI names are local to each public key in the system.
- Attribute credentials bind an authorisation to a name or group. These are generally used with identity credentials, as different issuers may generate the attribute and identity credentials.
- Authorisation credentials provide a means to delegate authority between names.

2.3.4 Advanced Trust Management Systems

The trust management systems outlined above (PolicyMaker, KeyNote and SDSI/SPKI) provide the means to specify general trust management policies for a variety of systems. In this section, we examine more specialised trust management systems, including SD3, REFEREE and QCM, and more advanced systems, including RT and DAL, that address specific problems with the traditional approach to trust management.

SD3

SD3 [97] is a Trust Management system consisting of a high level policy language, a local policy evaluator and a certificate retrieval system. Unlike trust management systems such as KeyNote or SPKI, SD3 has a built-in certificate distribution component, that allows a complete security infrastructure within SD3.

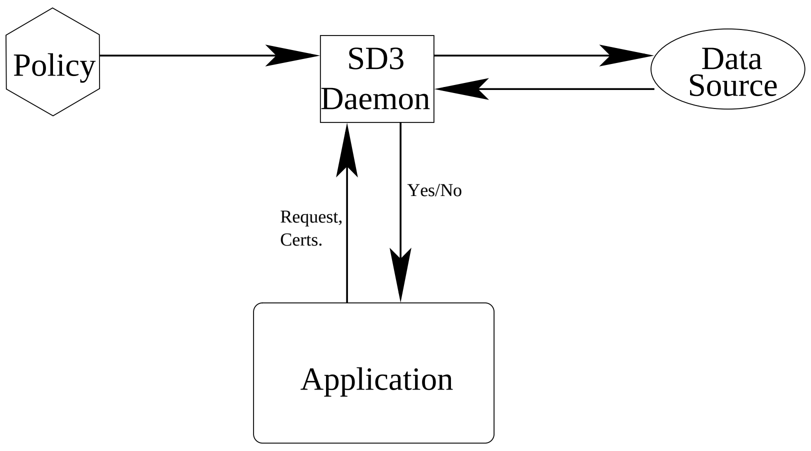


Figure 2.13: Overview of a SD3 Application

SD3 (“Secure Dynamically Distributed Datalog”) is an extension of the database programming language, datalog. Datalog [26] is a logic programming language specifically designed to be used as a database language. It is a nonprocedural, set-oriented language, with no order sensitivity, no special predicates and no function symbols. Datalog’s set of rules are logical implications. SD3 extends datalog into a trust management system by extending it with SDSI names.

SD3 provides an API that allows applications to query a daemon that maintains security policies. Figure 2.3.4 shows a overview of SD3 architecture. The SD3 daemon is initialised with the local policy. The application can then query the daemon for policy decisions. The application supplies a request and optionally some certificates it thinks appropriate for the query to the daemon. These are then examined by the daemon, the signatures are validated and a decision is reached determining whether or not the request is valid in accordance with the policy.

The local policy may depend on a remote policy. In this case the SD3 daemon will communicate with a remote data source (such as another SD3 daemon or some directory service, such as LDAP [52]) to check the request.

A unique and important feature of SD3 is its certified evaluator. In addition to computing the result of a query, the evaluator also computes a proof that the answer follows the policy. This proof is passed through a very simple checker before the result is reported by the evaluator. If the proof does not pass the checker, then the evaluator rejects the answer and reports an error. The fact that the checker is so simple, allows confidence in the correctness of the answer produced by the evaluator.

REFEREE

REFEREE [42, 44] (Rule controlled Environment For Evaluation of Rules and Everything Else) was created by Yang-Hua Chu et al as part of the World Wide Web Consortium’s (W3C) PICS (Platform for Internet Content Selection) project [176]. PICS is an effort to define meta-data to be associated with Internet content. This is intended to support parental control over access to web sites by children. REFEREE is designed to provide Trust Management support for the PICS project. The DSig project [43], that uses PICS labels, has also supported work on REFEREE. The DSig project looks at methods to provide a mechanism to make the statement: *signer* believes *statement* about an information resource.

REFEREE is based on the PolicyMaker trust management system. Similarly to PolicyMaker, REFEREE is a recommendation based query engine. In REFEREE, policies and credentials are programs. However, REFEREE differs from PolicyMaker in that it allows policies to control credential retrieval and signature verification. PolicyMaker makes the assumption that the calling application has gathered all the relevant credentials and verified all digital signatures before calling the trust management system.

REFEREE is designed with the principle that the policy controls everything, including the order of execution of a query and the retrieval of credentials. A policy has a fixed language syntax and may call other policies in order to satisfy a query. REFEREE supports three primitive datatypes: *Programs*, *Statement lists* and *Tri-values*. A tri-value is one of *true*, *false* and *unknown*. A statement list is a collection of assertions, or statements, expressed in a two element structure (Figure. 2.14). This structure consists of some *content* and a *context* for that content. These context and content attributes are specified as s-expressions [153]. The context determines how the content is to be interpreted. The interpretation of the context is subject to agreement between REFEREE and the calling application. Each program takes a statement list as an input and may take additional inputs. Programs may invoke additional programs during execution.

```
((‘‘certification module’’)
(‘‘Alice’’ (trustworthy yes)))
```

Figure 2.14: Example REFEREE statement indicating Alice is trustworthy in a certification module.

Policies are programs defining the suitability of certain actions, that return a tri-value for a query based on conformity or lack thereof to the stated properties of that policy. If a query can neither be satisfied or rejected based on the policy, then *unknown* is returned. Credentials are also programs that examine initial statements passed to them and derive additional statements. Unlike other trust management systems, REFEREE credentials generalise the usual concept of credentials as just supplying statements. These new statements supplied by a credential can be based on the initial statements or on environmental factors, such as the disk space remaining locally. Both policies and credentials return tri-values and statement lists. Policies can return a statement list that justifies the tri-value answer returned. Credentials return tri-values to indicate whether a execution was successful or not. Applications invoking REFEREE provide a database of available programs, an initial statement list and designates a particular program (policy) to run. It can also specify additional parameters to the program.

Policies calling other policies are central to the REFEREE system. Policies often defer judgement to other policies – Alice will trust a website because Bob trusts it. Evaluation of particular request may also require “dangerous” activity such as network access. These dangerous activities are allowed within policies, and these actions are controlled by other policies. One aspect of REFEREE’s creed is that everything is controlled by policies.

Profiles 0.92 is a rule-based trust policy language designed to work with REFEREE. Each rule is an s-expression with an operator as the first element followed by operands. The language includes support for a language construct *invoke* that supports calling of another REFEREE program. Invoked subprograms can return statements. These statements are prepended with the name of the subprogram. This feature allows tracing of a statement’s origin. “load labels” is another profiles-0.92 program that looks for PICS labels either embedded in documents or retrieved over the network.

These labels are parsed and converted into REFEREE statements. Profiles-0.92 also provides support for generalising the tri-values returned by REFEREE programs. This generalisation provides a means to convert a tri-value into a traditional Boolean value. Conversion is handled by using either the *false-if-unknown* or *true-if-unknown* constructs. There also exists support for the **AND**, **OR** and **NOT** Boolean operators within the language. Finally the language provides a statement-list pattern-matcher that can examine a statement list for statements of a particular form.

Example 2.6 Figure 2.15 shows an example of a REFEREE Policy. In this example policy we want to restrict users from accessing sites that don't follow UCC's acceptable use conditions. We trust website owners to declare honestly their rating according to UCC.

```
(invoke "load-label" STATEMENT-LIST URL
        "http://www.ucc.ie/ratings/acceptable.html"
        (EMBEDDED))
(false-if-unknown
  (match
    (("load-label" *)
     (* ((version "PICS-1.1") *
         (service
           "http://www.ucc.ie/ratings/acceptable.html") *
         (ratings (RESTRICT < uccrate 3)))))
    STATEMENT-LIST)
)
```

Figure 2.15: Example REFEREE Policy, enforcing UCC's acceptable use conditions.

This policy has two steps: First `load-label` is invoked to find and download labels for the embedded URL. Any labels found are added to the statement list. Next we run a pattern match with this statement list, looking for matches from the acceptable use rating service and with an `uccrate` rating of less than 3. If such a label is found, then a true or false is returned, based on the value associated with the label. If no such label is found, then the `false-if-unknown` condition forces a false to be returned. The result of this policy is to prevent viewing of a document (webpage) that has a `uccrate` rating of less than 3. △

QCM

QCM [82, 83] is a predecessor of SD3. QCM, or Query Certificate Manager, was proposed by Carl Gunter and Trevor Jim at the University of Pennsylvania. QCM was developed to address the problem of failed queries due to missing certificates. In a more conventional Trust Management system, such as KeyNote or SPKI, when an application queries the system, it provides all the certificates (credentials) needed to satisfy the query. However, if all of the required certificates are not present, then the query will fail due to insufficient information (In REFEREE, an *unknown* would be returned). QCM was developed to address this issue.

QCM uses what the authors of [83] call *policy-directed certificate retrieval* to determine what certificates to retrieve for a particular query. In previously existing trust management systems, if a query failed due to lack of relevant certificates, then a user (or application) would have to parse the policy and determine what additional certificates are required. In this case a three-fold duplication of effort is required: First the verifier must try to answer the query, when this failed, the policy must be parsed again to determine what is missing and finally the query must be resubmitted to the verifier. QCM attempts to eliminate this duplication.

The design of QCM was intended to draw upon the strengths of existing trust management systems and to add support for the automatic retrieval of relevant certificates. To address this aim, the developers chose a conservative approach and based the language on the language of sets that forms the core of some database languages. The language used (Caml [93]) serves as both the policy language and the query language of QCM. Verification of a QCM query, therefore, takes the form of a database evaluation and retrieval corresponds to a distributed database evaluation. QCM takes advantage of the extensive research into database query optimisations and distributed databases. This allows the system to form optimised queries to minimise message traffic. The design of the system around an existing database query language, allows users to write policies without specifically addressing remote queries. QCM automatically detects when a policy requires external certificates, formulates the correct query and retrieves the appropriate certificates.

Principals in QCM, as in PolicyMaker (Section 2.3.1), KeyNote (Section 2.3.2) and SDSI/SPKI (Section 2.3.3) are public keys. QCM natively uses SDSI linked local names, with support for global names as in SDSI. In QCM a global name refers to a set.

Example 2.7 Figure 2.16 shows an example of a QCM global name. In this example K$PKD is a global name referring to a set of (user, key) pairs. It states that K_{Alice} is Alice's key and K_{Bob} is Bob's key. K$PKD is the global name of PKD in K's namespace.

$$\text{K\$PKD} = \{(\text{"Alice"}, K_{Alice}), (\text{"Bob"}, K_{Bob})\}$$

Figure 2.16: Example QCM Global Name.

△

QCM can be used as a replacement for systems currently in place. For example, it was shown in [83] how a replacement for REFEREE could be built using QCM policies.

RT

RT [113] is a family of role based trust management frameworks, including RT_0, RT_1, RT_2, RT^T and RT^D. RT was created to combine the strengths of traditional trust management systems and those of role based access control. RT uses both the concepts of roles and the advantages of local naming to provide attribute-based access control. Traditional trust management systems, such as KeyNote or SDSI/SPKI, use credentials to delegate permissions (capabilities). These capability-based trust management systems, however, do not lend themselves to certain decentralised problems. For example, imagine an airline has an agreement with a rental car company that preferred customers of the airline receive special rental rates. This is difficult to cleanly represent in a traditional trust management system, as the airline does not want to give an external party access to its internal passenger database.

One approach is the rental company could delegate the "discount" permission to the airline, who then specifically delegates the permission to its customers. Another approach would be for the airline to create a new keypair to represent customers, and delegate all rights for this key to each customer. The rental company then delegates the discount to this key.

Neither of these approaches are particularly appealing as they cause significant overheads for one party or another. In the first case, the airline has the administrative overhead to determine what customers get the preferential rate. The second approach uses a separate keypair for each grouping. Every key pair must be distributed to all the parties in the system. For example, imagine that the rental company has different rates based on the type of airline customer. The airline now must create a key pair for each type of customer and each rental company. Such an approach can quickly become unreasonable.

RT introduces the concept of attribute based access control (ABAC). ABAC systems have a number of advantages over capability based systems, including (from [113]):

1. Decentralised attributes: an entity asserts that another entity has a certain attribute.

2. Delegation of attribute authority: an entity delegates the authority over an attribute to another entity, that is, the entity trusts another entity's judgement on the attribute.

3. Inference of attributes: an entity uses one attribute to make inferences about another attribute.

4. Attribute fields. It is often useful to have attribute credentials carry field values, such as age and credit limit. It is also useful to infer additional attributes based on these field values and to delegate attribute authority to a certain entity only for certain specific field values, for example, only when spending level is below a certain limit.

5. Attribute-based delegation of attribute authority. A key to an ABAC's scalability is the ability to delegate to strangers whose trustworthiness is determined based on their own certified attributes. For example, one may delegate the authority on (identifying) students to entities that are certified universities, and delegate the authority on universities to an accrediting board. By doing so, one avoids having to know all the universities.

The RT framework supports localised authority over roles; delegation in role definitions; linked roles; parameterised roles, and manifold roles. The different systems use a subset of these abilities.

In particular, RT_0 only allows atomic strings as role names; RT_1 extends RT_0 to allows parameterised roles. This is useful as the same roles in different domains often hold the same permissions. For example, manager roles hold the permission to set the salary for their employees. Parameterising this role allows the same salary permission to be applied to different manager roles within an organisation, with respect to their employees.

RT_2 extends RT_1 with the notion of o-sets which group logically related objects and access modes together similarly to parameterised roles. In the case of o-sets, it is sometimes useful to be able to apply the same permissions to sets of objects and access modes.

RT^T provides support for threshold schemes to the RT framework. Threshold policies define that more than one entity must agree before an action is authorised. For example, in a simple payment system, a cheque must be signed by two separate keys before it can be cashed.

In certain cases, an entity may not wish to use all of their rights all of the time. A simple scenario is where an administrator logs in to a system as an ordinary user so that they cannot make a catastrophic mistake. RT^D provides the ability to handle delegation of the capacity to exercise role memberships. This allows an entity to delegate parts of their role permissions to particular processes. In traditional trust management systems, this is not possible. An entity implicitly uses all of their rights in every request. They could reduce the number of credentials supplied when making the authorisation check, however, in a truly distributed system this may not be possible. RT^D adds the notion of delegation of role activations to support selective delegation of permissions to processes.

An entity in RT defines an uniquely identified individual or process. They can issue credentials and make requests. Roles in RT define a set of entities who are members of that role. They can be viewed as an attribute. Entities in RT correspond to users in RBAC systems. Roles can represent both roles and permissions in an RBAC system. RT views user and role assignments as dominations ($\succeq$):

Roles	$r_1 \succeq r_2$	defines that r_1 has every permission in r_2
Users to Roles	$u \succeq r$	defines that user u is assigned to role r
Permissions to Roles	$r \succeq p$	defines that permission p is assigned to role r

A role is denoted as $A.R$, where A is and entity and R is a role. $A.R$ can be considered *A's R*, similar to local naming in SDSI/SPKI. In this case, only A has the authority to assign members to the role R. This is achieved by issuing role definition credentials to the other entity. Each credential defines one role to contain either an entity, another role or certain other expressions that evaluate to a set of entities. A role may be defined by multiple credentials. The effect of such multiple role credentials is a union. Credentials used in RT are defined in terms of delegations ($\longleftarrow$), as shown in Figure 2.17.

RT supports common vocabularies between entities. If A defines $A.R$ to contain $B.R_1$, then A must understand what B means by the role name R_1. This is achieved in RT through the use

$A.R_1$	$\leftarrow A.R_2$	A defines that R_2 dominates R_1.
$A.R$	$\leftarrow B.R$	defines that A delegates authority over R to B.
$A.R_1$	$\leftarrow B.R_2$	defines a mapping between two organisations, A and B, and the roles they provide.
$A.R$	$\leftarrow A.R_1.R_2$	Using the role mapping credential above, this credential defines that $A.R$ contains any $B.R_2$ if $A.R_1$ contains B.
$A.R_1$	$\leftarrow B_1.R_1 \cap B_2.R_2$	defines that $A.R_1$ contains roles $B_1.R_1$ and $B_2.R_2$. This is the intersection of two credentials.

Figure 2.17: The different types of credentials in RT.

of application domain specification documents (ADSD). ADSDs define a suite of related datatypes, role identifiers (Role IDs) with the name and datatype of each parameter. This forms the vocabulary. ADSDs may also declare other common characteristics of Role IDs such as storage type information. ADSDs generally provide natural language descriptions of role identifiers, including the conditions under which they should be issued by the credentials defining them. These credentials also have a preamble where they specify the ADSD that is used with the credential, typically by specifying its uniform resource identifier (URI) [177].

When role identifiers are used in credentials, the vocabulary identifiers are incorporated as prefixes to the role identifier. While role identifiers are relatively short, they specify a globally unique Role ID. ADSDs can be linked together in order for one ADSD to use the data types defined in another.

DAL

DAL [186], or Distributed Authorisation Language, was specifically designed to avoid the problem of authorisation subterfuge. Subterfuge [68] is the ability of an entity to illicitly gain authorisation using malformed delegation chains. If two entities use the same permission in different ways within a coalition, then it is possible that one entity can misuse this permission in some unexpected way.

In order to avoid such problems, common vocabularies are used to provide a globally unique name space for use in credentials, such as ADSDs in RT or X.500 names in X.509. However, these approaches rely on *closed* delegation, that is delegation between users in coalitions that are effectively controlled by a single administrator. Ensuring that delegation credentials created in these schemes are subterfuge free requires formal analysis and/or providing pre-agreed global naming services.

DAL statements represent facts held by entities: identifiers, roles and threshold entities. DAL statements are made using basic logic operators, functions and *says* ($\mid\!\sim$) and *directly says* ($\|\!\sim$) operators. Identifiers represent global unique entities and are denoted by the triple (K, N, T), where K is the entity's signature key; N is a descriptive name for the entity, and T is the type (individual

(I) or coalition (C)) of the entity. Typically this triple is represented by the structure ID^T, where ID specifies the global identifier containing K and N.

Example 2.8 A simple DAL certificate is as follows:

$$Alice^I \mid\!\sim actAs(UnivA^C.student, Bob^I)$$

This defines that Alice *says* that Bob is a student of UnivA. This can be stated more informally as: Alice, who is an individual, signs a certificate stating that Bob, who is also an individual, is a student at University A (UnivA), which is a coalition. △

DAL has a number of advantages over other trust management systems. DAL provides an built-in proof system that ensures that statements are subterfuge free. As with RT, DAL supports role based authority and role based delegation.

2.4 Authentication

Authentication is the process by which entities can determine one another's identity, and use this information to establish a secured communication between the entities. Identities are usually stored as crytographically signed certificates, and are typically represented as cryptographic keys. In this dissertation, when we refer to authentication, we are specifically talking about *entity authentication*. Entity authentication is defined as (from [76]):

> *Entity authentication mechanisms allow the verification of an entity's claimed identity, by another entity. The authenticity of the entity can only be ascertained only for the instance of the authentication exchange.*

Two users, Alice and Bob, wish to communicate across a network. When Bob receives a message from Alice, how does he know it comes from Alice and not an attacker, Eve? This is the fundamental problem that authentication protocols address. Furthermore, can Alice and Bob create a shared encryption key so that they may communicate in private, without Eve intercepting their messages. These problems are addressed using authentication and key-exchange protocols.

2.4.1 Simple Authentication Protocol

A simple challenge-response authentication protocol is shown in Figure 2.18. In this protocol, when a client wants to access a server, it sends its name, C. The server then sends the client a challenge in the form of a nonce encrypted with a key shared by the client and server, K_{SC}. If the client can then send the server the nonce, incremented by one and encrypted using the shared key, then the client has been authenticated by the server.

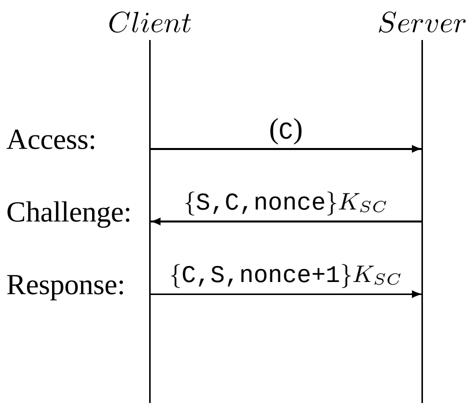


Figure 2.18: Challenge-Response Authentication Protocol

The challenge response protocol provides single-side authentication, that is the server has authenticated the client, but the client has no guarantee that it is talking to the correct server. There are many different authentication protocols providing a range of authentication guarantees. For example, some protocols provide single side authentication, while others provide two-way authentication. We now briefly examine two commonly used key exchange protocols, SSL/TLS and Kerberos.

2.4.2 SSL/TLS

SSL [92], or secure sockets layer, was originally developed by Netscape to provide authentication between web browsers and web servers on the Internet. It is probably the most common security protocol in use today. It is primarily used to provide secure communication between users and online shops, in order that personal and financial information is sent in an encrypted form between users and businesses. The SSL protocol (version 3) was submitted as a standard to the IETF, and when accepted was renamed transport layer security or TLS. Thus, SSL version 3 is virtually identical to TLS version 1. (We will refer to SSL/TLS henceforth simply as SSL).

SSL uses X.509 certificates, described in Section 2.5.1, to link identities to public keys. Each entity must share trusted certificates to be used to find a chain of trust to the other entity's certificate. Usually each entity will have *"root"* certificates from certification authorities (CA) whom they trust to properly verify the identities of the entitys they write certificates for. In order for a certificate to be trusted by a entity, there must exist a certificate chain between a known trusted certificate and the certificate provided.

In general SSL can operate in either client-side authorisation, where just the client verifies the identity of the server, or in client and server side authentication, where both sides authenticate each other's identity.

2.4.3 Kerberos

Kerberos [50, 138, 161] is an authentication protocol that uses a trusted third party to allow clients to authenticate themselves, and thus gain access to services on the network. The protocol uses two trusted parties, one to hold the shared encryption keys, and the other to control access to protected services on the network. The Kerberos model is based on the Needham-Schroeder trusted third party protocol [136]. The Kerberos server keeps a database of clients and their secret keys. Clients can be users or even software programs running on machines in the network. Clients requiring authentication, register their keys with the Kerberos server. As the Kerberos knows the secret keys of all the clients on the network, it can create messages that can convince one client of another client's identity.

Using the Kerberos protocol, when a client wants access to a protected service on the network, they must contact the ticket granting service. The ticket granting service (TGS) can grant authenticated clients access to a service. First however, the client must be authenticated to the TGS. This is achieved using the Kerberos server as a trusted third party. The client contacts the Kerberos server and asks to be authenticated to the TGS. The server sends a *"ticket granting ticket"* (TGT), or a message containing a session key, or ticket, encrypted using the client's secret key and also encrypted using the key shared by the Kerberos server and the TGS to the client. The client sends a message containing the TGT to the TGS, with its access request. The client and the TGS now share a secret, that is, the session key generated by the Kerberos server.

If the TGS determines that the access to the service is authorised, then it generates a new ticket for the client, containing a session key that the client and service will share (as before). This ticket will then be presented to the service by the client. In general, the ticket granting ticket will be long lived, for example it could be valid for a day, but the service ticket would be a more short-lived ticket, for example, suitable for a single access to the service.

Kerberos provides the ability to securely authenticate clients and servers using a trusted third party. Kerberos is used in a variety of applications, such as MIT's Project Athena [139] and most recently, a modified version is used by Microsoft to provide authentication in their Windows networking, since the release of Windows 2000 [50]. Kerberos is more than a simple authentication scheme. It also provides an access control mechanism to its users.

2.5 Other Security Technologies

There are many different certificate based access control technologies currently in use. In this section we will examine the most important of these technologies, including X.509 and several trust management systems. Each of these systems provide a means to attach attributes to public keys. In general, we refer to attribute certificates as *credentials*

Using certificated-based access control allows the creation of a decentralised security architecture. Credentials are portable, and can be presented by the subjects attempting to access security critical objects to prove they are authorised to access them. Access is granted only when the reference monitor controlling access to the object receives credentials sufficient to authorise access to the object according to the system policy.

2.5.1 X.509

X.509 [38, 70, 87] is one of the most ubiquitous security technologies in use today. It is the authentication framework designed to support the X.500 [182] directory services. Both X.500 and X.509 are international standards proposed by the ISO and ITU. X.500 is designed to meet the directory service requirements of large computer networks.

The naming service is rigidly hierarchical, that is best suited for large corporations and governments, where such a structure is common. X.509 provides a PKI [39] (Public Key Infrastructure) framework for authenticating X.500 services. X.500 directories have a tree-like structure (and will be described in Chapter 3). The root of the tree has branches to each of the countries. Each country has organisations, these form the next set of sub-branches. Organisation are made up of organisational units, that have users.

Example 2.9 A *distinguished name* for a user (Alice) in the Computer Science department in University College Cork is:

```
Distinguished Name (DN) : {
 Country (C) = IE,
 Organisation (O) = UCC,
 Organisational Unit (OU) = CS,
 Common Name (CN) = Alice
}
```

△

The X.509 PKI standard was originally proposed in 1988. It was the first attempt at standardising a PKI available at the time. It was developed to support the authentication of entries in an X.500 directory. Version 3 is the current standard. An X.509v3 certificate is shown in Figure 2.19. The certificate binds an identity to a key. The serial number is unique and is issued by the Certification Authority (CA). The CA and Subject names are X.500 names. Due to the close relationship with X.500, CA hierarchies generally follow X.500 hierarchies.

Example 2.10 Figure 2.19 displays how the issuing certification authority (*CS Root CA*) has created this certificate for *Alice User* in the Computer Science department in UCC.
This certificate is valid for one year from January 7^{th} 2002 at 18:06:51. △

```
Certificate:
 Data:
  Version: 3 (0x2)
  Serial Number: 1 (0x1)
  Signature Algorithm: dsaWithSHA1
  Issuer:C=IE,ST=Munster,L=Cork,O=UCC,OU=CS Dept,
         CN=CS Root CA/Email=rootca@cs.ucc.ie
  Validity
   Not Before: Jan  7 18:06:51 2002 GMT
   Not After : Jan  7 18:06:51 2003 GMT
  Subject:C=IE,ST=Munster,O=UC Cork,OU=CS,CN=Alice User
  Subject Public Key Info:
   Public Key Algorithm: dsaEncryption
    DSA Public Key:
    [..]
   X509v3 extensions:
    X509v3 Basic Constraints:
     CA:FALSE
    Netscape Comment:
     OpenSSL Generated Certificate
    X509v3 Subject Key Identifier:
     61:D0:B5:4A:0F:CA:1E:B0:49:59:73:59
    X509v3 Authority Key Identifier:
     keyid:0B9:2F:63:F6:26:ED:72:8F:C9:8C
    DirName:/C=IE/ST=Munster/L=Cork/O=UCC/OU=CS Dept/
            CN=CS Root CA/Email=rootca@cs.ucc.ie
    serial:00
  Signature Algorithm: dsaWithSHA1
  [..]
```

Figure 2.19: An example X.509v3 certificate

In order for a certificate to be valid it must be presented during the validity period, and must not have been revoked. Certificates are revoked using Certificate Revocation Lists (CRL). These CRLs are issued by the CA's periodically. When a user wishes to check whether a certificate is valid, they should also contact the issuing CA to procure the latest CRL and ensure the certificate has not been revoked. CRLs provide an additional layer of complexity to X.509. Not only does the user have to check if the certificate is semantically valid, they must also contact the CA to ensure the certification still authorised to.

X.509v3 also supports *certificate policies* that give CAs the ability to include a list of policies followed when the certificate was created. For example a certificate might be valid to support online email reading but not for online financial transactions. X.509 is widely used within Internet applications. It is, perhaps, best known as providing the infrastructure for securing websites with Netscape's SSL [92] protocol. SSL uses X.509v3 certificates to enable users to verify the identity of the website they are connected to. X.509 (version 1) has also been used to support privacy enhanced mail (PEM) [22, 100, 105, 114]. PEM is a proposed standard that provides encryption in email.

X.509 defines a rigid hierarchical structure, ideally with one super certification authority. In practice, the each major certification authority considers themselves the top of the hierarchy. This means that cross-CA certificates must be created for every pair of CAs. Furthermore, the compromise of a CA private key is extremely serious. Should this occur, every certificate that is signed by that key is potentially suspect. Revocation of an entire tree, plus re-keying is a daunting task. X.509 is not a security panacea. It is one useful tool that can be used to develop secure systems.

2.5.2 PGP

PGP [38, 116, 187] or pretty good privacy was developed by Philip Zimmerman as a secure asymmetric encryption system for the common man. PGP introduced the idea that each user is their own certification authority. This leads to a very ad hoc PKI system, unlike the rigid structure of X.509.

An important feature of the PGP system is its "web of trust". As each user is a certification authority, they certify others according to personal knowledge. The system works as follows: every user creates his/her own encryption key pairs. They then go to friends, who are also using PGP, and get their key "certified" (signed) by those friends. These friends will set a level of trust along with their signature, ranging from untrusted to fully trusted.

This is intended to build links from any one key to any other key in the system. When receiving a message from an unknown key, a user can decide to trust the identity of the message's signer based on the quality of the links between their known (and fully trusted) keys and the message signer's key. For example they could require two separate paths at a marginal trust level or one completely trusted path.

2.5.3 Secure Mobile Code

Mobile code [155], such as Java applets [79, 170], are executable components that run in remote locations. Unlike mobile agents, mobile code does not move from location to location, instead it is downloaded by the user and executed on their machine. For example, during a Formula 1 GrandPrix, interested viewers can execute a Java applet that provides live race information, such as driver lap times [13]. As mobile code is foreign code to the user, they must be confident that this code is not malicious. There are several techniques used to assure the user of the safety of the mobile code.

One of the primary solutions towards securing mobile code is to execute any remote code in a protection domain or *sandbox*. A sandbox limits the set of operations that the remote code may call. For example, Java's applet sandbox prevents remote code from creating a socket connection to any remote machine other than the machine that provided the applet. Sandboxing mobile code allows users to specify the mobile code they trust and what operations that the code is allows execute on the local machine.

Another solution is to digitally sign mobile code. Signing code reassures the user that the code was produced by a reputable software manufacturer. Thus, the user should only execute code from software makers that they trust not to provide malicious code. However, this approach has disadvantages. Digital signing certificates have been issued to people masquerading as a representative of a well known software maker [69]. Furthermore, small and open source software makers may not have the financial capability to purchase such signing certificates.

Chapter 3

Distributed Naming

Naming objects is a common requirement for many systems. Naming an object allows that object to be correctly identified by different entities in the system. For example, in in an email application, users are identified by their email address. One of the early distributed naming services for the Internet was the domain name service (DNS) [121]. DNS links recognisable strings to the numeric identifiers of machines on the Internet and provides simplified access for users to distributed resources.

Naming is a particular challenge for distributed systems. Each part of the distributed system must be able to refer to objects in the system. This is a common problem in computing, ranging from properly identifying users to providing a persistent means to identify objects on the Internet, for example, using digital object identifiers (DOI) [73]. Distributed computing relies on having names for objects in the system. For example, in order to access objects in CORBA, we must be able to uniquely refer to them.

We argue that when an object is properly named, it is then possible to make informed security decisions about that object. This philosophy is evident in the security architectures of each of the naming systems described in this chapter. For example, in order to make an access control decision about a CORBA object, one must first identify the object in question.

Developing names for objects is vital when referencing those objects. Object names range from simple descriptions to globally unique references. This is a common challenge in computing. For example, determining the exact version of a document available on the Internet can be vital in understanding information citing that document.

Developing a naming architecture for a distributed system entails having a means to represent the aspects of the objects in that system and providing a usable reference to the object for the system. Research in distributed names has concentrated around directory naming services such as LDAP and X.500 directories, and in object naming services such as CORBA names. We examine these topics separately.

In this Chapter we describe some approaches that are used to name objects in distributed systems. We examine directory naming services, specifically the classic Internet standard X.500 directory naming service and the increasing popular LDAP (lightweight directory access protocol) in Section 3.1. Section 3.2 investigates object naming services, specifically how the CORBA and Spring systems manage names for objects. Other types of naming systems are briefly examined in Section 3.3.

3.1 Directory Naming

Directory naming services are used primarily to store relatively static information, such as telephone and email directories. Such systems are optimised for reading, rather than writing. The most prevalent directory naming systems in use are X.500 and LDAP. We examine these systems in this section.

3.1.1 X.500

X.500 [179, 182] defines a directory naming service that is designed to meet the directory service requirements of large computer networks. X.500 was designed as a client / server architecture. Clients query a server that holds the directory information. X.500 is a decentralised system, where each site running X.500 is responsible only for the local part of the directory. The naming service is rigidly hierarchical. Such a structure is more appropriate for large corporations and governments. Another example is the X.509 PKI [39] (Public Key Infrastructure) framework for authenticating X.500 named services that was discussed in Chapter 2. X.500 provides a single homogeneous global namespace. X.500 directories have a tree-like structure (Figure 3.1). The root of the tree has branches to each of the countries. Each country has organisations, these form the next set of sub-branches. Organisation are made up of organisational units, which have users. Figure 3.1 gives an example of this structure.

X.500 names are based on a the perceived structure of an organisation for example, a government or a large company. Such organisations can be broken down as follows: country (C); state (ST); locality (L); organisation (O); organisational unit (OU), and common name (CN). The common name is defined by the organisational unit and the organisational unit is defined by the organisation. The country, state and locality define where the organisation is physically located.

While X.500 names are commonly used, their rigid structure makes them useful only for specific applications, such as a telephone or email directory in large organisations. In practice, X.500 names are primarily used to identify the entities referred to in X.509 certificates (see Chapter 2), and in large organisations. Smaller organisations, for example, a small company without internal organisational units, would find it difficult to justify the use of an X.500 based directory. X.500's

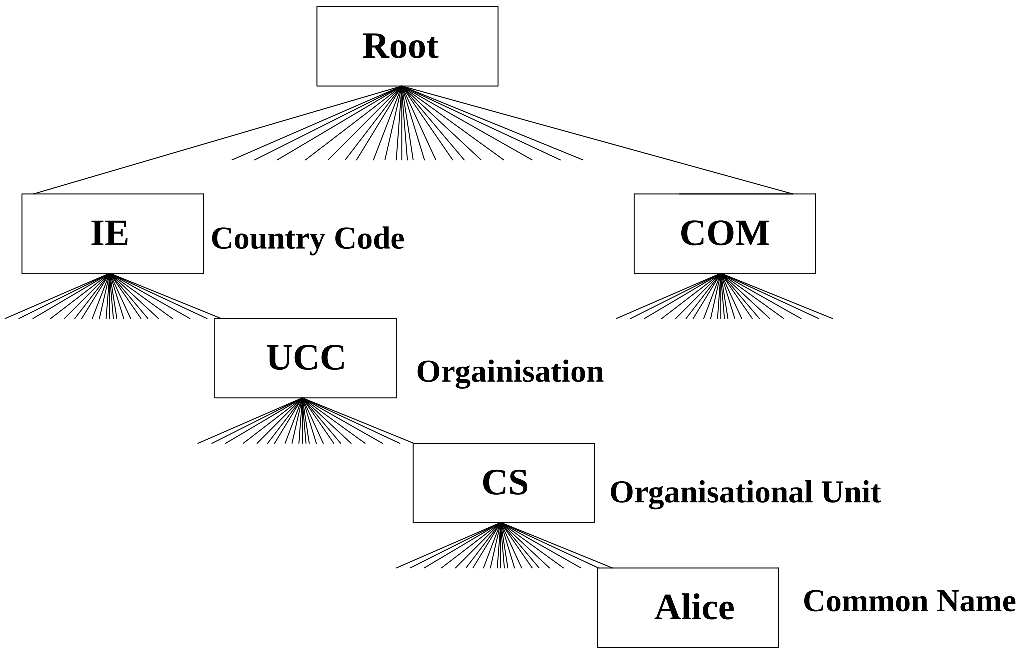


Figure 3.1: X.500 Directory Information Tree

decentralised structure does not suit every application. For example, it is difficult to imagine using an X.500 directory to store unstructured information.

3.1.2 LDAP

The Lightweight Directory Access Protocol (LDAP) [49, 89, 107, 178, 184] is based on X.500 directories, but is designed to be simpler and more customisable. LDAP is commonly referred to as a database, although this analogy is not completely accurate. Implementations of LDAP directories are typically optimised for read performance as LDAP is primarily used for looking up data, rather than updating data. Thus, LDAP is well suited for storing data that is not frequently changed, for example, a telephone directory. LDAP is extremely flexible. Any type of data may be stored in an LDAP directory. The structure of the directory is application specific, but in general follows a X.500 type layout, using X.500 headings.

Data is stored in LDAP in a hierarchical structure, similar to a X.500 directory. Figure 3.2 shows an sample directory tree layout for University College Cork. This example shows the layout without any data. In X.500 directories, the organisation unit (`ou`) name was used to distinguish the functional areas within a company.

The distinguished name for each LDAP entry is made up of two parts, the relative distinguished name (RDN) and the location within the directory where the data is stored. The relative distinguished name is the portion of the distinguished name that is not related to the directory structure, commonly stored in the `cn` or common name attribute. The root of each entry, called the *Base DN*,

```
  dc=ucc dc=ie
      ou=science
          ou=chemistry
          ou=computerscience
          ou=physics
      ou=humanities
          ou=law
...
```

Figure 3.2: A sample UCC LDAP directory tree

is stored in the `dc` field. This is typically a representation of the DNS entry for the organisation.

Example 3.1 A machine in the computer science department in UCC has the following entry (distinguished name) in an LDAP directory:

`cn=ceres,ou=computerscience,ou=science,dc=ucc,dc=ie`

In this case the base of the directory is `dc=ucc, dc=ie`. The record of the machine name is stored in `ou=computerscience, ou=science`. The relative distinguished name of this LDAP record is `cn=ceres`. △

```
  dn: cn=ceres,ou=computerscience,ou=science,dc=ucc,dc=ie
  cn: Ceres
  machineType: Workstation
  machineComponent: Pentium IV
  machineComponent: 1024MB Ram
  machineComponent: 80GB Hard Disk
  machineComponent: Gigabit Ethernet
  machineComponent: NVidia Graphics Card
  machineSoftware: Debian Linux
  machineSoftware: KDE
  machineSoftware: Xorg
```

Figure 3.3: Information about Ceres stored in an LDAP directory.

The entries in an LDAP directory can be customised for the specific requirements of an application. For example, Figure 3.3 shows the information stored in an LDAP directory about the machine `ceres`. This information might be used for auditing or statistical reasons. The fields `machineComponent` and `machineSoftware` are application specific, in this case to store the hardware and software components of the machine respectively. LDAP directories are designed to store multiple values of a simple type (for example, `machineComponent`) in this manner, rather than in the familiar row and column layout of a relational database. LDAP provides a flexible means to store (ideally) static information that can be quickly retrieved.

3.2 Object Naming

Object naming systems link references to objects on a system. Typically, this reference is passed to the component that wishes to use the object. For example, CORBA objects are represented by an object name. When an application wishes to use a CORBA object, it looks up the reference and uses the reference to access the object. Objects names are typically relatively static, in that objects are long-lived and their names do not often change. In this section, we examine some object naming services, specifically, the Spring naming service and CORBA object names.

3.2.1 Spring Naming Service

The Spring [120, 137, 149, 166] operating system was an experimental microkernel [171] based operating system designed by Sun to replace its Unix operating system. One of the basic design aspects of Unix is its concept that "everything is a file". Thus, directories are files, devices are files, and so on. However, this paradigm does not fit every aspect of a Unix system. For example, the capabilities of printers are represented in a printer specific namespace.

To address this limitation, subsystems in Unix have type-specific name services, such as the printer capabilities (/etc/printcap) or environment variables. Furthermore, distributed services of Unix, such as NIS or NFS, must also have a means to refer to objects distributed across a network. These systems use directory services to bind objects to names. The spring system provides a specific name service to support the requirements of the different subsystems of the operating system. This naming service provides users and normal Unix programs uniform naming access to most types of Unix objects.

Spring is an extensible distributed operating system that is inherently multi-threaded. It is structured around the concept of objects that act as *"an abstraction that contains state and provides a set of operations to manipulate that state"* [137]. Spring provides the concept of a *domain*, that is an address space with a set of threads. In a distributed system with multiple domains, Spring provides an unforgeable nucleus *door identifier* that identifies the server domain.

The Spring name service allows any object to be bound to any name. These name binding are stored in a *context*. A context is an object that stores one or more unique name bindings. A simple example of a context is a Unix directory file. Each file in the directory is an object-name binding and the directory file stores these bindings. Objects can be bound to more than one unique name at a time. The Spring name service provides the capability to bind objects to names and to resolve the name for any object. As contexts are themselves objects, they can also be bound to names. This leads to *naming graphs*. For example, the Unix file system is such a naming graph. An example of such a graph is shown in Figure 3.4. In a naming graph, directory contexts are bound to names (the directory name) and these contexts stored in another directory file (the parent directory).

Names in the Spring system provide the basis for the Spring security model. Objects that are

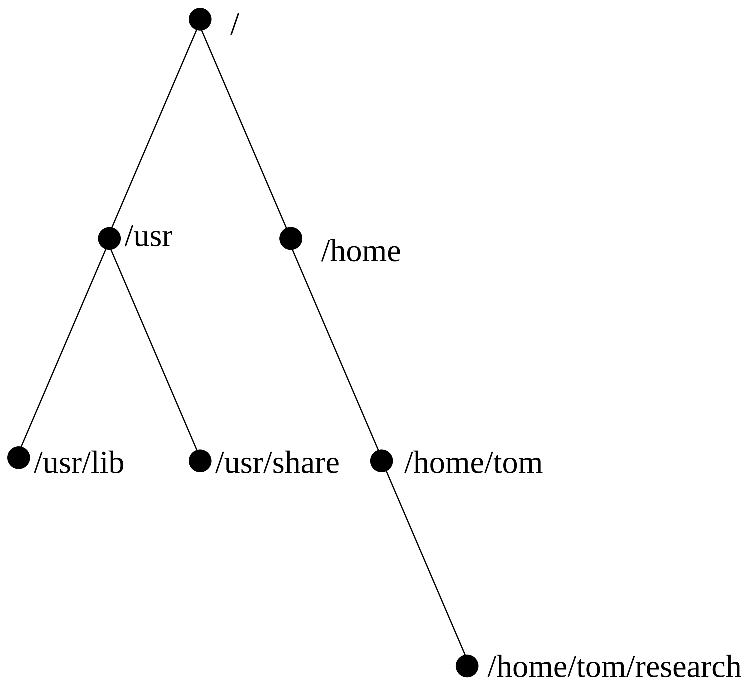


Figure 3.4: A File System Naming Graph

bound can be associated with access control lists (ACL). ACLs specify the access rights that principals have to objects within the system. One of the major advantages of this paradigm is that names in the Spring system provide the detail to specify security policies for all of the different subsystems that make up the Spring operating system.

3.2.2 CORBA Names

Common Object Request Broker Architecture (CORBA) [28, 81, 140] is a application component system that is specified and standardised by the Object Management Group (OMG) [8]. CORBA defines an API, communication protocol and object management system to enable heterogeneous components to inter-operate on various systems. CORBA objects can be thought of as services that are used by applications. In general, CORBA wraps code to provides a standard interface to that code for distribution across a network.

An important aspect of CORBA is how these objects are named. CORBA objects may be distributed across many different CORBA servers (called object request brokers or ORBs). The provision of means to refer to objects on remote systems allows these objects to be used. In most object orientated systems, objects have references that are used internally to identify them. In CORBA, objects are used remotely, a more systematic approach is required.

CORBA's naming service, like the Spring system, relies on the concepts of name to object associations called name bindings. Name bindings in CORBA are defined relative to naming contexts.

As with Spring name contexts, CORBA name contexts contain a set of name bindings in which each name is unique. Many different names can be bound to a single object. However, there is no requirement that every object have a name. Contexts can also be named, allowing the creation of naming graphs. CORBA uses linked contexts to form compound names to refer to an object. A compound name defines the path in the naming graph that leads to an object.

Names in CORBA are made up of a sequence of name components. Name components consist of two attributes: the *id* attribute and the *kind* attribute. Both the id attribute and the kind attribute are represented as interface definition language (IDL) strings. Kind attribute provides a textual description of the name. For example, a kind attribute could be "c_source" or "executable". Kind attributes are not interpreted by the naming system. Both attributes are arbitrary length ASCII strings. Name components cannot be empty and a name must consist of at least one name component. In contrast id attributes store the reference to the object in question.

As with the Spring system, the CORBA security model uses object names to form the basis of the access control system. Permissions are specified regarding object names.

3.3 Other Naming Systems

We have only examined a small sample of the naming systems currently in use. Other naming systems include Internet based naming services, such as DNS and Microsoft's Windows Internet Naming Service (WINS) [53], that link names to physical addresses on a network. In Chapter 2, we examined SDSI local naming. Recall that in SDSI all principals (keys) are equal and each key has its own name-space. When a principal refers to another principal in their own name space, they define the name themselves. Local naming has the advantage that arbitrary precise names can be represented within a name. Namespaces can be linked together to allow one principal to refer to objects defined by other principals.

Abadi et al [11] describe a tree naming scheme that uses certificates to provide context. In this system, only certified names would be allows in the trusted namespace. For example, if an application sought the name "/bin/ls", then it would have to provide a certificate that the administrator trusted so that the name would be assigned. This scheme is primarily aimed at operating systems, where specific namespaces have special meaning. System policies specify trusted namespaces in term of regular expressions that are used as permissions in the certificates.

Howell [88] proposes another tree based naming hierarchy that attempts to provide highly semantic and human readable (mnemonic) names for objects in a distributed system. In this system, names are bound to objects by users. These names provide symbolic connections to objects, rather like links in the Unix file system.

3.4 Discussion and Conclusions

In this chapter we have discussed several naming systems, ranging from structured naming systems (X.500 and LDAP) to flexible naming graph based systems (Spring and CORBA). Structural names have the advantage that the naming service can be easily decentralised. Names in a directory under a particular branch are controlled by the owner of that branch. However, this also limits the ability of such naming architectures to adapt for use in distributed environments. In contrast, flexible naming systems provide the ability to refer to complex compound names. Names can be linked together into naming graphs.

One similarity of all of these naming systems is that the objects they reference are static in nature. In the case of X.500 and LDAP, object names do not often change. In both CORBA and the Spring systems, while data contained in the objects may change, the names themselves do not change. Another approach to distributed names is the concept of SDSI local naming, discussed in Chapter 2. Unlike the naming services outlined in this chapter, SDSI names are capable of storing a more dynamic reference to an object.

In Chapter 6, we will develop a naming service for the condensed graph architecture that will be described in Chapter 4. A primary difference of the naming requirements for condensed graphs to the naming systems described in this chapter is the dynamic nature of condensed graphs. As the structure of a condensed graphs application evolves during execution, it requires a dynamic approach to naming. For this reason, we base our approach on SDSI-like local naming.

Chapter 4

Condensed Graphs and Distributed Computing

Condensed Graphs [122] is a graph based computational model. Applications are codified as graphs. Nodes in a graph correspond to operations, the arcs represent data paths between operations. The model supports three computational paradigms, which control the execution order of a graph.

- *Imperative* computation [175], where the sequencing of the operations in the graph drives the execution order. This corresponds to the traditional control-driven approach.
- *Eager* computation [102], where node execution is determined by the availability of parameter data to the nodes. This is equivalent to dataflow and is considered data-driven evaluation.
- *Lazy* computation [20, 143], where execution sequencing is results driven. It corresponds to the functional approach and is considered demand-driven evaluation.

Condensed Graphs may be used as a distributed job control language to describe the scheduling of operations in an application. Atomic operations are value-transforming actions and can be defined at any level of granularity, ranging from low-level machine instructions to mobile-code programs. Examples include computational primitives, Web Services [148], CORBA objects [28, 81] and commercial-off-the-shelf (COTS) components [118]. Atomic operations in a condensed graph application need not address synchronisation or concurrency concerns: such details are implicitly specified by the arcs between nodes and are managed by the condensed graph execution scheduler.

In this chapter, we describe both the condensed graph model and its execution engine. In Section 4.1 we examine the condensed graph computational model and investigate its execution mechanism. The Triple Manager [131] is used to schedule and execute condensed graphs. This is described in Section 4.2.

4.1 Computational Model

The condensed graph model unifies three different models of computation (imperative, lazy and eager) within a single graph based model. In the traditional imperative (control-driven) model of computation [175], the programmer explicitly determines the scheduling constraints of the computation. In the eager (data-driven) approach [12], sequencing is determined based on the arrival of data to the computational components. In the lazy (availability driven) approach, components are executed based on the need for their results. Both the data driven and availability driven [143] approaches are typically represented in the literature as directed acyclic graphs (DAG), for example, dataflow graphs [18].

Definition 4.1 *Condensed Graph.* A condensed graph, henceforth referred to as a graph, is a directed acyclic graph, representing an application and/or program. A condensed graph is made up of a collection of condensed nodes that are connected by arcs. A graph has a single *Enter* node (E) defining where operands enter the graph and a single *Exit* node (X) defining where results flow out of the graph. ♢

Definition 4.2 *Arc.* Arcs are directional connections between condensed nodes along which data flows, in the graph. ♢

Definition 4.3 *Port.* A port is a point on a condensed node where other condensed nodes can attach, via arcs. ♢

Ports on a node are where data from other condensed nodes enter and data for other condensed nodes exit. Operands enter through operand ports, operators enter through the operator port and data exits the condensed node via destination ports.

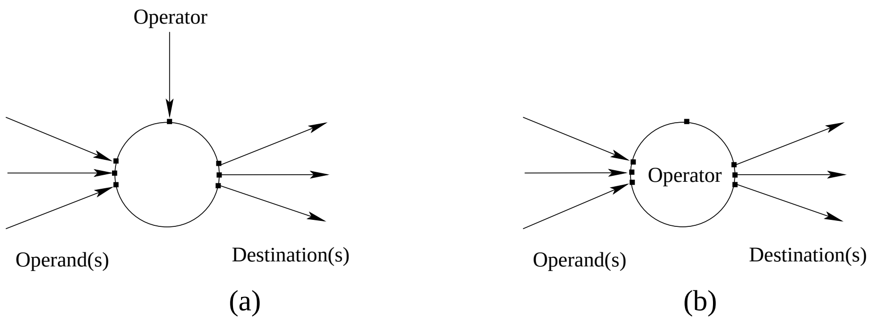


Figure 4.1: (a) shows a node with a dynamic operator, (b) a node with a static operator.

Definition 4.4 *Condensed Node (computational triple).* A condensed node, henceforth referred to as a node, provides one or more operand ports, a single operator port and one or more destination ports. ♢

Nodes are typically represented as a circle, as shown in Figure 4.1(a), with operands on the left, destinations on the right and the operator on top. Operators, like operands can flow along arcs to nodes. In most cases, the operator is statically defined and is represented as text in the centre of the node, as shown in Figure 4.1(b).

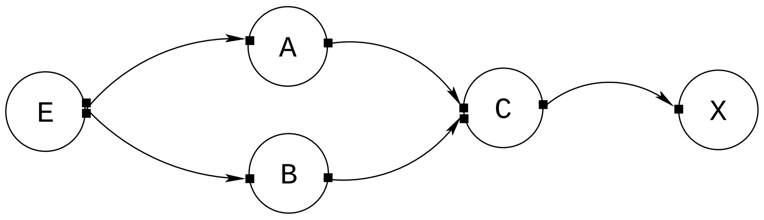


Figure 4.2: A simple Condensed Graph.

Figure 4.2 is an example of a simple condensed graph with an enter node (E) an exit node (X) and three condensed nodes A, B and C. Each node consists of three parts: one or more operands, an operator, and one or more destinations. Thus, nodes are referred to as *computational triples*. A node can only execute when this triple is complete, that is, they have all of their operands, an operator and all of their destinations. Arguments to the graph shown in Figure 4.2 are passed as operand data to both A and B, through the E node.

Definition 4.5 *Firable Node.* A node is considered firable when its triple is complete, that is when it has an operator associated with its operator port, operands associated with each of its operand ports, and its destination ports are bound to a destination node. ♢

When the graph from Figure 4.2 is scheduled for execution, both the A and B nodes are immediately firable, that is, they each have their operator, operands and a destination. These two nodes can execute in parallel. In contrast, the node C cannot execute until it has the results from both A and B. When the nodes A and B execute, the results flow along the arcs to the node C. Once C has its operands, it becomes firable.

4.1.1 Stemming and Grafting: a basis for lazy and eager evaluation

Stemming and grafting are used in condensed graphs to alter the execution sequence of nodes in a graph. Stemming has the effect of temporarily delaying the execution of parts of the graph. This provides for lazy evaluation of the graph.

Definition 4.6 *Stemming.* A node's operand is stemmed when the operand node's destination is not attached to the operand port of the node. ♢

Definition 4.7 *Grafting.* Grafting is the process of attaching a stemmed operand to the the operand port of the destination node[1]. ♢

[1]Grafting can also be used to graft operators to operator ports.

A stemmed node cannot be executed until it has been grafted. This allows control over the execution of parts of the computation. Nodes that are stemmed will not be grafted until their results are required by the computation[2]. This provides for lazy evaluation. In contrast, when all nodes in a graph are grafted, all nodes that are fireable can be executed in parallel. This provides for eager evaluation. There are advantages to both the lazy and eager models of computation. These will be explored in Examples 4.1 and 4.3.

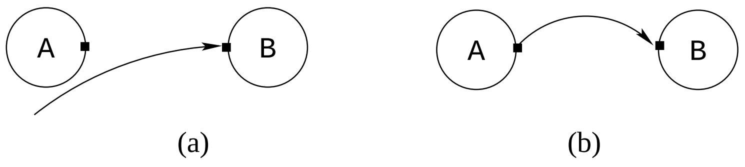


Figure 4.3: (a) A is a stemmed operand to B, (b) A is grafted to B

In the graphical representation of condensed graphs, we represent stemmed nodes as "sitting" on the arc, as shown by the node A in Figure 4.3(a). In contrast, the node A in Figure 4.3(b) is grafted to B. Constants are represented graphically as rectangular boxes enclosing the constant value. They are traditionally stemmed until the node is ready to be fired, whereupon they are automatically grafted. Ports are represented as small boxes on nodes where arcs may attach[3].

Stemming a node will delay its execution as stemmed nodes are only grafted when their results are required. Stemming is often used when there are multiple conditional branches within the computation. The branches are grafted after the conditional case is computed. Stemming all potential branches means that only one of these branches will ever be executed. If these branches were instead grafted, then all branches could be executed in parallel. This behaviour has both advantages and disadvantages. Potentially nodes in all possible branches could be executed before the correct branch is chosen and redundant computations discarded. Executing conditional branches of a computation simultaneously provides speculation, or eager evaluation. Eager evaluation of multiple branches can potentially provide much more parallelism to a computation, with the disadvantage of redundant computations. The condensed graphs execution engine has the ability to increase or decrease the amount of speculation depending on the amount of parallelism desired. This will be examined in more detail in Section 4.2.

A special case exists for nodes where one or more of their destinations are stemmed while their operator and operand(s) are present. Such nodes are considered *reducible*, that is, they will be fireable when their destinations are present. This reducible state is important for the execution scheduler. When a graph is executing, the scheduler will first look for fireable nodes. When no fireable nodes are present it will then examine the graph for reducible nodes. Reduceable nodes will then have their destinations grafted and thus become fireable.

[2]Although this rule is sometimes overridden to allow speculative computations.

[3]Note that where no ambiguity can arise: ports are not specifically depicted on subsequent graphs.

Definition 4.8 *Reducible Node.* A node is considered reducible if there is a condensed graph associated with its operator and with each of its operands. A node with stemmed operands is not reducible until all of the operands and the operator are grafted. ◊

Example 4.1 We define a simple graph, shown in Figure 4.4 to compute the factorial of an integer. An integer value is passed as a parameter to the graph through the E node and in turn is passed as an operand to the equalsto (=), minus (-) and multiply (*) nodes. The partially executed graph of Factorial(50) is shown in Figure 4.5. Fireable nodes are shown in the diagram as shaded.

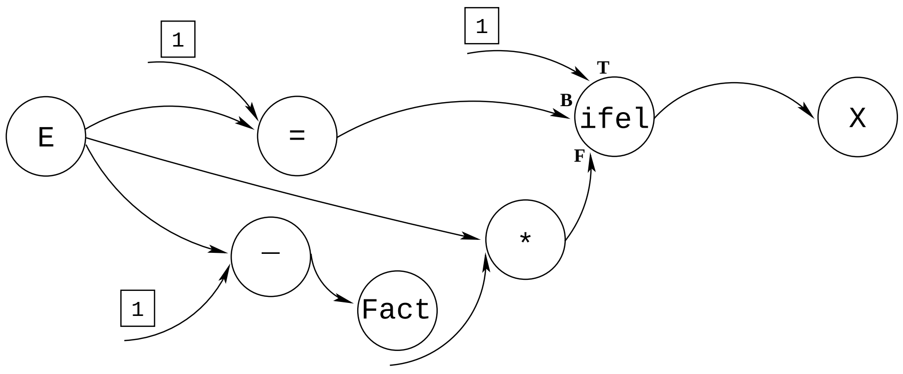


Figure 4.4: A lazy graph implementation of Factorial(n).

Nodes that have fired are not immediately garbage collected by the execution engine. They are simply dereferenced and are properly deallocated when the exit node fires. The ifel (conditional) node has conditional Boolean (**B**), True (**T**) and False (**F**) operand ports. The ifel node becomes fireable once its boolean port becomes bound to an operand value (Figure 4.6), even though its other ports may not be bound. The conditional node is a special case within the condensed graph model. Other nodes are fireable only when their all of their operand ports are bound. In contrast, the ifel node is fireable once the boolean operand port is bound[4]. Unexecuted nodes may be present on any of the other operand ports. Once the conditional node becomes fireable, the node (or nodes) present at the 'winning' operand port pass through the conditional node and attach to the operand port(s) of the conditional's destination(s).

If the operand to the graph was equal to 1, then the computation would continue using the true branch and the false branch would be ignored. Given that we are using the value 50 as the input to this graph, the computation proceeds along the *false* branch. Thus the nodes on the false branch now pass through the conditional node, and the * node is now attached to the X node's operand port, as shown in Figure 4.7.

The ifel node is now dereferenced and the * node is instead attached to the X node. At this

[4]Actually, this is not a special case. The Condensed Graph Model uses the concept of port *strictness*. This is discussed in detail in [122]. The ifel node is the only node with this behaviour that is discussed in this thesis.

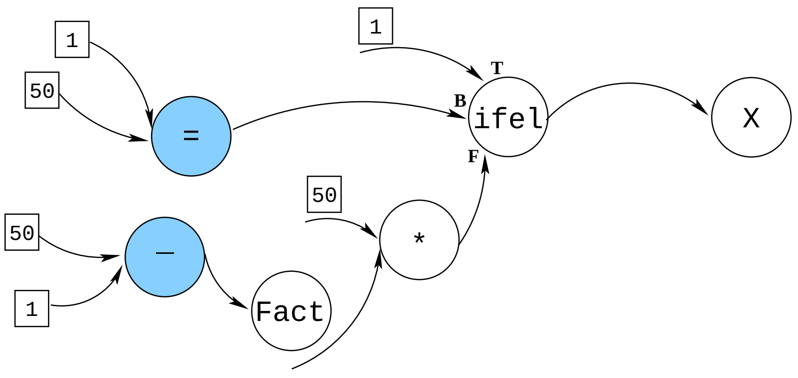


Figure 4.5: The partially executed graph of Factorial(n) after one step.

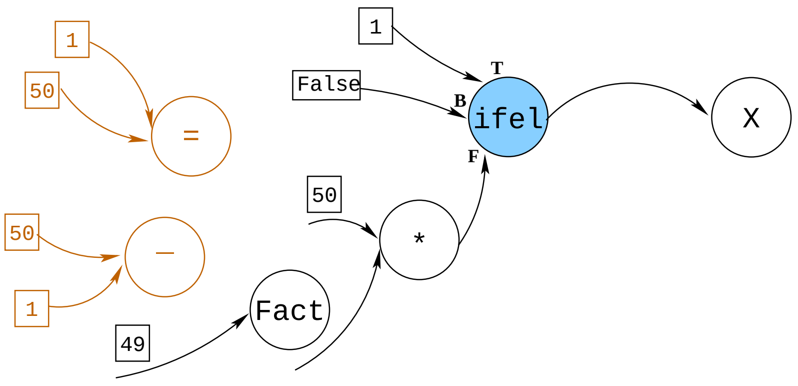


Figure 4.6: The partially executed graph of Factorial(n) after two steps.

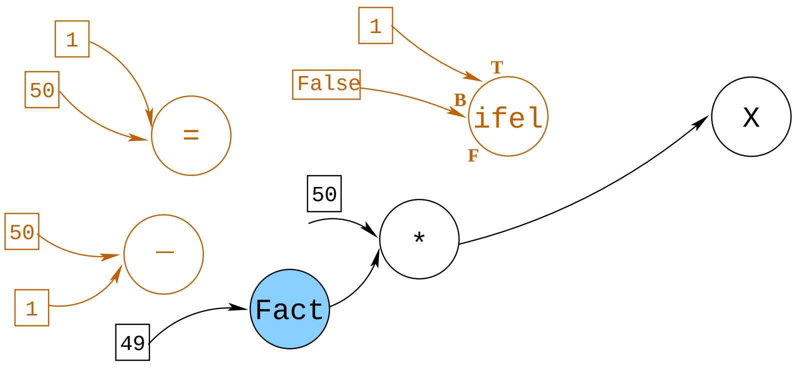


Figure 4.7: The partially executed graph of Factorial(n) after the execution of the `ifel` node.

point there are no fireable nodes, but as the `Fact` node is reducible, it is grafted and becomes fireable. Another instance of the graph is now spawned, using the operand 49. When the result of this graph (and all potential sub-graphs) are computed, the `multiply` node is executed returning a result to the `X` node, giving the factorial of 50. △

4.1.2 Condensation and Evaporation: embedding subgraphs

A node in the condensed graphs model can represent either an atomic action or a complete condensed graph. Atomic actions are primitives of the execution scheduler or operations provided by the underlying host. When a node representing a condensed graph is executed, that node is replaced by the graph it represents. The graph is then executed. Condensed graphs are hierarchical, that is, graphs can contain subgraphs. Subgraphs can represent reusable or recursive sequences within a condensed graph application. In the condensed graph model, subgraphs are represented as nodes in the parent graph. When these nodes execute they expand to reveal the subgraph. The subgraph is then executed. This process is known as *evaporation*.

Example 4.2 Figure 4.8 shows an instance of the factorial graph where the factorial node in the original graph shown in Figure 4.4 has been evaporated. In this case, two iterations of the graph are

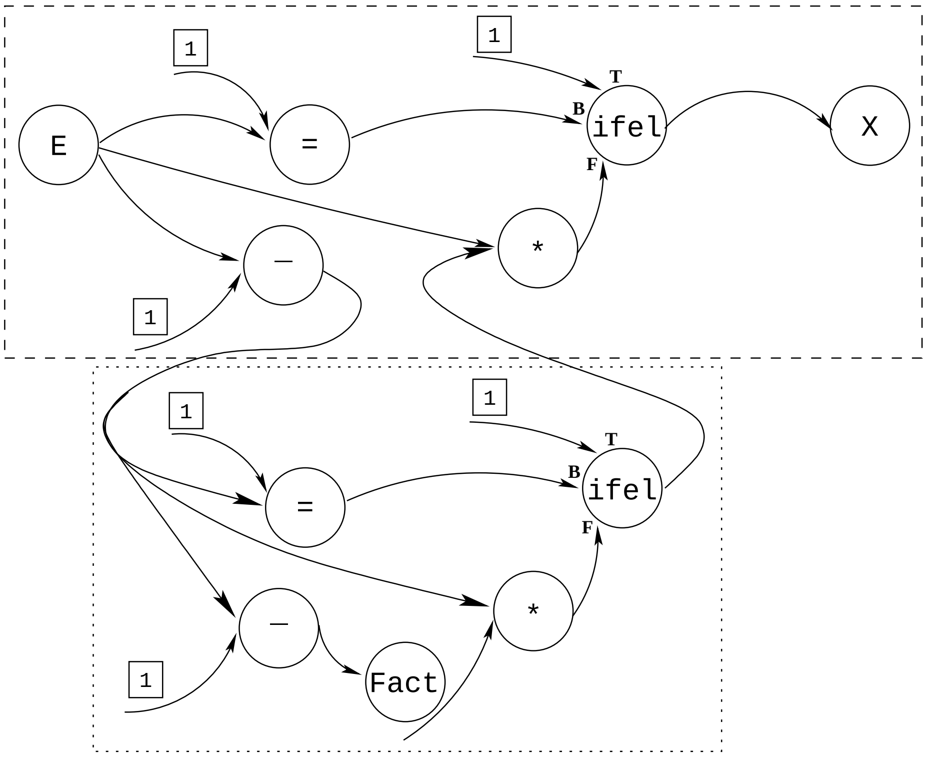


Figure 4.8: Evaporation of the recursive `Fact` node within the Factorial Graph

visible, the original shown in the dashed box and the evaporated graph shown in the dotted box. In practice, evaporation causes a separate condensed graph to be spawned, with the operands passed through the E node and the result returned, via the X node, to the next node, in this case the * node, in the parent graph. △

The process to convert complex graphs to simpler subgraphs is known as condensation. This process is not automated: condensing a graph is currently a task performed by programmers.

4.1.3 Unifying eager, lazy and imperative computations

The original aim of the condensed graphs model was to unify the eager, lazy and imperative sequencing of computations. As previously defined, a node in the model can be considered a computational triple of operands, an operator and destinations. A node is not firable until this triple is complete. In the condensed graphs model, each of these approaches rely on the restriction of one part of the triple in order to control executions. With the eager approach, computations are sequenced based on the availability of operand data. In the lazy approach, computations are sequenced based on the availability of destinations. The imperative approach, in contrast, relies on the availability of operators to sequence the computation. It is possible to mix the three models of computation within a single graph, depending on how the graph is written.

Example 4.3 The original Factorial graph (Figure 4.4) is a lazy condensed graph. It is called lazy, as nodes are only executed when the result of a node is required. In contrast, an eager version of this graph is shown in Figure 4.9. In this graph the recursive `Fact` node is grafted, allowing it to recurse

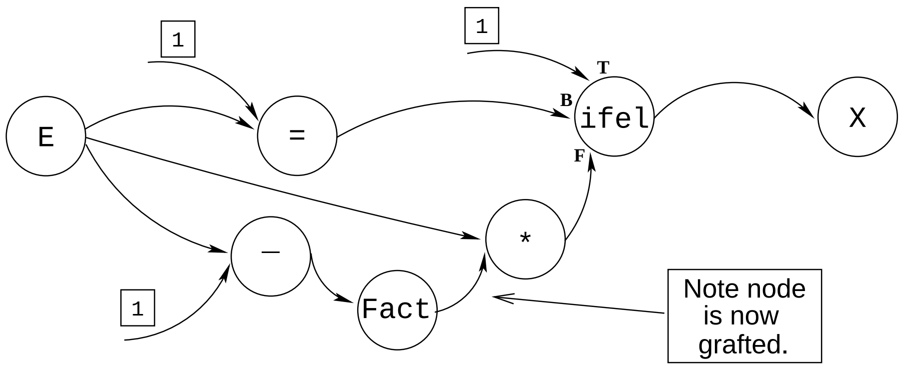


Figure 4.9: An eager version of Factorial(n)

before the boolean check has been performed. This allows much more parallelism as many more nodes are now fireable, both in the original graph and in its subgraphs. However, this eagerness has consequences. Since the boolean check is not performed prior to the spawning of the subgraph, graphs that recurse infinitely are possible.[5]. △

[5]In theory, a *throttling* mechanism can be used to selectively stem graphs, or throttle back on the infinite recursion in this graph [122]. However, throttling is not currently provided as part of the WebCom system.

4.2 Executing Condensed Graphs

Condensed Graphs are executed using a execution scheduler called the *Triple Manager*. The triple manager schedules nodes within a condensed graph for execution.

4.2.1 Triple Manager

The Triple Manager manages and executes computational triples. It consists of two basic parts, the graph memory and a scheduler, as shown in Figure 4.10. Nodes in a condensed graph are either executed by the host operating system or are executed by the triple manager itself. The triple manager will execute special nodes, called triple manager primitives. Examples of triple manager primitives include the `E`, `X` and `ifel` nodes, and nodes that are condensed subgraphs.

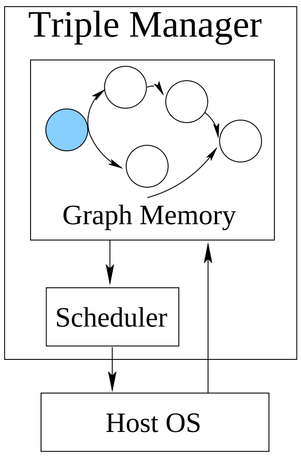


Figure 4.10: The architecture of the Triple Manager

Graph Memory stores the current status of an executing graph. When computational components are executed, their results are integrated into the status of the graph. The scheduler selects fireable nodes to be executed. If no fireable nodes are present, it will select reducible nodes and graft their operands.

4.2.2 Distributing Computations

Triple managers can be linked together to form a distributed computation architecture. There are several implementations of this type of architecture [85, 104, 132, 144–146]. These implementations provide parallel execution of graphs across different machines. In each case, the initial triple manager schedules fireable nodes to different machines. Whenever a condensed node that defines a condensed graph is executed, its defining graph is scheduled to a triple manager. In a distributed implementation, this evaporated graph may be sent to a different triple manager. This results in a hierarchical n-tier architecture where the connections between distributed triple managers will evolve to match the executing graph.

WebCom is the primary architectural platform for distributed condensed graphs. It is a distributed n-tier metacomputer and is examined in detail in Chapter 5. Other examples include the peer-to-peer metacomputer, ComPeer [146] and an implementation on field programmable gate arrays, ARC [85, 128].

Chapter 5

WebCom

WebCom [104, 124, 126, 130–132, 144, 145] is a metacomputer [24, 103] that is designed to execute condensed graphs in a distributed manner. It uses a variant of the client/server paradigm to distribute operations for execution over a network. WebCom provides an n-tier approach to distributed computing. In contrast to a traditional two-tier metacomputer, where there is a single parent and many children, WebCom's n-tier structure allows each child WebCom to act as a parent to other children. WebCom handles the issues associated with distributed computations, such as communication, load balancing, fault management and, as is proposed in this dissertation, security. These features are transparent to the execution of condensed graph applications.

The WebCom architecture has proven itself to be adaptable and extensible. It can be used to support a variety of architectures including middleware [67, 126], web services [148], and the Grid [124]. WebCom is designed to be modular, that is, each of its major components are developed as modules. The implementations of these modules are *"pluggable"* – they can be easily replaced with different implementations of that type of module. For example, there could be a load balancer that provides round-robin scheduling and an alternative that uses a more sophisticated scheduling policy based on feedback from its children. This allows WebCom to be easily extended. The security manager in WebCom has been extended to support a micropayment system [64], a decentralised system administration tool for grids [48, 147] and a security policy tool for enterprises [63]. Each of these extensions are described in detail in Chapter 8.

This chapter will examine the architecture of WebCom, concentrating in particular on the design of the security architecture. The development of WebCom was a collaborative effort within the Centre for Unified Computing in University College Cork. The fault tolerance architecture is described in [104]. In [144], the load balancing architecture is described. The contribution detailed in this dissertation is the design and implementation of WebCom's security architecture.

Providing a security architecture for WebCom entails first identifying the security risks that metacomputers suffer and creating an architecture that addresses these risks. The security system must also consider, and work with, the other systems that WebCom provides, such as fault tolerance and load balancing.

In Section 5.1 we will introduce the WebCom system and identify the type of problems it can be used to solve. The WebCom architecture is examined in Section 5.2. Section 5.3 describes the types of problems WebCom can be used to solve. These include processor intensive parallel computations, such as a distributed key cracking and distributed workflows. One of the major benefits of the WebCom system is the ability to separate functionality from scheduling control. This is described in Section 5.4. We discuss the advantages and limitations in Section 5.5.

5.1 Distributing Computations

WebCom essentially functions as a distributed condensed graph execution engine, known as the Triple Manager that was described in Section 4.2.1. WebCom operates as a virtual machine running on top of the host architecture. Thus, we refer to the system as the WebCom Virtual Machine, or *WVM*. When a WVM is executing a graph, it can schedule nodes within that graph to its children. The child WVMs then execute the nodes and return the results to their parent. The results are integrated into the graph and the execution continues. When a condensed graph is executing and a subgraph is uncovered, this subgraph can be sent to another WVM for scheduling. This subgraph is then maintained by a second WVM. This is known as *promotion* [130]. The promoted WVM will schedule the nodes contained within the subgraph to its children As an application executes, there can be many subgraphs uncovered within the parent graph. Subgraphs can themselves contain further subgraphs. As these graph are discovered, the network structure of the WVMs will ideally change based on the discovery of subgraphs[1]. Thus, as a computation evolves, the network architecture of WVMs executing the application dynamically evolves to account for the needs of the graph [104]. This type of evolving network is known as an n-tier structure. A representation of WebCom's n-tier structure is shown in Figure 5.1.

WebCom also supports child migration, where child WVMs are adopted from other WVM parents. A WVM can either make this request to its parent, or can make the request to a central child repository, such as Cyclone [133]. Volunteers first register with Cyclone, and are then migrated to WVMs that need workers. Other repositories include Grid machines [145], that is, machines that are provided as workers within a Grid architecture.

WebCom has been used to manage the execution of applications ranging from simple workflows to complex Grid applications. Recall that nodes in condensed graphs can represent atomic actions of any level of complexity, from simple computational primitives to complete application components. A WVM makes no differentiation (other than load balancing) between simple and complex nodes. Both are handled transparently by the scheduler.

[1]Provided that sufficient WVMs are available.

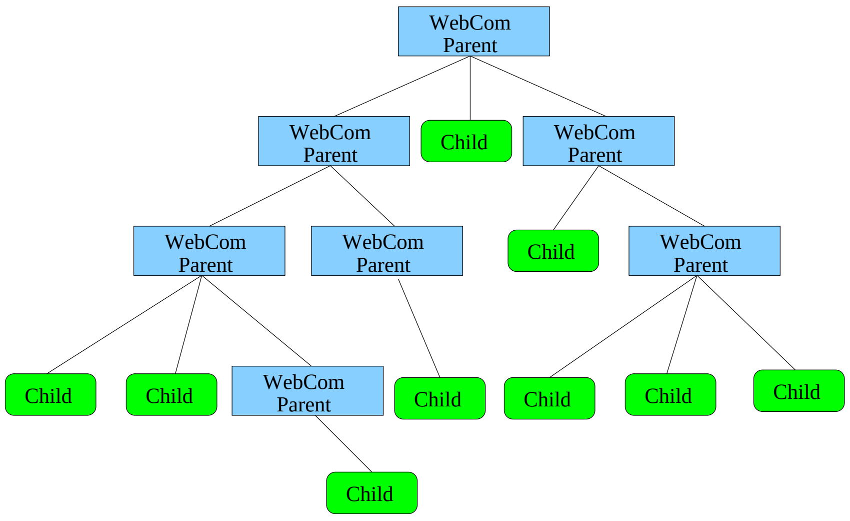


Figure 5.1: WebCom's n-tier architecture.

5.2 Architecture

WebCom is composed of a number of replaceable modules connecting to a central scheduler. The core WebCom modules are the *Execution Engine Module*, the *Communications Manager Module*, the *Load Balancing Module* the *Fault Tolerance Module*, the *Naming Manager Module* and the *Security Manager Module*. These modules make up the core of the WebCom system. Each module has its own local policies that define the configuration of that module. Non-core modules, called user modules, are also supported. The architecture is "pluggable", in the sense that a module can be re-implemented to replace the reference implementations of that module. WebCom supports the use of multiple instances of a core module type concurrently. When multiple instances are present, decisions are made jointly. For example, if multiple security manager modules are present, each module makes decisions on requested actions and only when all modules agree will any action take place. Figure 5.2 displays a representation of the core WebCom modules. These modules are examined in more detail below.

WebCom provides a complete messaging infrastructure, allowing modules to communicate between themselves on both the same WVM, and on other WVMs in the network. For example, a load balancing module can ask another load balancing module in a different WVM the extent of the load on its children. The messaging infrastructure is regulated by the security manager.

When a node is firable by the execution engine and is to be scheduled by the WVM, the scheduler asks the load balancing module and the security manager module to find a child that is both unloaded and is authorised to execute such a node. Whether a child is unloaded and authorised is determined by the load balancer and security manager modules respectively.

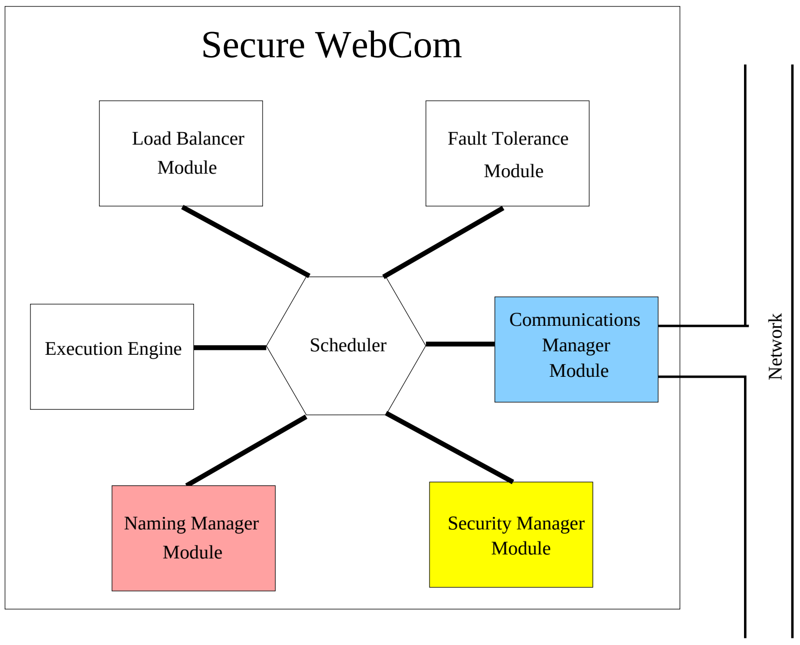


Figure 5.2: Secure WebCom Architecture

5.2.1 Execution Engine Module

The execution engine module is used to execute operations sent to the WVM. Execution engine implementations can be written to execute any type of operation. The reference implementation is a Triple Manager that is used to execute condensed graphs. Implementations also exist to execute *CORBA*, *J2EE*, *.NET* and *Grid* nodes. If a WVM is operating as a child, the execution engine will execute the nodes sent to it locally and will return results to its parent. If the WVM is operating as a parent, the execution engine selects nodes for scheduling and maintains the current state of the executing graph.

5.2.2 Communications Manager Module

The communication manager module is used to manage communication between WVMs. Typically this network communication uses the TCP/IP protocol. However other types of communication manager module also exist, such as the Web Services communication manager module [129], that provides a web services interface to WVM. The communication manager module is used to route the traffic between a WVM, its children and parent.

The current prototype of Secure WebCom uses SSL [92] to provide secure and authentic communication links. This will be discussed in more detail in Chapter 7. The *secure* communications manager manages the cryptographic keys and store trusted keys. Implementing a secure communication manager module entails replacing the standard communication manager module with an implementation that uses the SSL protocol.

5.2.3 Load Balancing Module

The load balancing module manages the load on the children of a WVM. This allows the parent WVM to choose to schedule nodes to children that are relatively unloaded. The load balancing module makes the scheduling decisions in conjunction with the security manager. The load balancing policies range from simple round-robin schedulers to complex schedulers using history to predict future loads [144].

The load balancer is also used to match nodes to specific WVMs. For example, a node may require data that is only available to a specific WVM. The load balancer policy can contain such details and match nodes to specific WVMs[2].

5.2.4 Fault Tolerance Module

The fault tolerance module managers faults that occur within a network of WVMs. This can range from fault avoidance to fault recovery [104]. Work that was scheduled to WVMs that have since failed will be rescheduled to other WVMs by the fault tolerance mechanism.

The default fault tolerance mechanism is sophisticated. If a WVM that is acting as a parent fails, then the only work lost is the work that is taking place on the failing machine. Its children will decide on a new parent between themselves. This new parent will contact their “grandparent”, that is, the WVM that was the parent of the failing machine. The work that the children have executed up to that time will be reintegrated by the new parent and the execution will continue as before.

5.2.5 Naming Manager Module

The naming manager module manages the names of node objects in a condensed graph. When WebCom modules make decisions they require a fully qualified name for nodes. The naming manager generates, stores and updates the names as the execution progresses. These names hold the execution context of the nodes in a condensed graph application. This execution context can be used by the other modules to make scheduling and security decisions about the nodes.

Chapter 6 describes how nodes in condensed graphs can be properly named. The naming manager uses these techniques to manage descriptive names for nodes in the WebCom system. In the current prototype the security manager alone uses these names. However, it is envisioned that the other modules will take advantage of the naming architecture. For example, it is proposed to use the naming system to properly identify WVMs for use by the fault management mechanism.

[2]The security manager module is also capable of ensuring that nodes are sent to specific WVMs. This will be examined in Chapter 7.

5.2.6 Security Manager Module

The security manager module enforces the local security policy of WebCom. The security manager module acts as a reference monitor [25], determining whether it is safe to execute security critical actions. These actions could be the scheduling of a node, the sending of a message, a result to be processed or the local execution of a node. The security manager makes decisions based on its local security policy.

The security manager can be implemented in a number of ways, ranging from a simple manager that permits everything, to a sophisticated access control system. An early prototype implemented a trust management based access control system. In this implementation permissions are delegated to principals, representing WVMs, using cryptographic credentials. The trust management security manager uses the credentials provided by child WVMs to decide where to schedule nodes. The implementation of the Trust Management security module is discussed in more detail in Chapter 7.

5.2.7 User Modules

WebCom supports the development a variety of non-core module types. These are referred to as user modules. One example is the information gathering module [123] that gathers machine execution information from WVMs. This is used to map the load on a cluster of WVMs to measure both performance and usage. User modules cannot directly effect the scheduling decision making logic of WebCom. User modules are restricted to gathering information. This information could be used by a core module, and thus effect a scheduling decision.

WebCom provides a complete application programming interface (API). Using this API, third parties can create different implementations of core WebCom modules, or create entirely new modules.

5.3 WebCom Applications

WebCom allows us to execute applications across many machines in a network. It is used to solve both complex scientific computations, such as identifying astronomical phenomenon [141] and simple workflow applications [126]. Workflow applications are sequences of operations that must execute in the order specified for the application to be successful. For example, Figure 5.3 shows a simple purchase ordering system specified as a workflow. In this workflow, an order must be first proposed (`prop`) and then verified (`ver`) to be considered a valid order.

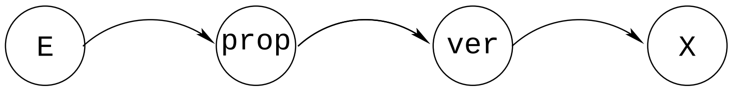


Figure 5.3: A Simple Purchase Ordering Application

Specifying this application as a condensed graph has the implicit benefit that the `prop` node will be executed before the `ver` node. WebCom provides the ability to distribute such application components to different machines in a network. WebCom applications are primarily created using a graphical integrated development environment, the WebCom IDE.

WebCom applications use the benefits of the architecture of WebCom transparently. Consider the following scenario: the node `prop` has been scheduled to a WVM running on machine `A`. This machine breaks down before the node finishes executing. As WebCom has a fault tolerance (in this case a fault recovery) mechanism the node is rescheduled for execution without user interaction. This mechanism is outside of the condensed graph model.

5.4 Separation of Concerns

WebCom provides the inherent ability to separate computation functionality from computation control. Computation functionality is provided by the implementation of the nodes and graphs, the control is provided by the WebCom modules. In most distributed computation architectures, such as PVM [9] and MPI [7], control is embedded within the functional code. For example, the number, type and order of resources to be used to execute the computation are known a priori. This provides a tight coupling between functionality and control. In contrast to this type of architecture, WebCom provides for a separation of concerns at the code level, allowing a loose coupling between functionality and control: functional code is separate from control code.

Example 5.1 Traditionally when Trust Management is used within an application, the calls to the trust management system are embedded into the functional code. For example, Figure 5.3 shows a condensed graph with two nodes `prop` and `ver`. If the implementation of the `prop` node used the KeyNote trust management system (as described in Section 2.3.2), the calls to KeyNote would be embedded into the code as shown below[3].

```
// Initilise JKeyNote Objects
KeyNoteFactory knf = new KeyNoteFactory();
KeyNoteParser trustedParser = knf.getParser(true);
KeyNoteParser untrusted = knf.getParser(false);
KeyNoteNavigator nav = knf.getNavigator();

// Set up a list of variables to act as query.
VariablesList vL = new BasicVariablesList();
vL.addStringVar("App_Domain", "OrderApp");
vL.addStringVar("Operation", "prop");
knf.addVariablesList(vL);

// setup the compliance values and load policy
```

[3] In the code fragments shown in this chapter we use the Java implementation of the KeyNote Trust Management system, JKeyNote [90].

```
       knf.setComplianceValues("untrusted,trusted");
       String pol = ...
       trustedParser.parse(pol);

       // Parse the user credentials
       String usercreds = ...
20     untrusted.parse(usercreds);

       // Perform the query, using the client's key.
       PublicKey ClientKey = ...
       int res1 = nav.findAuthorizer(ClientKey);
       if (res1 > 0)
       {
         prop();
       }
       else
30     {
         // deny access...
       }
```

In this code, before the `prop()` function call is made, the system calls KeyNote and verifies that the call is authorised. It must first initialise the JKeyNote objects and define the query (lines 1–14). Next the local policy (trusted) credentials and the user (untrusted) credentials are loaded (lines 15–20). Finally KeyNote is queried whether the client's public key is authorised for the given query (lines 23–25). In this case, the client must hold credentials authorising the actions `App_Domain == "OrderApp"` and `Operation == "prop"`, such as the credential shown in Figure 5.4.

```
Authorizer: "KAlice"
licensees:  "KBob"
Conditions: App_Domain=="OrderApp"
            && Operation=="prop";
...
```

Figure 5.4: A skeleton credential authorising KBob to perform the actions `OrderApp` and `prop`.

△

WebCom offers a different approach to scheduling of nodes. Nodes that represent atomic components are scheduled based on the characteristics of those nodes. For example, the WebCom security manager decides whether a nodes are permitted to execute on specific resources. If a node is not so authorised, then it will not be scheduled to that resource. Furthermore, if a resource receives a node that its security model does not permit, then that node will be rejected. In the WebCom system, the node application code has no explicit calls to the WebCom security model. The security policy is thus independent of the application code. This allows, for example, programmers who have little experience of the security model to create nodes and graphs[4]. The security policy is created later, before the application is executed in a secure environment, thus providing a loosely

[4]The security model is examined in detail in Chapter 7.

coupled architecture. All WebCom's modules share this characteristic and are, as such, all loosely coupled. Typically, security policies are created during application development, as developers will have greater insight into the security requirements of their application. However, a loosely coupled architecture allows policies to be easily changed without modifying the application code.

Example 5.2 When a node is to be scheduled by WebCom, the scheduler asks the Load Balancing Module and the Security Manager Module to find a suitable child WVM to execute the node. The Load Balancer and Security Manager search for a child based on their decision logic. If the Security Manager is using the KeyNote trust management system as its decision logic, the search for a suitable child WVM takes the form of a check of each candidate until an authorised child is found. This search is performed by a `check` method, parts of which is shown below.

```
   // Node name stored in instrname
   // Enviroment Variables
   vL.addStringVar("App_Domain", "WebCom");
   vL.addStringVar("Domain", instrname.getDomain());
   vL.addStringVar("Graph", instrname.getGraph());
   vL.addStringVar("Function", instrname.getFunction());
   // Multiple Inputs and Destinations
   Vector inputs = instrname.getInputs(); // Go through inputs
   for (Iterator iter = inputs.iterator(); iter.hasNext(); )
       vL.addStringVar("Input", (String) iter.next());

   Vector dests = instrname.getDestinations(); // Go through destinations
   for (Iterator diter = dests.iterator(); diter.hasNext(); )
     vL.addStringVar("Destination", (String) diter.next());

   knf.addVariablesList(vL);
   knf.setComplianceValues("untrusted,trusted");

   // Initilise the credential search engine and load credentials
   KeyNoteNavigator nav = knf.getNavigator();
   try {
     for (Iterator polsiter = pols.iterator(); polsiter.hasNext(); ) {
       String pol = (String) polsiter.next(); // Policy (trusted) Credentials
       trustedParser.parse(pol);
     }
     for (Iterator iter = creds.iterator(); iter.hasNext(); ) {
       String cred = (String) iter.next(); // User (untrusted) Credentials
       untrusted.parse(cred);
     }

     // Check if the client key supplied is authorised.
     int res1 = nav.findAuthorizer(ClientKey);
     if (res1 > 0) {
       return true; // it is, let the SecurityManager know.
     }
     else {
       return false; // not authorised, inform the SecurityManager
     }
   }
 }
```

In this code fragment, jKeyNote is initialised in the same way as before, and acts in a similar manner to the embedded version discussed earlier. However, the implementation used by the Security Manager is generic, that is, it can be used for any node. The environment variables used for the query are extracted from the name of the node[5] (lines 4–15). The properties of the nodes are stored in a name variable, `instrname` in this code fragment, and are extracted for use by the KeyNote query. The KeyNote credentials used will reflect these environment variables. A sample credential is shown in Figure 5.5.

```
Authorizer: "KAlice"
licensees:  "KBob"
Conditions: App_Domain=="WebCom"
            && Domain=="bob.ucc.ie"
            && Graph=="PurchaseOrder"
            && Function=="prop";
...
```

Figure 5.5: A credential allowing KBob to execute a `prop` node with any input(s) and any destination(s) in the domain `bob.ucc.ie`.

△

WebCom's pluggable design allows different implementations of security managers for specific types of security policy. For example, it is easy to imagine that a separation of duties security policy [135] would be required for the purchase ordering system shown in Figure 5.3. Such a policy could define that one person should perform the `prop` operation and another the `ver` operation. In this simple application, the security policy would reference both the `prop` and `ver` nodes. The simplest policy would define a specific WVM that should execute `prop` nodes and a different WVM that should execute `ver` nodes. A specific implementation of the security manager module could be used to enforce this policy.

5.5 Discussion and Conclusions

The WebCom architecture is a metacomputing environment that provides the basis for secure, fault tolerant, load balanced distributed applications. The pluggable nature of the WebCom architecture allows the development of modular components. The reference implementations of the core modules can be replaced, allowing different implementations of those modules to be used. WebCom can therefore adapt to different execution conditions as required.

From a security standpoint, the architecture of WebCom prompts some unique challenges. Ensuring the integrity of the computations executing in a distributed environment entails controlling the configuration of both the computation and the network of WVMs that the computation will be

[5]The process by which the node's name is determined is discussed in Chapter 6 and is not important at this point.

executing on. The security requirements of WebCom are managed by the Security Manager Module. Different implementations of the security manager can be used to enforce different types of access control. As WebCom is a distributed environment, enforcement of the security policy must also be distributed. WebCom can exist outside of the control of a single administrator. Instances of WebCom running on different resources can have different administrators. The security system must support this type of architecture.

Another challenge that faces the security architecture of WebCom includes the localisation of security policy. Each WVM will have its own local policy, and so will only execute computations that comply with these policies. WebCom must support this type of policy localisation and manage the problems that this creates.

WebCom provides built-in support for the separation of functional and control code. Condensed nodes implement functional code; WebCom's modules provide control over this code. Functional and control code requirements are typically disjoint. When a component is modified, that does not necessitate modifying the control code, and vice versa. This provides a loosely coupled architecture. Conventionally, trust management does not support this separation of concerns: security code is embedded within the functional code. However, separating the functional and control code does not provide the same level of detail when making a trust management check. When this check is embedded within the functional code, all of the information available to the application is potentially available to the trust management system. Removing the trust management check from the application requires the ability to extract that same level of detail from WebCom. We argue that a rich naming system for WebCom is necessary so that we can carry out the same kind of check outside of the application code. Chapter 6 will address this issue.

In this chapter, we examined the architecture of the WebCom metacomputer. In Chapter 7, we will examine WebCom's security model in detail and addresses the threats that WebCom must defend against. Specific implementations of security managers for WebCom are also described in Chapter 7, with the types of security policies that these implementations can support.

Part III

Security in Distributed Systems

Chapter 6

Naming for Condensed Graphs

Distributed computing technologies, such as Grid [71, 72] and cluster computing [167], raise some unique problems when articulating security policies. The complexity of these problems depends greatly on what type—open or closed—of distributed system is in use. Closed distributed systems are those where the entire system is owned and/or operated by a single organisation, such as an organisation's cluster or enterprise application. In contrast, open distributed systems consist of shared resources across many administrative domains, such as a computational grid.

Distributed applications are made up of computational components that are executed on distributed resources. With traditional closed systems, computations are performed within domains where the characteristics of the system, such as the resources available or the types of operating system in use, are a priori known. However, in open distributed architectures, computational components may be executed across widely distributed domains, where developers may have less a priori information about the resources the components use and where these components will execute. In a closed distributed system, these resources are known to the developer and the security requirements are more easily configured.

In practice, open decentralised security architectures, such as Trust Management [29, 32, 56], provide an approach towards addressing this concern. Security policies are maintained by the stakeholders in the computations: the users who initiate the computations and the computational resources that host the jobs.

Creating security policies for open distributed computations is a challenging prospect. In this dissertation, different applications have a large range of security goals. In particular, we are primarily interested in access control. Providing a means to create such access control policies implies having the ability to refer to components throughout the computation in a consistent and potentially unique way. For example, how does one specify that sensitive parts of a computation are only sent to be executed on appropriately trusted resources? More complex problems can also be imagined, such as creating a history-based policy [40, 135, 164, 180]. Such policies require representing the contexts that the computation has passed through to this point, for example, a separation of duty rule on a financial transaction could entail ensuring that a different principal approve the transaction that another principal initiates. The context must maintain the information necessary to uphold such policies. We argue that this fundamental problem can be reduced to a *naming* problem. The central premise of our argument is *"if you can name it, then you can make authorisation decisions about it"*. If every component (or, in the case of a condensed graph application, a node) in the computation is properly named, then it can be referred to with as much precision as is required.

Naming distributed components is not a new problem, for example CORBA [81], the Spring naming system [149], the X.500 naming architecture [182] and Enterprise Java Beans (EJB) [169] each provide solutions towards the naming of distributed components. However, each of these solutions addresses naming as a static problem. Distributed objects in these systems have a priori defined names as they do not change often. In contrast, nodes in a condensed graph evolve continually during execution, therefore, the names of these nodes must also evolve. A naming scheme for condensed graphs must consider this evolving nature of components in the computations.

In Chapter 3, we examined the background of distributing computations and described a number of the technologies used to address some of the security issues that arise. Chapter 5 described the distributed architecture used to execute condensed graphs. In particular, Chapter 5 outlined the security architecture of WebCom. This chapter will further examine the structure of condensed graphs in order to properly develop a naming architecture for them. Section 6.1 examines the contextual information required to name a computational component. These requirements are applied to condensed graphs in Section 6.2. In Section 6.3, we propose a rigorous model for condensed graph names. This naming model provides the ability to define name reduction rules that provide a translation from complex and unwieldy names to a more usable reduced form. These reduction rules are described in Section 6.4. Section 6.5 describes some examples of history-based naming policies. A practical implementation of this naming architecture for the WebCom system is then described in Section 6.6.

6.1 Context

A distributed application consists of a number of computational components. Naming such components entails capturing the attributes of the computation at a specific moment in time. One of the applications of these names is in making access control decisions regarding the ongoing computation; for example, whether or not it should be allowed access to certain resources. This is not the only application: names can be used to help define many types of policy, ranging from load balancing to fault tolerance. To properly name a component in a computation, we must first identify the computational context that incorporates the component. This context can be broken down into a number of attributes:

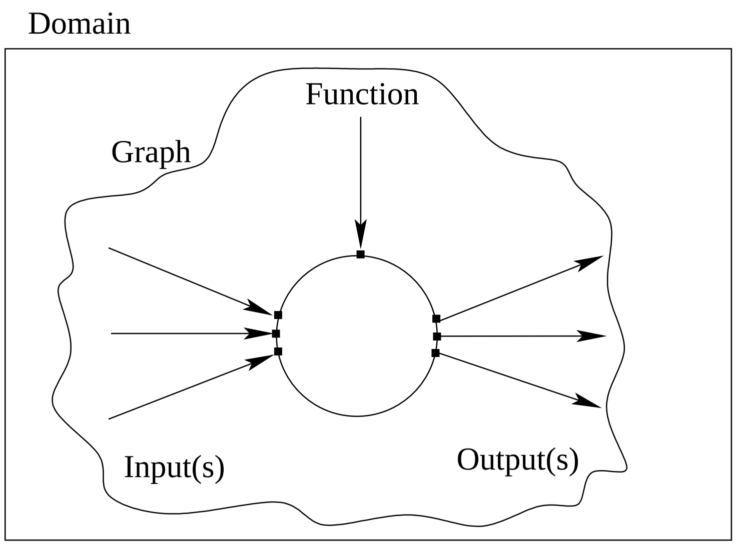


Figure 6.1: The Components of a Distributed Name

- *Domain*: the context in which this component is executing, or in which context it will be executed.
- *Application*: the application (graph) that this component a part of.
- *Function*: the operational function of this component.
- *Inputs*: the computational context(s) that have led to this value.
- *Outputs*: the context(s) that this computation is destined for in the future.

SDSI/SPKI [152] proposes linked local namespaces as a way to build complex names. For example, Alice has a computer that she simply calls *"Computer"*. Bob also has a computer, that he too refers to as *"Computer"*. As Alice and Bob are separate principals, this is perfectly acceptable. However, how does Bob refer to Alice's Computer? Suppose Bob knows Alice simply as *"Alice"*; Bob, therefore, refers to Alice's Computer as *"Alice's Computer"*. More precisely: it is the object that Alice refers to as *''Computer"*. SDSI uses s-expressions to encode these names. In an s-expression, the `ref:` keyword can be regarded as a formalisation of the 's relationship between related components of the local name. Thus, the s-expression representation of *"Alice's Computer"* is `(ref: Alice Computer)`.

Each principal in a system names objects according to their local view of the system. This same rational can be applied to the nodes of a condensed graph. Through the use of s-expressions, local naming provides the ability to use names from other principal's namespaces.

6.2 Naming Condensed Graphs

WebCom applications are specified in terms of condensed graphs. In order that the context of a condensed graph application is properly named, components (nodes) in condensed graphs are named in terms of *WebCom name*s. A WebCom name is a five-tuple, made up of a domain, a graph, a function, zero or more inputs and zero or more outputs. Each of these tuples can themselves refer to a WebCom name, or can be empty. The tuples represent different aspects of a node's context, and are defined as follows:

- the `domain` tuple in a WebCom name is the execution context in which the node that the name refers to has executed, is currently executing, or will execute;
- the `graph` tuple in a WebCom name is the condensed graph in which the node referred to is a member;
- the `function` tuple in a WebCom name is a description of the operator of the node;
- `input` tuples in a WebCom name are the names of the operands to the node There can be multiple inputs, each of which can be referred to separately;
- `output` tuples in a WebCom name are the names of the destinations of the node. There can be multiple outputs, each of which can be referred to separately.

```
<webcomname> ::=
  (WebComName
    [(domain <webcomname>)]
    [(graph <webcomname>)]
    [(function <webcomname>)]
    [(inputs
       {(input <webcomname>)}
     )]
    [(outputs
       {(output <webcomname>)}
     )]
  )

<webcomname> ::=
  (WebComName S-Expression)
```

Figure 6.2: The structure of a WebCom name.

Using these attributes, Figure 6.2 defines a WebCom name as a s-expression. All parts of the name are optional. A name can be represented by a combination of any of these fields or by a simple representative s-expression. There can be one or more `input` and/or `output` fields when the `inputs` and/or `outputs` fields, respectively, are present. This structure represents all of the aspects of a condensed node, including a representation of where that node executes.

Example 6.1 The condensed graph shown in Figure 6.3 defines a simple web services application as a workflow of atomic actions, that implements a travel agent application. The application operates as a simple travel agent, using web services from different sites. Users of the application are directed to fill in the details required to purchase an airline ticket (`BuySeat`). Once this purchase is completed, the relevant details are sent to hotel reservation (`RentRoom`) and car rental sites (`RentCar`). The user can then fill in any extra details. Finally all the details are collated and printed out for the user (`Print`).

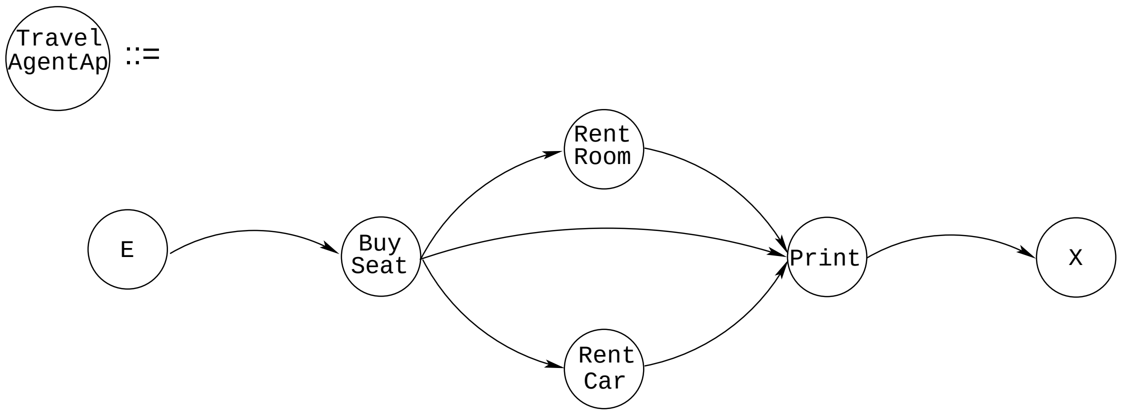


Figure 6.3: A simple Travel Agent Web Services application, specified as a Condensed Graph.

From Alice's perspective, a version of the `RentCar` node from Figure 6.3 executing on her computer, named "`Computer`", can be named using an s-expression as:

```
(WebComName
 (domain Computer)
 (graph (ref: Computer TravelAgentAp))
 (function (ref: Computer (ref: TravelAgentAp RentCar)))
 (inputs (input (ref: Computer (ref: TravelAgentAp (ref: RentCar Input )))))
 (outputs (output (ref: Computer (ref: TravelAgentAp (ref: RentCar Output)))))
)
```

From Bob's perspective, the components of the name must specify the principal Alice, in whose namespace these name-components exist. Bob's name for the `RentCar` node becomes:

```
(WebComName
 (domain (ref: Alice Computer))
 (graph (ref: Alice (ref: Computer TravelAgentAp)))
 (function (ref: Alice (ref: Computer (ref: TravelAgentAp RentCar))))
 (inputs (input (ref: Alice (ref: Computer
                      (ref: TravelAgentAp (ref: RentCar Input ))))))
 (outputs (output (ref: Alice (ref: Computer
                      (ref: TravelAgentAp (ref: RentCar Output))))))
)
```

While the node's name changes based on the perspective of the namer, it is immediately obvious that the same node is being referred to. △

Local naming provides the ability to store the required detail to identify each portion of the node in as much, or as little, detail as is necessary.

Example 6.2 Taking the node `RentCar` from the condensed graph shown in Figure 6.3, we can use information about the node's inputs and outputs to create a more contextual representation of the node in terms of a specific airline (Aerlingus), car rental company (Hertz) and travel agent (eBookers), as shown in Figure 6.4.

```
(WebComName
  (domain (ref: Hertz (ref: Paris)))
  (graph (ref: eBookers (ref: Dublin (ref: Alice TravelAgentAp))))
  (function (ref: Hertz (ref: Paris (ref: TravelAgentAp RentCar))))
  (inputs (input (ref: Aerlingus (ref: EI220 (ref: Paris
                              (ref: TravelAgentAp BuySeat))))))
  (outputs (output (ref: eBookers (ref: Dublin (ref: Alice
                               (ref: TravelAgentAp Print))))))
)
```

Figure 6.4: A possible name for the `BuySeat` Node.

In this example, the `RentCar` node is executed in Hertz's Paris office. The web service application (`TravelAgentAp`) was launched by the eBookers travel agent in their Dublin office, by a principal called Alice. The flight involved is Aerlingus flight EI220 to Paris, and the details will be sent back to eBookers Dublin office for printing.

This node has a single input and a single output. Both the input and output node are referred to concisely using their function tuple. How names are constructed in practice is examined in detail in Section 6.4. △

As some nodes have multiple inputs and/or outputs, for example, the `BuySeat` and `Print` nodes in Figure 6.3, the names of such nodes must have the ability to refer to these multiple inputs/outputs.

Example 6.3 The output result of the execution of the `BuySeat` node acts as the input to three other nodes, `RentRoom`, `Print`, and `RentCar`. The name for the `BuySeat` node could be as shown in Figure 6.5

These names can be used directly in authorisation credentials, such as the SPKI credential shown in Figure 6.6. This credential authorises a user to execute a specific `BuySeat` node, for example, when the output is destined for "*Hilton's Paris's TravelAgentAp's RentRoom*" and so forth.

In all of these cases, the names shown are not the only potential names for these nodes. Node names depend on the amount of information required for a particular application. Credentials can often be simplified, for example, when the input or output constraints are not required and instead

```
(WebComName
 (domain (ref: Aerlingus (ref: Dublin)))
 (graph (ref: eBookers (ref: Dublin (ref: Alice TravelAgentAp))))
 (function (ref: Aerlingus (ref: Dublin (ref: TravelAgentAp BuySeat))))
 (inputs (input (ref: eBookers (ref: Dublin (ref: Alice
                                            (ref: TravelAgentAp E))))))
 (outputs
  (output (ref: Hilton (ref: Paris (ref: TravelAgentAp RentRoom))))
  (output (ref: eBookers (ref: Dublin (ref: Alice
                                      (ref: TravelAgentAp Print)))))
  (output (ref: Hertz (ref: Paris (ref: TravelAgentAp RentCar)))))
)
```

Figure 6.5: An extended name for the `BuySeat` node.

any `BuySeat` nodes from any input and going to any output is allowed. When a detail is not specified, the credential becomes more general. In Section 6.4 we will examine how we can control the amount of information stored in a name.

△

Node names are dynamic in the sense that they can change as the nodes pass through different contexts and may, therefore, grow in size to record the history of transited contexts. While these names provide the contextual detail required to enable naming policies to be articulated before computation takes place, it is clear that the size of these names may cause them to become unusable in computations of a non-trivial nature. A consistent system is required to provide a compact reduced form for these names, yet still containing enough detail to allow informed authorisation decisions to be made. For example, a more compact representation of the `RentCar` node from Example 6.2 might include less information, such as the simple s-expression:

```
(WebComName (ref: Hertz (ref: Paris RentCar)))
```

This represents a node that Alice refers to as "*Hertz's Paris' RentCar*". In Bob's namespace, this would be referred to as "*Alice's Hertz's Paris' RentCar*". How names can be transformed and/or simplified using *reduction rules* is examined in Section 6.4.

6.2.1 Unique names

It may not be immediately possible to uniquely name nodes in a condensed graph. Consider a graph with two nodes that are functionally equivalent, such as the two `B` nodes in Figure 6.7. Both nodes are functionally identical, have the same input node, `A` and the same node, `C`, receives their output. For example, this behaviour may be desirable when replicated component executions are required.

On the one hand it can be argued that this is not an issue for an access control policy. If the nodes are identical, then the security policy should also be identical. Consider, on the other hand, a separation of duties style policy [40, 58, 180] that states: "*Only one node* `B` *may execute in any single*

```
(cert
 (issuer (hash sha1 |dsEFA73213jsDDF3784JDFjfsFsd=|))
 (subject (ref: UCC (ref: CSDEPT (hash sha1 |dasdk...|))))
 (propagate)
 (
  (tag
   (execute
    (WebComName
     (domain (ref: Aerlingus (ref: Dublin)))
     (graph (ref: ebookers (ref: Dublin
                         (ref: Alice TravelAgentAp))))
     (function (ref: Aerlingus (ref: EI220 (ref: Paris
               (ref: TravelAgentAp BuySeat))))
     (inputs
      (input (ref: ebookers (ref: Dublin (ref: Alice
                          (ref: TravelAgentAp E))))))
     (outputs
      (output (ref: Hilton (ref: Paris
              (ref: TravelAgentAp RentRoom)))))
      (output (ref: ebookers (ref: Dublin
              (ref: Alice (ref: TravelAgentAp Print)))))
      (output (ref: Hertz (ref: Paris
                     (ref: TravelAgentAp RentCar))))))
    )
   )
  )
 )
 (not-before "2006-01-01_00:00:00")
 (not-after "2006-12-31_23:59:59")
)
```

Figure 6.6: A SPKI credential authorising a user to execute a `BuySeat` node.

domain". On first examination, this contradicts our previous argument that the security decision for two identical nodes should be identical. However, in practice, this is not the case with dynamic names as such names contain a context. Once a node `B` is assigned to a domain, its name changes. Dynamic names are updated continually as their context changes.

It is possible for two nodes to have the exact same name even after they have been assigned to a domain. If, for example, the two `B` nodes from Figure 6.7 are scheduled to the same domain, then the names of these nodes would be identical. However, we argue that the responsibility lies with the application developer to avoid such situations. One approach would be to use different function names to refer to the nodes `B`.

While it is possible in most cases to generate unique dynamic names for nodes, this is often undesirable. For example, when specifying a security policy, it would be easier to refer to more "generic" names that can refer to many nodes rather than requiring every possible node name to be uniquely identified.

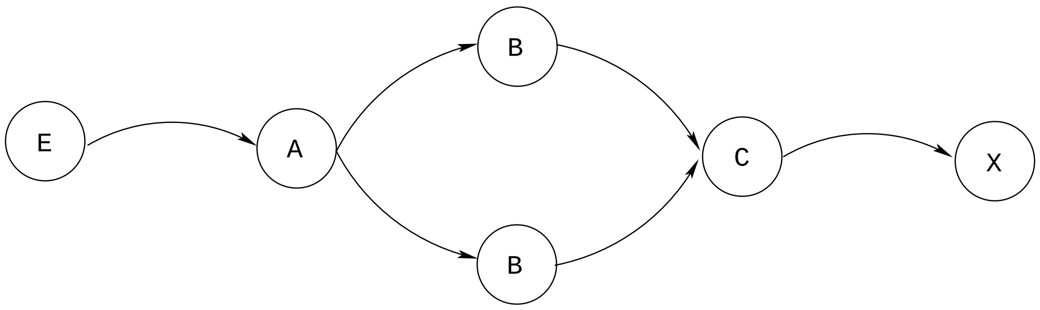


Figure 6.7: A condensed graph with non-unique nodes.

6.2.2 Self Referencing Names

An important aspect of a naming architecture for condensed graphs is how these names are generated. As every node in a condensed graph is potentially (indirectly) connected to every other node, then a single node's name may potentially reference every other node in the graph. In the examples of node names provided up to this point in this dissertation, we have referred to other nodes simply by the name of their function tuple. We will see in Section 6.4 that this corresponds to "de facto" name reduction. If instead, we use the full name of the connected nodes, then the node's name becomes infinite.

Example 6.4 In Figure 6.8, the `RentCar` node has been defined in a self referential manner with one round of unfolding, that is, WebCom names of the input nodes are included in the node name. One of the inputs to the `RentCar` node is the `BuySeat` node. Consequently if this node's name is constructed, then one of its outputs is the `RentCar` node.

△

This increase in name complexity is not necessarily disadvantageous. With increased complexity in a name comes increased precision. For example, executing `B` can be distinguished from executing `B` using input that came from `C`. Name complexity can be addressed when generating names for nodes. When a node is being named, reduction rules are applied to that name in order to simplify the name of the node.

6.3 A Naming Model for Condensed Graphs

SDSI-like local names provide an adequate means to refer to nodes in condensed graphs. In the implementation of the naming architecture for WebCom, s-expressions are used to provide the internal representation of node names. For the sake of compact exposition, rather than use s-expressions, we use haskell [98] in this dissertation to describe the naming model. Haskell provides a precise and concise way to specify WebCom names and reduction rules and provides a rigorous notation that can be type-checked and executed to confirm consistency.

```
(WebComName
 (domain (ref: Hertz (ref: Cork)))
 (graph (ref: eBookers (ref: Dublin (ref: Alice TravelAgentAp))))
 (function (ref: Hertz (ref: Cork (ref: TravelAgentAp RentCar))))
 (inputs
  (input
   (WebComName
     (domain (ref: Aerlingus Cork))
     (graph (ref: eBookers (ref: Dublin (ref: Alice TravelAgentAp))))
     (function (ref: Aerlingus (ref: EI220 (ref: Paris (ref: BuySeat)))))
     (inputs
      (input (ref: eBookers (ref: Dublin (ref: Alice (ref: TravelAgentAp E)))))
     )
     (outputs
      (output (ref: Hilton (ref: Paris (ref: TravelAgentAp RentRoom))))
      (output (ref: Hertz (ref: Paris (ref: TravelAgentAp RentCar) )))
      (output (ref: eBookers (ref: Dublin (ref: Alice
                                    (ref: TravelAgentAp Print))))
      )
     )
   )
 (outputs
  (output (ref: eBookers (ref: Dublin (ref: Alice (ref: TravelAgentAp Print)))))
 )
)
```

Figure 6.8: A recursive definition of the `RentCar` node.

Figure 6.9 gives the haskell definition of a WebCom name. A name is a `Pnam` (*primitive* name), `Snam` (*structured* name) or `Lnam` (*linked* name). There is also a zero-arity constructor function, `Empty`, that represents an empty, or null, name. Primitive names are strings that cannot be reduced any further. Structured names consist of the familiar five-tuple, domain (`dom`), graph (`grph`), function (`fun`), inputs (`ins`) and outputs (`ops`). Each of these tuples can themselves contain complete `Name` objects. As there can be multiple inputs and outputs, these tuples can hold lists of `Name` objects.

The linked name constructor function (`Lnam`) is used to hold representations of SDSI-like local

```
data Name  =
          Empty
       |  Pnam {pnam :: String}
       |  Snam {dom  :: Name,
                grph :: Name,
                fun  :: Name,
                ins  :: ([Name]),
                ops  :: ([Name])}
       |  Lnam {lnam :: [String]}

   deriving (Eq, Show)
```

Figure 6.9: Haskell representation of WebCom names.

names. These local names form lists of linked names, for example:

```
(Lnam ["Hertz","Cork","RentCar"])
```

is equivalent to the s-expression:

```
(ref: Hertz (ref: Cork RentCar))
```

The definition of a linked name could have been given as a list of linked `Name` objects. However, while such a definition allows for a more expressive syntax, using linked `String` objects allows for a clearer presentation, and avoids having lists of strings with embedded `Pnam` constructors. For example, the name of `RentCar` from above would instead be specified as:

```
(Lnam [(Pnam "Hertz"),(Pnam "Cork"),(Pnam "RentCar")])
```

Linked names consist of irreducible primitives and so are represented as a list of strings.

Example 6.5 Figure 6.10 gives a haskell representation of the name of the `CarRental` node from the Condensed Graph shown in Figure 6.3. In this case, `Lnam` (linked names) are used to represent

```
(Snam
   (Lnam ["Hertz","Cork"])
   (Lnam ["eBookers","Dublin","Alice","TravelAgentAp"])
   (Lnam ["Hertz","Cork","TravelAgentAp","RentCar"])
   [(Lnam ["Aerlingus","EI220","Paris","TravelAgentAp","BuySeat"])]
   [(Lnam ["eBookers","Dublin","Alice","TravelAgentAp","Print"])]
)
```

Figure 6.10: Representation of the `RentCar` node

the linked local namespace of each of the components of the name. For example, the domain of the node is Hertz's Cork office. △

Haskell provides three basic derivations for each data type, equality (`Eq`), display (`Show`) and ordering (`Ord`). `Eq` defines how values are tested for equality; `Show` defines how the value is represented on output; `Ord` defines how objects are ordered with respect to each other. For the moment, we will use the default definitions for both `Eq` and `Show` relations, but will define the `Ord` relation separately.

The default definition of the ordering relation (`Ord`) for the `Name` data type does not operate as we require. The standard `Ord` relation generated automatically by Haskell does not give us the level of control over name ordering that we desire. For example, we could not support a naming policy that does not want to consider differing domain tuples as causing two node names to be different. For this reason, we define our own ordering for `Name` in Figure 6.11.

In our ordering for names, the `Empty` name is the lowest name in the ordering; a structured name is less than another when the components of the former are each less than their respective

```
instance Ord Name where
  Empty              <= x                     =  True
  (Snam a b c d e)  <= (Snam v w x y z)  =  (a <= v) && (b <= w) &&
                                              (c <= x) && (d <= y) && (e <= z)
  (Pnam p)           <= (Pnam q)           =  p == q
  (Lnam l)           <= (Lnam m)           =  l <= m
  x                  <= y                  =  x == y
```

Figure 6.11: Definition of the `Ord` relation.

components in the latter; primitive names are either disjoint, or are equal, and so are not ordered; one linked name is less than another when the former is a prefix of the latter. In practice, this relation is redefined based on the requirements of particular applications.

As WebCom names can be self referencing, evaluating this ordering may not terminate when one or more self referencing names are compared. Therefore, care must be taken to avoid this when writing names; one strategy is to design reduction rules that ensure finite names.

6.4 Reduction Rules

The design of a WebCom name provides the flexibility to refer to a node with as much, or as little, contextual detail as needed. The ability to store a lot of contextual detail comes at the cost of possible redundant information stored in the name. This cost can be lessened through the use of *reduction rules*. A reduction rule is a function that converts a complex name for a node into a simpler representation. The writer of these rules must ensure that consistent names are produced.

Having a unique reference to an node is not always ideal. Creating application policies sometimes requires genericity of the node references. For example, it may not be desirable to have to specify a security policy in terms of the path that an execution has taken to some point. This would require knowledge of all potential paths that the computation would be allowed to take. Reduction rules are used to create more generic names for use within application policies. For example, rather than refer to the name in Figure 6.10, it may be preferable to simply refer to:

```
(Lnam ["Hertz","Cork","TravelAgentAp","RentCar"])
```

Another aspect of name reduction is how these reduced names are maintained. There are two options: store the original unreduced name and a list of reduction rules to apply to the name, or store the reduced form of the name. Storing both the original name and the reduction rules allows different reduction rules to be applied, when necessary, during the lifetime of the name. We argue that it is easier to simply store the reduced name. The model does not ensure name consistency, for example, one reduction rule could remove information from a name that another requires. Ensuring that inconsistencies do not occur are the responsibility of the developer.

While the storing of reduced names approach is more optimal, the consistency argument applies only to static names. As a condensed graph executes, the names of the nodes change. In order to use dynamic names, reduction rules must be applied to names that have changed. Achieving this requires that we use the first approach: store the name and the list of reduction rules to apply to that name.

Reduction rules form two classes: tuple reduction rules that are applied to the tuples that make up a WebCom name and tuple elimination rules that are applied to the whole name to remove one, or more, of these tuples. We examine these types of reduction rules separately.

6.4.1 Tuple Reduction

Tuple reduction operates on the individual tuples of WebCom names. It reduces that tuple to what is considered an equivalent form. Reducing the components of a name is an application specific process, as, for example, the type of the inputs and results to nodes are potentially unique to a single application. For example, in the Travel Agent application (Figure 6.3), we are interested only in the airline name and the destination of the flight. Therefore, any name should be reduced to just these components, for instance, `(Lnam ["Aerlingus","EI220"])` from the name shown in Figure 6.6.

We can define a `tupleReduction` function to represent generic tuple reductions as follows: Let `tupleReduction :: Name -> Name` be a name rewrite rule that defines the tuple reduction rules used by an application. We use this construct to represent tuple reduction that is made up of a series of specific rules.

6.4.2 Tuple Elimination

A Tuple Elimination reduction rule is a rule that eliminates one of the component tuples—domain, graph, function, inputs or outputs—from a WebCom name.

```
tupleDElimination :: Name -> Name
tupleDElimination n = n{dom=Empty}
```

Given a name, `n`, then `tupleDElimination(n)` replaces the domain tuple of the name `n` with an empty name. Equivalent rules can be easily created for each of the tuples, as shown in Figure 6.12. We can use these basic rules, along with application specific rules to build complex security policies.

We define a `tupleElimination` function to represent generic tuple eliminations as follows: Let `tupleElimination :: Name -> Name` be a name rewrite rule that defines the combined tuple elimination rules used by an application.

Example 6.6 Figure 6.10 shows the name of a `RentCar` node. We can refer to this node as `rentCarName`, as shown below.

```
tupleGElimination :: Name -> Name
tupleGElimination n = n{grph=Empty}

tupleFElimination :: Name -> Name
tupleFElimination n = n{fun=Empty}

tupleIElimination :: Name -> Name
tupleIElimination n = n{ins=[]}

tupleOElimination :: Name -> Name
tupleOElimination n = n{ops=[]}
```

Figure 6.12: The Remaining Tuple Elimination Rules.

```
rentCarName =
  (Snam
    (Lnam ["Hertz","Cork"])
    (Lnam ["eBookers","Dublin","Alice","TravelAgentAp"])
    (Lnam ["Hertz","Cork","TravelAgentAp","RentCar"])
    [(Lnam ["Aerlingus","EI220","Paris","TravelAgentAp","BuySeat"])]
    [(Lnam ["eBookers","Dublin","Alice","TravelAgentAp","Print"])]
  )
```

Applying the tuple elimination rule `tupleIElimination(rentCarName)` results in the name:

```
(Snam
  (Lnam ["Hertz","Cork"])
  (Lnam ["eBookers","Dublin","Alice","TravelAgentAp"])
  (Lnam ["Hertz","Cork","TravelAgentAp","RentCar"])
  []
  [(Lnam ["eBookers","Dublin","Alice","TravelAgentAp","Print"])]
)
```

△

Example 6.7 We can take the node name `rentCarName`, from Example 6.6, and apply the following tuple reduction rule.

```
domTupleRedux :: Name -> Name
domTupleRedux (Snam d g f i o)
     | d == (Lnam ["Hertz","Cork"]) = (Snam (Lnam "Ryans","Cork") g f i o)
     | otherwise                    = (Snam d g f i o)
```

The `domTupleRedux` tuple reduction rule checks the domain tuple of a name and replaces any instance of `("Hertz","Cork")` with Hertz's Cork agent, `("Ryans","Cork")`. Applying `domTupleRedux(rentCarName)` results in the name:

```
(Snam
   (Lnam ["Ryans","Cork"])
```

```
      (Lnam ["eBookers","Dublin","Alice","TravelAgentAp"])
      (Lnam ["Hertz","Cork","TravelAgentAp","RentCar"])
      [(Lnam ["Aerlingus","EI220","Paris","TravelAgentAp","BuySeat"])]
      [(Lnam ["eBookers","Dublin","Alice","TravelAgentAp","Print"])]
   )
```

We can then apply the tuple elimination rules `tupleGElimination`, `tupleIElimination` and `tupleOElimination` in turn to result in the reduced name:

```
   (Snam
      (Lnam ["Ryans","Cork"]) (Empty)
      (Lnam ["Hertz","Cork","TravelAgentAp","RentCar"]) [] []
   )
```

△

6.4.3 Name Equivalence

Determining whether two names are equal is an important aspect of any enforcement mechanism using WebCom names. There are two basic approaches for comparing names. The first approach entails a simple comparison of unreduced names.

In contrast, the second approach is to apply reduction rules to the names and then compare the names. Figure 6.13 shows a definition of name equality based on reducing the names of nodes first, and then checking for equality.

```
instance Eq Name where
    Empty  ==   Empty    = True
    Empty  ==   x        = False
    n      ==   m        = tupleElimination(tupleReduction(n)) ==
                                   tupleElimination(tupleReduction(m))
```

Figure 6.13: Equality defined in terms of Reduction

Example 6.8 Consider comparing the (original) `rentCarName` name described in Example 6.7 to the name `bobRentCar` using the definition of equality shown in Figure 6.13.

```
 bobRentCar =
   (Snam
      (Lnam ["Ryans","Cork"])
      (Lnam ["USIT","Limerick","Bob","TravelAgentAp"])
      (Lnam ["Ryans","Cork","TravelAgentAp","RentCar"])
      [(Lnam ["Ryanair","FR776","Berlin","TravelAgentAp","BuySeat"])]
      [(Lnam ["USIT","Limerick","Bob","TravelAgentAp","Print"])]
   )
```

These names initially appear very different, as they refer to different travel agents, airlines and rental companies. However, a naming policy might only be interested in specific aspects of these names. If we first apply the tuple reduction rule, `domTupleRedux`, and then apply the graph, function, input and output tuple elimination rules to both `rentCarName` and `bobRentCar`, both of these names are reduced to the name:

```
(Snam (Lnam ["Ryans","Cork"]) (Empty) (Empty) [] [])
```

△

While we can define equality in this manner, in reality, it is not necessary to apply reduction rules to determine equality. Instead, we can effectively implement equivalence within the definition of `Eq`.

Example 6.9 An application makes a policy decision based on the domain where a node is to execute. In this case, we redefine the derivation of the `Eq` (Figure 6.14) or equality property of `Name` to check only this information.

```
instance Eq Name where
    Empty                   == Empty     = True
    Empty                   == x         = False
    (Snam a b c d e)        == (dom:f)   = (a == f)
    (Pnam p)                == (Pnam q)  = (p == q)
    (Lnam l)                == (Lnam m)  = (l == m)
    x                       == y         = False
```

Figure 6.14: A definition of the Equality relation.

This definition of equality acts implicitly as a reduction rule. Primitive and linked names are compared based on their text, while structured names are compared based only on their domain tuple. The result is that any structured name is implicitly reduced to its domain tuple.

△

Example 6.10 We modify the definition of equality from Example 6.9 to:

```
instance Eq Name where
    Empty                   == Empty                   = True
    Empty                   == x                       = False
    (Snam a b c d e)        == (dom:f)                 = (a == f)
    (Pnam p)                == (Pnam q)                = (p == q)
    (Lnam ["Hertz","Cork"]) == (Lnam "Ryans","Cork")   = True
    (Lnam l)                == (Lnam m)                = (l == m)
    x                       == y                       = False
```

In this definition, we have embedded the tuple reduction and tuple elimination rules used in Example 6.8. This definition compares names based on their domain tuples, and implicitly replaces any instance of `(Lnam ["Hertz","Cork"])` with `(Lnam ["Ryans","Cork"])`. In this case comparing the names `rentCarName` and `bobRentCar` will again determine that they are equivalent according to this definition of equality.

△

Note that there is no canonical definition for the name of a node. This means that a node could be referred to using two, or more, different (non-equivalent) names. Further research work is required to develop a naming theory whereby two different names that refer to the same object can be reduced to a canonical form.

6.4.4 Reduction Rule Application Order

The order in which tuple reductions take place is significant. Tuple reduction rules are not necessarily commutative, that is if we have two tuple reduction rules, S and T and a name n then

$$S(n), T(n) = T(n), S(n)$$

does not necessarily hold. Thus, applications must specify the order that tuple reduction rules are applied. However, tuple elimination rules are commutative. Tuples are atomic, in that applying a tuple reduction rule on one tuple does not effect any other tuples. Removing any one tuple also does not effect any other tuples.

6.4.5 Creating and Updating Names

Node names in a condensed graph are continually being updated during the execution of the graph. There are several conditions under which a node name can change:

1. When the name of a node is first created, it contains the basic information available at that time, namely the function and graph tuple of the node. Therefore, an initial node name appears as: `(Snam (Empty) g f [] [])`. Inputs and outputs may also be represented by the name of the node(s) that the input arrives from, and the name of the node(s) that the result of this node is sent to, respectively.

2. When input arrives to a node, the name of the node is updated to reflect this change. The name can be changed to the value of the input. Alternatively, the name of the node that provided the input is retained. This decision depends on the naming policy. A naming policy could require that values replace node names whenever a node executes, or alternatively, that the name of the node that has executed is retained.

3. When a node executes and returns a result, the name of the node may be updated to represent the value that has resulted from the execution. The result might be incorporated into the function tuple. For example, the result “5” of a divide operation could be represented in the function name: `(Lnam ["divide","result","5"])`.

4. When a node is scheduled to a domain, the name of the node is updated to include this domain name.

In each of these cases, when the name of a node in a graph is updated, this update may potentially effect every other node name in the graph.

6.5 History-based Names

Frequently, policies are constructed to consider the historical contexts of a computation. For example, access control mechanisms, such as Separation of Duties [135], Chinese Walls [40, 58, 106] and High Watermark [180] policies, all depend on the history of the computation in order to make decisions. Other types of policy, such as a reactive load balancing policy that bases its future scheduling decisions on the load of the resources utilised to date, can also rely on the contexts that a computation has passed through. With condensed graphs, the context information is stored in the node names. Thus, in order to make an history-based decision for a condensed graph, the relevant contextual information must be extracted using history-based reduction rules

A node `n` has a name `(Snam d g f i o)` whose attributes provide the names of execution contexts for domain (`d`), application graph (`g`), function (`f`), list of inputs (`i`) and list of outputs (`o`), respectively. The *history* reduction of the name of node `n` is defined as

```
tupleElimination(Snam d g f (map tupleReduction i) o)
```

where `tupleElimination` and `tupleReduction` represent the application of tuple elimination and tuple reduction rules, respectively. The tuple reduction rule is only applied (using the `map` function) to the inputs. The `map` function applies the function, in this case `tupleReduction`, in turn to each member of a list. This characterises history reduction as the inputs to a node specify the path that the execution has taken to this point.

Example 6.11 To perform a history based policy decision, we take the inputs to a node and reduce their execution contexts. Figure 6.15 shows a representation of such a rule in the form of a function `saveDomHist`.

This reduction rule defines that an execution context should contain the context(s) of the input(s). The relevant details, in this case the domain tuple, from the input tuple is extracted and

```
saveDomHist :: Name -> Name
saveDomHist (Snam a b c [(Snam v w x y z)] e) = (Snam a b c [v] e)
```

Figure 6.15: A simple Reduction Rule to retain domain history

integrated into the name of the current context. This replaces the domain tuple of the current context with the domain of the input. This reduction rule is applied to the input tuple of the name, that is `tupleElimination(Snam d g f (map saveDomHist i) o)`. This is a very simple example of a `tupleReduction` reduction rule.

△

Example 6.12 A more useful example of this type of reduction rule can be seen in Figure 6.16. In this example, the domain of the input replaces the domain of the current context only when the domain of the input is strictly "greater" than the domain of the current context. The `saveDom` function acts as filter. If it is applied to each node in the graph in turn, then it ensures that the "greatest" domain is retained in every name.

```
saveDom :: Name -> Name
saveDom (Snam a b c [(Snam v w x y z)] e)
          | a < v       =   (Snam v b c [(Snam v w x y z)] e)
          | otherwise   =   (Snam a b c [(Snam v w x y z)] e)
```

Figure 6.16: Domain ordering rules

In Section 6.3, we described how to define an ordering between names using the `Ord` relation. In Figure 6.17, we explicitly define the ordering `(Pnam "Alice")` $<$ `(Pnam "Bob")` $<$ `(Pnam "Claire")`.

```
instance Ord Name where
  Empty              <= x                     =  True
  (Snam a b c d e)   <= (Snam v w x y z)      =  (a <= v) && (b <= w) &&
                                                 (c <= x) && (d <= y) && (e <= z)
  (Pnam "Alice")     <= (Pnam "Bob")          =  True
  (Pnam "Bob")       <= (Pnam "Claire")       =  True
  (Pnam p)           <= (Pnam q)              =  p == q
  (Lnam l)           <= (Lnam m)              =  l <= m
  x                  <= y                     =  x == y
```

Figure 6.17: Definition of the `Ord` relation.

△

Example 6.13 A node that is to be executed in domain `Claire` has the name:

```
(Snam (Pnam "Claire") (Pnam "MathGraph") (Pnam "MulFunction")
      [((Pnam  "Bob") (Pnam "MathGraph") (Pnam "DivFunction") [] [])] [] )
```

If we apply the `saveDom` reduction rule to this name, then the reduced name becomes:

```
(Snam (Pnam "Bob") (Pnam "MathGraph") (Pnam "MulFunction")
      [((Pnam  "Bob") (Pnam "MathGraph") (Pnam "DivFunction") [] [])] [] )
```

This means that the node will now be executed in domain `Bob`. Such reduction rules could be used to help ensure data. For example, results of executions, is not sent to inappropriate domains, or computations executed on inappropriate resources. However, enforcing such requirements is the responsibility of the relevant policy enforcement mechanism. △

6.5.1 Naming Grid Submissions

The computational Grid allows the sharing of compute resources between different organisations. WebCom, using condensed graphs, can be used to create Grid applications [124]. As Grid applications execute across many domains, it is important to ensure that application components can be referred to properly wherever they execute.

Example 6.14 A national compute Grid is organised in a tiered structure. In order to support system policies, the name of context where every job (graph) is submitted must be retained in every node name. The Grid resources are structured to allow job submission at various layers in the overall architecture, shown in Figure 6.18.

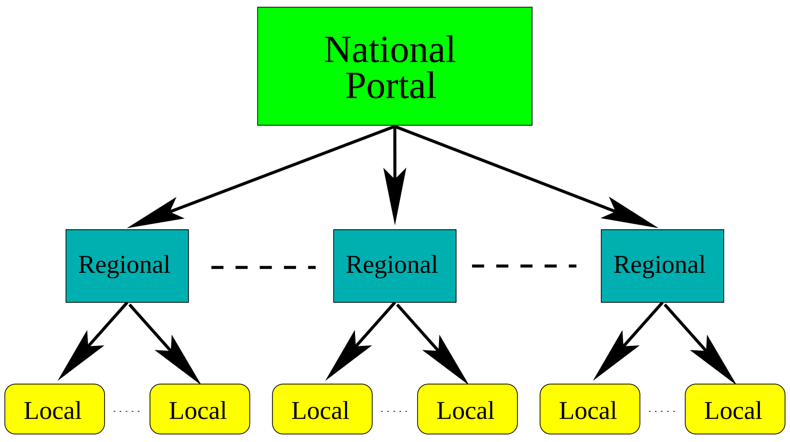


Figure 6.18: GRID Portal Structure.

In this architecture, jobs can be submitted at national, regional or local portals. The specific policy requirement is that a computation launched at a regional or local portal must not move up the architecture, even though the portals are interconnected. For example, when a job is submitted on a regional portal, it can move down to local resources underneath that regional portal, but not up to the national portal or across to other regional portals. We call this a "*ceiling*" restriction and can specify it in a naming policy.

This requirement can be described by a reduction rule that maintains the submission portal as part of each node's name. For example, the travel agent condensed graph application shown in Figure 6.3 can be submitted to a Regional portal. The name of the portal becomes part of the name of every node in the submitted graph.

```
(Snam
    (Lnam ["Grid Ireland","Munster","UCC","Compute Cluster"])
    (Pnam "eBookers")
    (Lnam ["eBookers","RentCar"])
    [(Lnam ["eBookers","BuySeat"])]
    [(Lnam ["eBookers","Print"])]
)
```

Figure 6.19: The name for the `RentCar` node, executing on a particular GRID resource.

Normally the domain portion of a name is defined as the computational resource that is executing, will execute, or has executed, the node, when this is known. When making a naming policy decision about a node, the *ceiling* reduction rule is applied to the name of the node.

```
saveDom :: Name -> Name
saveDom (Snam a b c [(Snam v w x y z)] e)
                | a < v       =   (Snam v b c [(Snam v w x y z)] e)
                | otherwise   =   (Snam a b c [(Snam v w x y z)] e)
```

Figure 6.20: Ceiling Reduction Rule

The tuple reduction rule, `saveDom`, shown in Figure 6.20[1], is an example of such a reduction rule. When applying this rule, we assume that the names we are applying it to are of the correct form. When used with an appropriate definition of the `Ord` relation, this rule ensures that the parent graph's domain is maintained within the node name and thus the ceiling is never exceeded. The `Ord` relation would be similar to the relation shown in Figure 6.17, but with Local $<$ Regional $<$ National $\triangle$

6.5.2 Web Services Policy

Up to this point, we have seen how reduction rules that can be applied to names to ensure existing information is retained. However, this is only one application of such rules. Reduction rules can also be used to inject new information into a name.

Example 6.15 Airlines have business relationships with specific hotel and car rental companies. For example, in Ireland the Hertz car rental company gives specific discounts to customers who fly with either Aerlingus and Ryanair; in contrast, Easyjet discounts Europcar. Another example is when flying into London on an Irish airline, customers are recommended to stay in one of the

[1]This is the same reduction rule as shown in Figure 6.16.

Jury group's hotels. We can represent these types of relationship using reduction rules in the web services graph shown in Figure 6.3.

```
selectCarCo :: Name -> Name
selectCarCo (Snam a b c [(Snam v w x y z)] e)
             | ((v == (Pnam "Aerlingus")) && (c == (Pnam "RentCar"))
                   =   (Snam a b c [(Snam (Pnam "Hertz") w x y z)] e)
             | ((v == (Pnam "Easyjet")) && (c == (Pnam "RentCar"))
                   =   (Snam a b c [(Snam (Pnam "Europcar") w x y z)] e)
             | otherwise         = (Snam a b c [(Snam v w x y z)] e)
```

Figure 6.21: Airline Car Rental preference rules

Figure 6.21 defines such a reduction rule. The `selectCarCo` reduction rule matches the domain that an input node has executed in with known airlines. If the current node is a `RentCar` node, then it updates the domain name of the current node to that of the airline's car rental partner. If an unknown airline is selected, then no change is made to the node name.

△

This example identifies some interesting possibilities. As the naming model is outside of the condensed graph model, it is possible to use the naming system to effect where nodes in the graph are scheduled. WebCom can use these names to help determine where to schedule nodes. The flexibility of the naming system is of the order of an additional programmable system outside of the condensed graph model. The question now arises: are these programmable side-effects an advantage or a hindrance to the condensed graph model? Traditionally in the condensed graphs model, side effects are frowned upon. However, a number of features in individual applications rely on side effects. For example, a side effect can be used to support concurrency in condensed graphs [134]. Another example of a common side effect is how files are transfered between resources for certain condensed graphs applications. This is also provided outside of the model.

6.6 An API for Naming in WebCom

In this Section, we outline an implementation of the naming model for WebCom. We provide an API to support WebCom names throughout the entire WebCom architecture. Maintaining the names of the nodes is the responsibility of WebCom's naming manager module. The naming manager maintains a representation of each node in the executing graph and updates this name during the graph's execution. As the graph executes, the names of the nodes evolve, representing changes that occur in the graph itself. For example, when a node executes and returns a result, the names of the nodes receiving that result are updated to reflect this change.

The implementation of the naming system in WebCom consists of the naming manager module, a number of name generators, an abstract name type, and an abstract representation of reduction rules. These classes provide the ability to define and modify node names for use by the other WebCom modules.

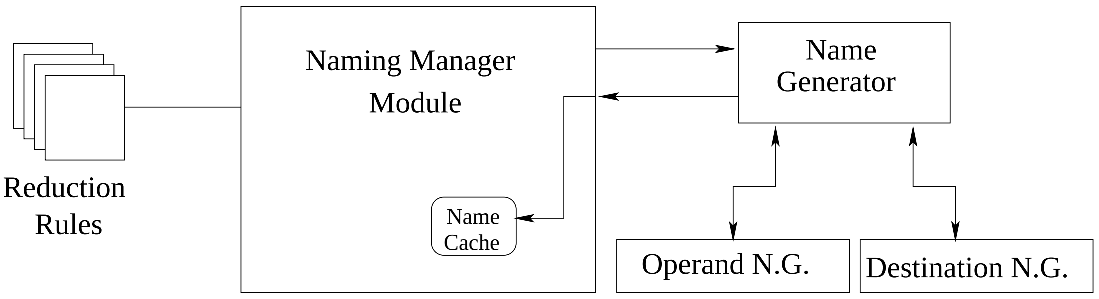


Figure 6.22: WebCom's Naming Architecture

Figure 6.22 shows a representation of the naming software architecture. The main component of this architecture is the NamingManagerModule. However, we will first describe the other components of the architecture, as they are all used by the NamingManagerModule.

6.6.1 webcom.core.naming.ReductionRule

Class ReductionRule is the abstract definition of a name reduction rule. It is passed a WebCom name (described in Section 6.6.3) and returns the reduced form of that name. Reduction rules are application (graph) specific. Each application has its own security policy requirements. These requirements drive the requirements for the reduction rules.

```
public abstract SexpList reduce(SexpList Name);
```

ReductionRule is a simple class with one major method `r.reduce(N)`. This method takes a list of s-expressions and performs (rule dependent) actions on that list. The reduced form of the s-expression is returned to the caller. The default reduction rule is defined *abstractly*, that is, it has no implementation code. Instead, it must be inherited so that subclasses provide the implementation.

A simple reduction rule can take a WebCom name and return a representation of the node that describes the function of the node. This emulates of the operation of the early Secure WebCom prototypes [66]. A sketch of this reduction rule is shown in Figure 6.23.

While we typically associate naming policies with access control policies, WebCom naming policies can also be used to address other concerns, such as load balancing or fault tolerance. The enforcement mechanism remains the same. In these cases, new reduction rules must be defined for such policies.

6.6.2 webcom.core.naming.NameGenerator

Node names can be generated either a priori by the developer of the graph or, more typically, when the name is needed to make a security decision about the node. Generating names for nodes entails

```
public class SimpleReductionRule extends ReductionRule
{
  public SexpList reduce(SexpList Name)
  {
     //Extract Function tuple from Name and return this as a new
     // S-Expression.
     SexpList newname = new SexpList(extractFunction(Name));
     return (newname);
  }
}
```

Figure 6.23: The implementation of a simple reduction rule's *reduce(n)*

examining the node at a particular point in time and extracting the relevant details.

The name of a WebCom node is returned by its name generator. The name generator examines an instance of a condensed node and extracts the relevant details. Name generators are used to create fresh names for nodes. When WebCom loads a graph for execution, the NamingManagerModule identifies the nodes without names in the graph and generates names for those nodes. A node's name is generated using an internal condensed graph examiner class. This examiner can investigate the internals of a loaded condensed graph on a "read-only" basis.

As nodes in condensed graphs refer to each other, their names will also contain references to other node names. In Section 6.2.2 we discussed the potential for self-referencing names. As each node's destination can be another node operand, the name of the destination node must also be included in the original node's name. However, as the destination node will also have a reference to the original node, care must be taken to avoid recursive loops. When a node name is generated, this situation must be addressed.

The solution is to apply specific reduction rules when generating the name of a node's operand and/or destination. We use, respectively, the OperandNameGenerator class and the Destination-NameGenerator class to accomplish this goal. These name generators are in fact special reduction rules that are only called by the name generator. Their specific task is to prevent name recursion. For example, the simplest approach to preventing name recursion is to apply tuple elimination rules to the inputs and/or outputs of a node, removing any reference to the current node.

```
public WebComName generateNameForNode(Node node, ReductionRule reduxrule,
                                          String Domain)
public void setOperandGenerator(OperandNameGenerator operandGenerator)
public void setDestGenerator(DestinationNameGenerator destGenerator)
```

NameGenerator classes are called by NamingManagerModules to generate new names for nodes. The application interface for this class is relatively simple, with the most significant method being the `ng.generateNameForNode(n,r,d)` performing the name generation. Name generation can be potentially application specific. In most cases application requirements are handled by customised reduction rules. In Chapter 8, we will examine some specific WebCom applications.

webcom.core.naming.DestinationNameGenerator

Destination name generators generate representive names for the destination of a node. When a node's name is generated, it is important to prevent recursion in the destination. In Section 6.2.2, we identified the problem of self referencing names. Destination and operand name generators address this problem by explicitly preventing recursion. These name generators act as reduction rules that are applied to the names of destination nodes to derive a non-recursive representation of that node. However, they are called name generators as these reduction rules are only used during name generation. DestinationNameGenerators are exclusively called by NameGenerators.

```
protected String generateNameForDestination(CondensedGraph cg,
                                            int DestinationNodeID);
```

The `dng.generateNameForDestination(g, did)` is the only major method in this class. As it is only called from a name generator, condensed graph internal representation is used to identify the specific destination node. The reference version of this class simply returns the function tuple of the destination node's name, such as shown in Figure 6.23. This is equivalent to applying the haskell reduction rule function `destReduxRule` shown in Figure 6.24.

```
retainFunc :: Name -> Name
retainFunc (Snam d g f i o) = f

destReduxRule :: Name -> Name
destReduxRule (Snam d g f i o) = (Snam d g f i (map retainFunc o))
```

Figure 6.24: A destination reduction rule retaining the function tuple

In this example, the `retainFunc` function is used instead of applying the domain, graph, input and output tuple elimination rules. However, as destination name generators use the internal condensed graph API to determine the nodes in question, standard reduction rules cannot be substituted.

webcom.core.naming.OperandNameGenerator

Operand name factories are the corollary to destination name factories. They are specific reduction rules that address the potential for mutual recursion in the names of operands. OperandNameGenerators are also exclusively called by NameGenerators.

```
protected String generateNameForOperandNode(CondensedGraph cg,
                                            int OperNodeID);
public String generateNameForOperandValue(Object Input);
```

As with destination name generators, the main method in an operand name generator is the `ong.generateNameForOperandNode(cg, oid)` method. This method investigates the operand node and extracts a representation of the operand node.

Unlike Destinations, operands may, when the operand node has executed, consist of the result of the operand node. In condensed graphs, a result can be anything, ranging a simple numerical value to complex objects such as a spreadsheet. When an operand has returned a result, the operand name generator can use the `ong.generateNameForOperandValue(i)` method to extract a representation of this result and use this to refer to the operand.

In some cases, it is more important to specify where the operand result came from rather than the value itself. In Section 6.5, we described some history based naming policies. The operand name generator is an important part of such policies. In such cases, the operand name generator has the ability to look at the historical path the operand has taken to this point. A representation of this historical path can then be used as the name of the operand. This capability can be used, for example, to enforce history-based security policies, such as high watermark or Chinese wall [40] policies.

6.6.3 webcom.core.naming.WebComName

Recall from Section 6.2 that a node's name consists of a five-tuple: the *domain* it is (or will be) executing in; the *graph* it is a member of; its *function*; the *operand(s)* to the node; and its *Destination(s)*. WebComName stores a representation of a node's name. When a node has not been selected for execution, its domain may not be a priori defined. In this case, only when a node is scheduled will the domain be known, and thus be reflected by the name the naming manager module stores. Alternatively, a user could define the domain in which a node must execute. In this case the naming manager would store this information within the node's names. When no domain name has been assigned to a node, the domain tuple is unassigned. This is equivalent to an `Empty` name in the naming model.

The WebComName class stores the details defining a node. All component tuples of the WebComName can be individually set and retrieved. In the current implementation of a WebComName, s-expressions [153] are used to store this information. The Java s-expression library from the JSDSI project is utilised [5]. A s-expression containing a representation of the node's name can be retrieved from the class. This representation can be modified by applying one or more reduction rules to the name. These reduction rules are specified within ReductionRule classes. The number, type and the order of reduction rules to apply are application specific settings and are specified when configuring WebCom.

The WebComName API is split into two categories, the simple string-based interface and the more complex s-expression based interface. The string based interface is used to convert WebComNames into simple text and to parse text to represent it as s-expressions. However, this interface does not maintain the full name of a node. S-expressions provide context to a string. Extracting the string implies losing this contextual detail.

```
public void setDomain(java.lang.String domain)
public void setGraph(java.lang.String graph)
public void setFunction(java.lang.String function)
public void setDestination(java.lang.String Destination)
public void setInput(java.lang.String input)
public void addDestination(java.lang.String Destination)
public void addInput(java.lang.String input)
```

The API has simple “setter” methods to store the detail of a node. As a node is investigated, these methods are used to build up a representation of the node. There are equivalent “getter” methods to extract a string-based representation of the five-tuple.

The s-expression interface represents how information contained within the WebComName is stored and manipulated. This interface allows the calling module to retrieve the current name of a node without losing any contextual detail. In this case, the WebComName class provides the ability to retrieve the full name of the node as well as a reduced form, based on the reduction rule(s) applied to the name.

```
public jsdsi.sexp.SexpList getName()
public jsdsi.sexp.SexpList generateName()
public jsdsi.sexp.SexpList reduceName(ReductionRule rule)
```

6.6.4 webcom.core.naming.NamingManagerModule

The naming manager module in WebCom maintains a representation of all the nodes in the currently executing graph. As nodes become fireable and execute, their names evolve to represent these changes. All nodes in a graph are linked and so changes to a single node effect every node in the graph. The naming manager provides information about nodes for the other modules in WebCom.

Node names reflect the current state of the graph along with any predefined policies within WebCom. For example, when a graph is created, specific naming information can be embedded into node definitions. Typically, predefined names are stored using an XML representation of a WebComName[2] [127]. When WebCom is executing an application, the NamingManagerModule is constantly updating the names of the nodes of that application.

The NamingManagerModule class provides the following public interface:

[2]The XML representation of WebCom names is outlined in Appendix A.

```
    public WebComName getNameForNode(Node n, ReductionRule r,
                                     Descriptor domain);
    public void loadProperties(String propertiesFile);
    public void setNameGenerator(NameGenerator generator);
    public void setReductionRule(ReductionRule rule);
```

The NamingManagerModule is initialised by WebCom's scheduler, known as the *Backplane*. As with all WebCom modules, it loads its initial properties from a stored configuration file using the `nmm.loadProperties(pFile)` method. These settings include selecting the particular name generator and reduction rule(s) to use. These setting can be changed at runtime using the `nmm.setNameGenerator(g)` and `nmm.setReductionRule(r)` methods. This functionality is utilised when customised name generator and reduction rules are needed for a specific application.

Names for nodes are constantly changing as a graph executes. As all nodes are connected, each node that is scheduled or executes causes the names of all the other nodes in the graph to change. The NamingManagerModule manages how these names are modified and stores a representation of each nodes name in a *name cache*. Node names are updated in two cases: when a node is scheduled and when a node executes and returns a result. These actions cause the NamingManagerModule to update the names of all the nodes in the current graph, using the specified reduction rules. If a node does not have a current name, then the NamingManagerModule calls the configured name generator to generate a name for that node. This happens, for example, when a subgraph is evaporated and no predefined names are present.

When another module asks for a name for a specified node, the NamingManagerModule checks its name cache for the current name of that node. It will then update the name when required. Updating the name is needed when, for example, the node is to be scheduled to a particular domain. The naming manager will then apply the specified reduction rule(s) on the name and return the reduced name to the calling module.

6.7 Discussion and Conclusion

In this chapter, we have proposed a naming model for condensed graphs. This model allows the creation of sophisticated policies using these names. Naming the nodes in a condensed graph entails capturing a representation of the context in which the node is executing, as well as the specific details regarding that node, such as function, inputs and outputs. These names are dynamic: as the computation progresses the names of the nodes evolve.

WebCom names are structured so that they can represent nodes with arbitrary precision. However, using these names requires having the ability to convert them to a more usable form. Reduction rules are used to take complex names and remove unnecessary information. We use reduction rules to help create specific policies, such as history-based policies. A history-based policy is one where information about what has transpired during the execution of the graph is stored within the names of nodes. For example, we can use history-based policies to store the names of the domains that the nodes have executed during the computation.

Complex distributed systems, such as CORBA, use abstract names to hide underlying detail of the operating system from application developers. However, hiding operating system details can cause problems when developing security policies for those systems [111]. Names in WebCom are not abstractions. The contextual details of the operating system that are considered important can be made available within the name.

WebCom policies are limited to a single execution context and will not effect future executions of the same graph, nor the authorisation of any other graphs being executed concurrently. Work is ongoing towards providing support for concurrency within Condensed Graphs [134].

The definition of reduction rules is also limited in the current model. The model does not guarantee name consistency. For example, one reduction rule could remove information that a later reduction rule relies upon.

Example 6.16 Recall the tuple reduction rule `domTupleRedux` applied to the `rentCarName` from Example 6.7. The `domTupleRedux` rule modifies the rental company name so that Hertz's local agent, Ryans, replaced Hertz when renting a car in Cork. Consider a second reduction rule, `upgradeCar`, that is applied to Aerlingus customers who rent from Hertz:

```
upgradeCar :: Name -> Name
upgradeCar (Snam d g f i o)
   | (d == (Lnam ["Hertz","Cork"]) &&
      f == (Lnam ["eBookers","RentCar"]) &&
      True ‘elem‘ [ip <= (Lnam ["Aerlingus"]) | ip <- i])
                         = (Snam d g (Lnam ["RentCar","Upgrade"]) i o)
   | otherwise           = (Snam d g f i o)
```

This reduction rule upgrades qualifying customers to a better vehicle. If `domTupleRedux` is applied before `upgradeCar`, then Hertz customers renting in Cork will never receive this upgrade. However, when `upgradeCar` is applied, and then `domTupleRedux`, customers will receive the upgrade, and the rental company name will be updated to local agent, Ryans.

△

Our position in this dissertation is that it is the responsibility of the specifier of the rewrite rules to ensure these inconsistencies do not occur. However, the development of a formal model to prevent such inconsistencies is a potential topic of future work. Such a model could be used to prove the consistency and completeness of a set of reduction rules. There has been substantial research [51] into rewrite systems. Such systems provide formal analysis and proof for name rewriting. Developing such a model for names is a topic of future research.

This chapter outlines the current implementation of the naming model in WebCom. Chapter 7 uses this naming architecture as a basis for WebCom's access control model. In Chapter 8, we examine some complete case studies using WebCom's naming and access control architectures.

Chapter 7

WebCom Security Model

In Chapter 5, we introduced the WebCom distributed metacomputing environment. In this chapter we describe the security model and architecture of WebCom in more detail. WebCom's architecture provides the ability to make security policy decisions about the execution of computations. The enforcement of security policy decisions is the responsibility of WebCom's security architecture.

Threats to a distributed computing environment include the illicit modification of data used in a computation; the modification of the computation itself; the unauthorised access of data by principals; the unauthorised execution of computations, and identity theft. These threats are addressed in WebCom through the provision of data and computation integrity and by authenticating the principals using the system.

Data and computation integrity involves ensuring that both the data used in the computation and the computation itself are not modified illicitly. Integrity is important as computations can potentially execute across compute resources that are controlled by many different entities. For example, spoofing of results is a common problem in volunteer based distributed computations [45]. An authorisation mechanism to ensure that computation execution and data access is only performed by authorised principals is needed. Such a mechanism will ensure the integrity of both computations and data.

Managing and verifying the principals using a distributed computing environment entails ensuring that there is a systematic way to determine the authenticity of the principals and the resources used in the computation. This can be provided through the use of authentication mechanisms.

The WebCom security architecture is designed to address both authorisation and authentication. The authorisation mechanism is access control based. We argue that the goal of access control for distributed computations is fourfold. It can be characterised as the need to ensure that: computations will only be executed on resources that are explicitly authorised; resources will only execute computations that come from authorised servers, results of computation execution will only be accepted from resources that are authorised, and these results are received only by the authorised recipient. In an access control based security architecture, access to an object is authorised whether the subject has been granted permission to use the object in the requested way.

The authentication problem in WebCom can be characterised as the requirement of two principals to set up a communication channel whereby each principal believes that they are communicating only with the other principal.

WebCom's security architecture addresses authorisation and authentication separately:

- WebCom's authorisation architecture is supported by the Naming and Security Manager Modules. In Chapter 6, we described how nodes in condensed graphs can be named. By specifying the exact conditions under which a named node may execute, sophisticated security policies may be written for those nodes.

- Authentication is supported by WebCom's communications manager. This entails using a secure authentication and/or communication protocol, such as SSL [92] or IPSec [172], and providing support for a public key infrastructure (PKI), when necessary. Providing authentic and secure connections between WebComs ensures that data is sent to the correct destination, and cannot be intercepted, or modified, by a third party.

We can regard the security manager as acting as a reference monitor, checking security critical actions and ensuring that these actions comply with the access control policy. In this chapter, we examine the security model and architecture of WebCom. Section 7.1 describes WebCom's access control model. In Section 7.2, we examine some examples of security policies that can be enforced using the WebCom security architecture. Section 7.3 describes the implementation of the security architecture in WebCom. The architectural support for authentication is outlined in Section 7.4.

7.1 WebCom Access Control Model

An access control model captures the set of allowed actions as a policy within a system. Access control addresses a primary concern for security in a system: deciding whether access to a resource is permitted or denied. Recall from Chapter 2 that a *reference monitor* implements access control. A reference monitor typically operates as follows: a security critical action is required, for example, an access request for sensitive data, the reference monitor intercepts the action and checks whether the action is authorised according to the security policy. If it is, then the action proceeds. Otherwise the security critical action is not authorised and the caller is notified of this failure.

WebCom's security architecture follows this model of policy enforcement. Figure 7.1 shows a representation of the WebCom reference monitor. When a node is to be executed, this "security critical" action is mediated by the "WebCom reference monitor". In this case, part of the Scheduler, NamingManagerModule and SecurityManagerModule act in concert to decide whether the security critical action is authorised. The Scheduler module selects one or more potential WVM

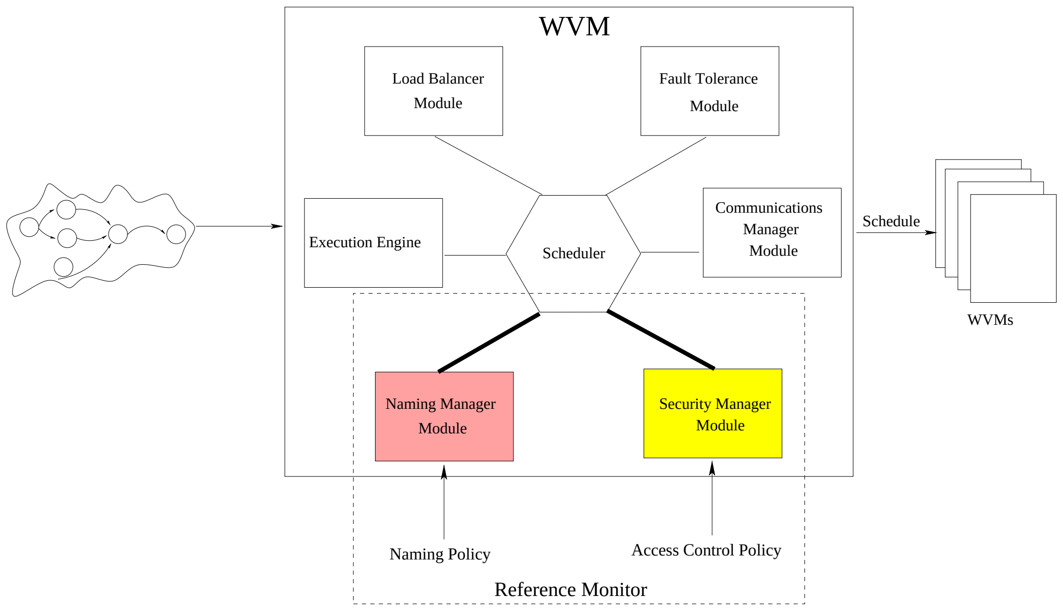


Figure 7.1: WebCom's Reference Monitor

(WebCom Virtual Machine) targets for the action (node to be executed). The NamingManagerModule extracts the relevant details of this action. The SecurityManagerModule then makes the decision based on the security policy and the action's details supplied by the NamingManagerModule. If authorised, then the node is scheduled to the authorised WVM. Otherwise, the node is rejected and the scheduler is informed.

7.1.1 WebCom Permissions

The security architecture in WebCom allows control over the execution of applications running in the system. Authorised actions are specified in the security policy, and the WebCom reference monitor ensures that only authorised actions take place. Recall from Chapter 6 that the definition of a WebComName is:

```
data Name  =
             Empty
          |  Pnam {pnam :: String}
          |  Snam {dom  :: Name,
                   grph :: Name,
                   fun  :: Name,
                   ins  :: ([Name]),
                   ops  :: ([Name])}
          |  Lnam {lnam :: ([String])}
```

Names in WebCom are defined in terms of Haskell [98] equations.

The access control policy ensures that only authorised WVMs are assigned work. The names of WVMs are represented within the WebCom naming scheme. The set of all possible names for

WVMs is defined as a subtype of `Name`, and represented as:

```
type WVM = Name
```

In a WebCom environment, nodes are assigned as work to WVMs. The set of all possible names for nodes is defined by a subtype of `Name` and represented as:

```
type Node = Name
```

Let `Permission` denote the set of all permissions in WebCom. WebCom provides two main capabilities: scheduling and execution of nodes. These capabilities are considered as permissions and are represented as follows:

```
data Permission :: Sch Name | Exe Name
```

Example 7.1 Figure 6.19 showed the (reduced) name of `RentCar` node. The permission to execute this node can be specified as follows:

```
(Exe (Snam
        (Empty)
        (Pnam "eBookers")
        (Lnam  ["eBookers","RentCar"])
        [(Lnam ["eBookers","BuySeat"])]
        [(Lnam ["eBookers","Print"])]
     )
)
```

The permission does not constrain the WVM domain in which the node is executed. A variation on this permission for a specific WVM domain is as follows:

```
(Exe (Snam
      (Lnam ["Grid Ireland","Munster","UCC","Compute Cluster"])
      (Pnam "eBookers")
      (Lnam ["eBookers","RentCar"])
      [(Lnam ["eBookers","BuySeat"])]
      [(Lnam ["eBookers","Print"])]
     )
)
```

This permission specifies that the UCC compute cluster is authorised to execute this `RentCar` node.

△

7.1.2 Ordering Permissions

A partial ordering (transitive, asymmetric, reflexive) is defined on the set of permissions, whereby, `(p1 <= p2)` is interpreted to mean that holding permission `p2` implies holding the permission `p1`. This partial ordering (the `Ord` of datatype `Permission`) is used to specify the access control policy. Ordering permissions is a traditional convention used for access control. For example, both the Java security model and RBAC also use a partial ordering of permissions [78].

Example 7.2 Figure 7.2 shows an example of the `Ord` relation for `Permission`. As `Sch` and `Exe` permissions are disjoint, they are compared separately. In both cases, permissions are compared based on the names of the nodes they refer to. In this example, one permission is less than another when the name of the second node is less than (or equal to) the name of the first node.

```
instance Ord Permission where
  Exe n    <= Exe m         =  m <= n
  Sch n    <= Sch m         =  m <= n
  x        <= y             =  False
```

Figure 7.2: Definition of the `Ord` relation for `Permission`.

In this example, `Permission` partial orderings are functionally opposite to `Name` partial orderings. Names are ordered such that when more information is present in the name, that name more precisely identifies the node. For example, a node name with input value “5” is more precise than the same node name without this input information. In contrast, permissions are ordered so that a less precise permission means that the holder has more rights. For example, a permission for the node that specifies the input value “5” is more restrictive than one that does not have this requirement. Precision in the case of both names and permissions refers to how much uniquely identifying information is present in the name or permission. The most precise name for any given node will have all possible information stored within the name.

In `Permission` orderings, an `Empty` permission represents the greatest possible permission in the ordering. Granting an `Empty` permission to an entity is equivalent to granting the Java permission `AllPermissions` to an entity in the Java security model. △

Example 7.3 The WVM permission shown in Example 7.1 shows a very specific permission. In contrast, the *highest* execute permission for the `Compute Cluster` WVM is as follows:

```
(Exe (Lnam ["Grid Ireland","Munster","UCC","Compute Cluster"])
                                             (Empty) (Empty) [] [])
```

As this permission does not explicitly refer to any specific node, or set of nodes, it is greater than any permission that has the same references in its domain tuple. This permission authorises the WVM to execute *every* possible node.

Furthermore, the execute permission:

```
(Exe (Lnam ["Grid Ireland"]) (Empty) (Empty) [] [])
```

is less precise again, and so is greater than the `Compute Cluster` execute permission. △

In principle, `(Permission,<=)` can have have any definition (provided it is a partial ordering). However, in practice, there are a number of constraints that we place on the ordering. The name of a WebCom entity changes only in accordance to the behaviour of the execution engine (to reflect inputs arriving to nodes, node firing, and so forth) and the rewrite rules; we must ensure that any potential permission ordering is consistent with this behaviour.

Many of the changes to a name simply extend it. For example, the arrival of an input value to a node results in its name changing from `Empty` to a name reflecting its value. Node names are either increasing in detail, when, for example, new information arrives, or decreasing in detail, for example, when reduction rules are applied to the names. We require that orderings on permissions are monotonic with respect to the names that they refer to, that is,

```
MonoNames :: Name -> Name -> Bool
MonoNames n1,n2 = if (n1 <= n2) then
                     (Sch n2) <= (Sch n1) &&
                     (Exe n2) <= (Exe n1)
```

`MonoNames(n1,n2)` defines the relationship between the names of two nodes in terms of the permissions required to execute and/or schedule those nodes. It defines that as the name of a node grows more precise, the permission based on that name becomes more restrictive.

Example 7.4 If the name of a node is:

```
(Snam (Empty) (Pnam "GraphA") (Pnam "FuncA") [] [])
```

then the permission required to schedule that name is simply:

```
(Sch (Snam (Empty) (Pnam "GraphA") (Pnam "FuncA") [] []))
```

However, if the name of the node is updated to contain a domain:

```
(Snam (Pnam "DomainA") (Pnam "GraphA") (Pnam "FuncA") [] [])
```

then, a schedule permission for this name only authorises the holder to schedule the node to a specific domain, `DomainA`, while the original permission allowed the holder to schedule the node to any domain. △

This relationship is required as `Name` ordering and `Permission` orderings between the names of nodes are "opposite" to one another. As we have seen, names of nodes grow more precise as they increase in size. In contrast, permissions grow more restrictive as node names increase in size. This is similar to how the KeyNote trust management system [29] operates. In KeyNote, if a attribute is not specified, then no restriction is placed on the value of that attribute. With WebCom permissions, if any part of the name is `Empty`, then the permission implicitly authorises *any* value in that part of the name.

We argue that the ordering shown in Figure 7.2 meets this requirement. This ordering upholds this requirement as one permission is considered greater than another when the node name contained in the first permission is *less* than the node name in the second permission. Thus permissions for a specific node are of a lower order than those for a less specific name for that node.

Example 7.5 When the `BuySeat` node has executed and a seat purchased on Aerlingus' EI220 to Paris, the name of the `RentCar` node changes to reflect this result. The `Exe` permission for this node on the UCC compute cluster is as follows:

```
(Exe (Snam
       (Lnam ["Grid Ireland","Munster","UCC","Compute Cluster"]))
       (Pnam "eBookers")
       (Lnam ["eBookers","RentCar"])
       [(Lnam ["Aerlingus","EI220","Paris","TravelAgentAp","BuySeat"])]
       [(Lnam ["eBookers","Print"])]
     )
)
```

In order for permission monotonicity to hold, this permission must be less than (or equal to) the WVM permission shown in Example 7.1 in the ordering of permissions. As this permission is in fact a *lower* permission, in the respect that it refers to a more specific node, this property holds. △

7.1.3 Binding Permissions to Entities

Execute and schedule permissions are disjoint sets, and are thus not comparable. A WVM may hold either execute or schedule or both permissions for nodes of any given name. Every WVM and node have associated schedule and execute permissions.

```
schedule :: WVM -> [Permission]
execute  :: WVM -> [Permission]
```

Given `w:WVM`, then `schedule(w)` returns the scheduling permissions associated with the WVM `w` and `execute(w)` returns the execute permissions that WVM `w` holds.

Given a node `n:Node`, then `(Sch n)` represents the *lowest* permission that must be held by a WVM in order to schedule the node `n`; `(Exe n)` represents the *lowest* permission that must be held by a WVM to execute the node `n`. The node permission shown in Example 7.1 shows a permission for a `RentCar` node. However, the *lowest* possible permission for any node is specific to the inputs and outputs of that node. For example, the lowest permission for a node with input value “5” is different to the lowest permission for a node with input value “6”, as each permission would specify the input value.

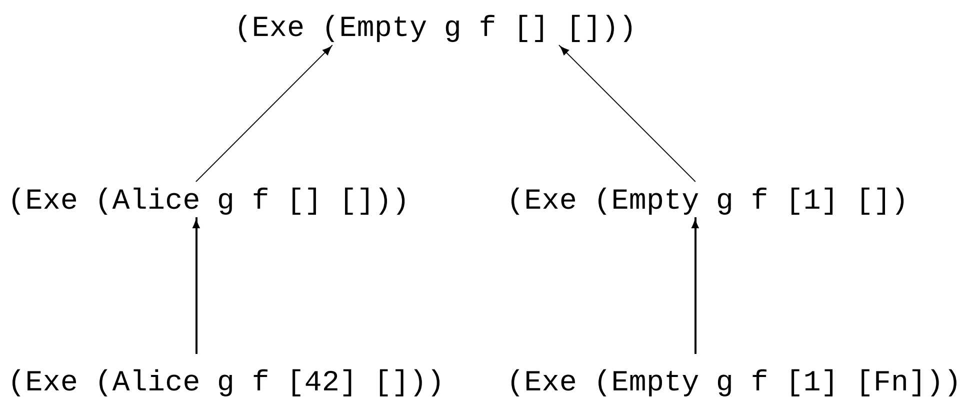


Figure 7.3: Sample ordering of execution permissions

Example 7.6 Figure 7.3 shows an example of how permissions are ordered for a graph `g` and Function `f`[1]. In this example, the permission that is the least precise, that is `(Exe (Empty g f [] []))`, is highest ordered permission. Lower permissions are ordered according to the details that are present in these permissions. For example, `(Exe (Alice g f [] []))` is a higher order permission than `(Exe (Alice g f [42] []))`, as the latter has additional information, that is an input value of “42”. △

We can define what is meant by a secure WebCom system as one where every schedule and execute operation is authorised.

Definition 7.1 *Secure Execution.* A WVM `w` should hold the permission to execute a node `n` that is scheduled to it, that is `mayExecute(w,n)`, where,

```
mayExecute :: WVM -> Node -> Bool
mayExecute w n = any (\p -> (Exe n) <= p) (execute w)
```

`mayExecute(w,n)` defines that in order for a WVM, `w`, to execute a node, `n`, the WVM must hold an execute permission for that node. ♢

Definition 7.2 *Secure Schedule.* When a node `n` is scheduled to a child WVM for execution, the scheduling WVM (in this case, the parent WVM that executes the graph that contains `n`) must hold the permission to schedule that node, that is, `maySchedule(w,n)`, where,

[1]The permissions shown use a simplified syntax for display reasons.

```
maySchedule :: WVM -> Node -> Bool
maySchedule(w,n) = any (\p -> (Sch n) <= p) (schedule w)
```

`maySchedule(w,n)` defines that in order for a WVM, `w`, to schedule a node, `n`, the WVM must hold an schedule permission for that node. ♢

Example 7.7 Consider a `RentCar` node with name:

```
(Snam
  (Lnam ["Grid Ireland","Munster","UCC","Compute Cluster"]))
  (Snam (Lnam ["eBookers","Alice"]) (Pnam "TravelAgentAp") (Empty) [] [])
  (Lnam ["eBookers","RentCar"])
  [(Lnam ["eBookers","BuySeat"])]
  [(Lnam ["eBookers","Print"])]
)
```

We can use the `mayExecute` or `maySchedule` functions to determine whether this node is authorised to be executed or scheduled, respectively. As the name of the WVM forms part of the name of the node, we use this information to determine whether the action is authorised. Thus, in this instance, the `RentCar` node is to be scheduled to the UCC Compute Cluster. This action is authorised only when there exists a relevant execute permission held by the WVM with (element of `execute(w)`) name:

```
(Lnam ["Grid Ireland","Munster","UCC","Compute Cluster"])
```

and a relevant schedule permission held by the WVM (element of `schedule(w)`) with name:

```
(Lnam ["eBookers","Alice"])
```

△

It is possible for a WVM to hold a permission that they cannot use. For example, consider the permission

```
(Sch Lnam ["eBookers","Alice"])
```

If `Bob` held this permission, then he would not be able to use it to schedule any nodes. However, `Bob` could delegate this permission to `Alice`, who would be able to use it.

7.1.4 Implementing the Security Model in WebCom

We can interpret this model in WebCom in terms of attribute credentials using a trust management system. Each WVM holds cryptographic credentials that represent their schedule and execute permissions for various nodes. As these credentials are crytographically signed, they are computationally infeasible to forge, and so provide a secure means to encode permissions.

```
(cert
 (issuer (hash sha1 |dsEFA73sahfdDF3784JDFjfsFsd=|))
 (subject (ref: UCC (ref: CSDEPT (hash sha1 |dasdk...|))))
 (propagate)
 (
  (tag
   (execute
    (WebComName
     (domain (ref: Grid_Ireland (ref: Munster (ref: UCC
             (ref: Compute_Cluster)))))
    )
   )
  )
 )
 (not-before "2005-11-31_17:00:00")
 (not-after "2006-11-31_16:59:59")
)
```

Figure 7.4: SPKI Credential authorising `Compute_Cluster` to execute any node.

Example 7.8 Figure 7.4 shows an execute permission, encoded as a SPKI credential, for a WVM that authorises the WVM to execute every possible type of node. △

Permissions can be directly encoded as trust management credentials, as shown in Example 7.8. The trust management compliance check directly corresponds to performing `mayExecute` and `maySchedule` checks on nodes that are to be executed and scheduled respectively. For an individual WVM to be considered secure, every node that the WVM receives for execution from a parent WVM, and every node that the WVM schedules, must be authorised. The security compliance check required in order for a node to be scheduled and executed can be considered in two parts:

- the WVM scheduling the node must perform a check to ensure that the WVM that the node is being scheduled to is authorised to execute it (Definition 7.1).
- the WVM executing the node must perform a check to ensure that the WVM that scheduled the node was authorised to do so (Definition 7.2);

In both these cases, if either, or both, of these condition fails, then the action is rejected.

A critical requirement for a distributed implementation of Secure WebCom is that a WVM should only make mediation decisions about other WVMs and should never be relied upon to mediate about themselves. A WVM cannot be trusted to decide whether it is authorised to schedule (to itself) or execute a node that it holds.

These mediations can be performed using a trust management system. When permissions are cast as trust management credentials, the WVM scheduling the node provides the relevant scheduling credentials to the child WVM. The child WVM can then perform the trust management mediation. This is equivalent to the child WVM performing a `maySchedule` check. Similarly, the child WVM provides the parent WVM with execution credentials, so that the parent can use the trust management system to ensure that the child is authorised to execute a node. This is equivalent to the parent performing a `mayExecute` check. These separate, but linked checks are necessary in order for the entire WebCom system to be considered secure. In any implementation, every node that is scheduled and executed must have these two checks applied to them.

Each WVM maintains a local policy, consisting of a list of permissions held by both the local and remote WVMs. These permissions are used to determine whether security critical actions are authorised. In terms of a trust management system, this policy consists of a list of trusted policy credentials. The global policy of the system is made up of all the local policies.

Example 7.9 Considering the name of the `RentCar` node from Example 7.7, in order for the WebCom system to be secure, a `mayExecute` check must be performed on the (parent) WVM with the name:

```
(Lnam ["eBookers","Alice"])
```

regarding the (child) WVM with the name:

```
(Lnam ["Grid Ireland","Munster","UCC","Compute Cluster"])
```

and a `maySchedule` check must be performed on the (child) WVM with the name :

```
(Lnam ["Grid Ireland","Munster","UCC","Compute Cluster"])
```

regarding the (parent) WVM with the name:

```
(Lnam ["eBookers","Alice"])
```

The parent WVM must hold at least the permission:

```
(Sch (Snam
      (Lnam ["Grid Ireland","Munster","UCC","Compute Cluster"]))
      (Snam (Lnam ["eBookers","Alice"]) (Pnam "TravelAgentAp") (Empty) [] [])
      (Lnam ["eBookers","RentCar"])
      [(Lnam ["eBookers","BuySeat"])]
      [(Lnam ["eBookers","Print"])]
     )
)
```

The child WVM also must hold at least the associated execute permission. Only when both of these WVMs hold these permissions will the node be scheduled and executed by a secure WebCom system. △

```
(cert
 (issuer (hash sha1 |dsEFA73sahfdDF3784JDFjfsFsd=|))
 (subject (ref: UCC (ref: CSDEPT (hash sha1 |dasdk...|))))
 (propagate)
 (
  (tag
   (execute
    (WebComName
     (domain (ref: Grid_Ireland (ref: Munster
             (ref: UCC Compute_Cluster))))
     (graph ebookers)
     (function (ref: (ebookers RentCar)))
     (inputs
      (input (ref: Aerlingus (ref: EI220
             (ref: Paris (ref: TravelAgentAp BuySeat))))))
     (outputs (output (ref: ebookers Print)))
    )
   )
  )
 )
 (not-before "2005-11-31_17:00:00")
 (not-after "2006-11-31_16:59:59")
)
```

Figure 7.5: A SDSI/SPKI credential authorising a `RentCar` node.

Example 7.10 Figure 7.5 is an sample SPKI credential for the `RentCar` node discussed in Example 7.5. In this case, the licensee is authorised to execute a `RentCar` node. The parent WVM uses such credentials to determine whether a child WVM is authorised to execute a specific node. △

Proposition 1 The implementation of the security model in terms of credentials is valid. ○

7.2 Sample Security Policies for WebCom

We now describe some specific application policies that can be enforced using the WebCom security architecture. Section 7.2.1 discusses a share trading workflow application that uses names to ensure that only authorised users receive secured components. We then examine a high watermark policy style policy in Section 7.2.2. This policy uses historical information stored within node names to ensure that executions do not travel to WVMs that are of lower classification than the application. Finally, Section 7.2.3 examines *push* authorisation using WebCom names. Push authorisation ensures that future requirements are taken into account when current authorisation decisions are being made.

7.2.1 ShareTrader

We can configure Secure WebCom to support specific application security requirements. In a workflow, application components may interact with specific users. Condensed graphs are ideally suited to providing the necessary sequencing constraints. The WebCom security architecture can then be configured to ensure that the correct users (domains) execute specific nodes.

Example 7.11 A workflow based WebCom application is shown in Figure 7.6. This application has three application components: `PriceDeal` (Figure 7.7), `Verify` (Figure 7.9) and `CaptureDeal` (Figure 7.8). The application is executed by customers who wish to purchase shares. The user passes in the share symbol as a parameter to the application. A stock trader will check the price of the stock and suggest a purchase using the `PriceDeal` component.

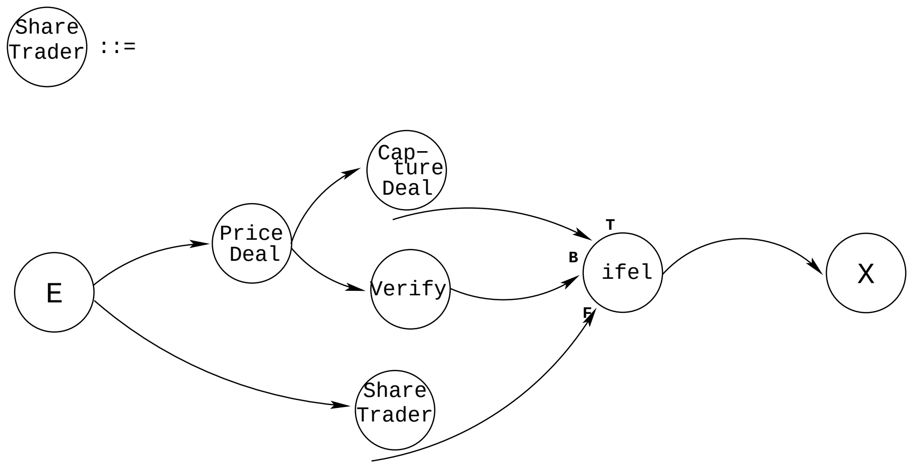
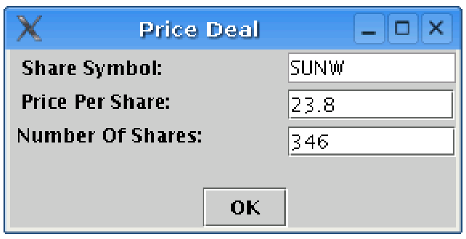


Figure 7.6: The Share Trader Application

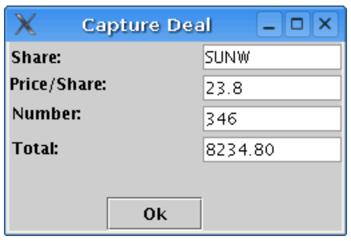

Price Deal

Share Symbol:	SUNW
Price Per Share:	23.8
Number Of Shares:	346

OK

Figure 7.7: The `PriceDeal` Component

The customer verifies the transaction using the `Verify` component.


Capture Deal

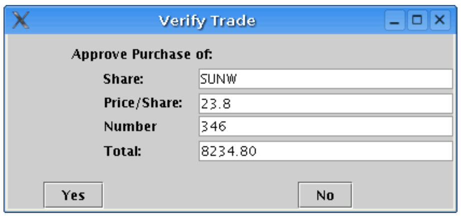
Share:	SUNW
Price/Share:	23.8
Number:	346
Total:	8234.80

Ok

Figure 7.8: The `CaptureDeal` Component

If they approve of the transaction, then the stock is purchased for the agreed price using the

`CaptureDeal` component. If the customer does not approve, then the stockbroker will get a new quote. The `ShareTrader` graph is defined recursively.



Approve Purchase of:

Share:	SUNW
Price/Share:	23.8
Number	346
Total:	8234.80

Figure 7.9: The `Verify` Component

When creating security policies for this application, we must represent the requirements of the stakeholders. For example, the customer's security policy for this application must consider which stockbrokers they trust to make purchases for them. The stockbrokers will only accept requests from trusted children. Both parties will only allow the components to execute in trusted domains.

Each user of the system stores their own credentials, and provides them as required to WebCom, in order to prove their authorisation. Nodes are scheduled only to authorised users. Using application specific naming, very specific policies can be enforced by the security architecture. For example, the stock-brokerage may allow a junior broker to capture sales up to a certain value, say €200. Once a sale over this value is requested, a trading manager must capture the sale. These conditions are specified in the trust management policy, and are aided by the naming system. When a `CaptureDeal` node is scheduled for execution, the name of the node must contain information about the input to the node.

In order to achieve this, when the node name is generated, it must contain the details of the stock trade from the inputs to the `CaptureDeal` node. An application specific `NameGenerator` class, as shown in Appendix B, is implemented in Java and is used to ensure this takes place. Any required reduction rules, again implemented in Java and shown in Appendix B, are then applied to the name to convert it into reduced form. Such a node name is shown in Figure 7.10.

The share trading policy is shown in Figure 7.11. This KeyNote policy authorises the senior trader to perform any operations on `CaptureDeal` nodes in the `ShareTrader` graph. The senior trader then delegates part of this authority to the junior trader by signing the credential shown in Figure 7.12. This credential limits the junior trader to executing `CaptureDeal` nodes of value less than €200.

△

```
(Snam (Empty) (Pnam "ShareTrader") (Pnam "CaptureDeal")
                                   [(Lnam ["stock","SUNW","210"])] [] )
```

Figure 7.10: The name of a `CaptureDeal` node before scheduling

```
KeyNote-Version: 2
Comment: ShareTrader Policy File for Sales Team
Local-Constants: seniortrader =
            "rsa-base64:MIGfMAIb3DQEBAQUAA4GNADCBiQKBg\
            skpav8kfrw7OKnNgFMHDuVc69wIDAQAB"
Authorizer: "POLICY"
Licensees: seniortrader
Conditions: App_Domain == "WebCom" &&
            Graph == "ShareTrader" &&
            Function == "CaptureDeal";
```

Figure 7.11: The ShareTrader policy authorising the Senior Trader.

```
KeyNote-Version: 2
Comment: ShareTrader Junior Trader Credential
Local-Constants: seniortrader =
            "rsa-base64:MIGfMAIb3DQEBAQUAA4GNADCBiQKBg\
            skpav8kfrw7OKnNgFMHDuVc69wIDAQAB"
                 juniortrader =
            "rsa-base64:MIGfMASDJASD123d3DDd932J3kk32f\
            yy123kJJ2304kbwqiMaAlwpo8rJKWEQT"
Authorizer: seniortrader
Licensees: juniortrader
Conditions: App_Domain == "WebCom" &&
            (Graph == "ShareTrader" &&
            Function == "CaptureDeal" &&
            Input < 200) &&
            operation == execute;
Signature: ...
```

Figure 7.12: The ShareTrader credential authorising the Junior Trader.

7.2.2 High Watermark style policy

Many traditional authorisation policies are based on the context in which computational components have been executed in the past. For example, Separation of Duties [135], Chinese Wall [40, 58, 106] and High Watermark [180] policies. With such history based policies, the access control state needed to make decisions must be gathered from all the distributed mechanisms that are involved in the computation in the past. For example, in the case of trust management based authorisation, this means that all relevant credentials issued must be gathered to accurately determine the access control history of a user. With condensed graphs, the context information is stored in the node names. Thus, in order to make an access control decision for a condensed graph, the relevant contextual information can be preserved using history-based reduction rules.

In a high watermark policy component levels (names) rise to reflect the classification of the data written to it. High watermark policies incorporate the concept of ordered classifications from multi-level security (MLS) policies. In the case of a condensed graph application, this is cast as a policy where a node may only execute on a resource of “equal” or “higher” classification. The classification of a node will depend on the path that the execution has taken to this point. In a distributed computation, such a policy may be informally expressed as: “once a computation is executed on a resource running at a certain security level, resources that are of a lower security level are never again used in the future execution of the computation".

We use node names to store security state in a decentralised manner during application execution. In Chapter 6, we described how the WebCom naming architecture has the ability to store information in the names of the nodes, and ensure, using appropriate reduction rules, that this information is stored in the names of all the nodes in a graph. The contextual detail required to make authorisation decisions is "pulled" from the nodes that have executed.

The reduction policies applied to the names must encode the highest security clearance of the resources that the execution has used to this point. In the case of a distributed computation, the resources that are used to execute components in the future depends on the resources used to date.

Example 7.12 Figure 7.13 shows a condensed graph that defines an implementation of a flight reservation application. This graph can be considered as an implementation of the `BuySeat` node from Figure 6.3. The airline's security policy includes a basic ordering of domains, shown in Fig-

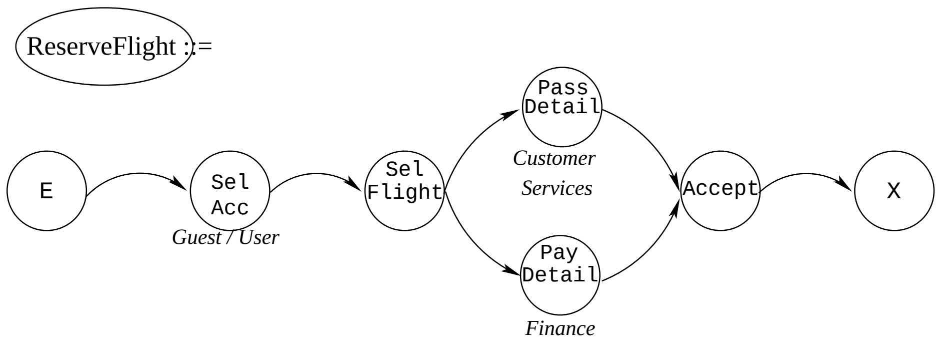


Figure 7.13: Reserving a Flight specified as a Condensed Graph.

ure 7.15. Under this ordering, *Finance* > *CustomerService* > *Guest*. For example, if a node executes on a resource that is classified *Finance*, then no subsequent node should execute on resources classified *CustomerService* or lower.

If this graph executes on the Airline's network, and in the domains shown, then the security policy requires that the `Accept` node should only be executed in the *Finance* domain. This can be achieved using reduction rules that maintain the high watermark within a node's name.

We use the high watermark reduction rule shown in Figure 7.14. This is the original input reduction rule from Figure 6.15 in Chapter 6. However, for specific applications, we need to redefine

```
saveDom :: Name -> Name
saveDom (Snam a b c [(Snam v w x y z)] e)
          | a <= v      =   (Snam v b c [(Snam v w x y z)] e)
          | otherwise   =   (Snam a b c [(Snam v w x y z)] e)
```

Figure 7.14: High watermark reduction rule.

the `Ord` relation from Chapter 6. Recall that permission ordering are opposite to name orderings.

```
    instance Ord Name where
      Empty                   <= x                      =  True
      (Snam a b c d e)        <= (Snam v w x y z)   =  (a <= v) && (b <= w) &&
                                                       (c <= x) && (d <= y) &&
                                                       (e <= z)
      (Pnam "Finance")        <= (Pnam "CustomerService") =  True
      (Pnam "Finance")        <= (Pnam "Guest")         =  True
      (Pnam "CustomerService") <= (Pnam "Guest")         =  True
      (Pnam p)                <= (Pnam q)               =  p == q
      (Lnam l)                <= (Lnam m)               =  (l == m)
      x                       <= y                      =  x == y
```

Figure 7.15: Company Domain orderings.

Permission orderings (shown below), use these name orderings.

```
    instance Ord Permission where
      Exe n    <= Exe m         =  m <= n
      Sch n    <= Sch m         =  m <= n
```

In this case, we use some application specific rules to support the required domain ordering. When the `saveDom` function reduces names, the name ordering from Figure 7.15 is used.

```
(Snam
  (Pnam "CustomerService")
  (Pnam "BuySeat")
  (Pnam "Accept")
  (Snam  [(Snam (Pnam "CustomerService") (Empty)
                (Pnam "PassDetail") [][])
          (Snam (Pnam "Finance") (Empty)
                (Pnam "PayDetail")[][])
         ])
  [(Pnam "X")])
```

(a) Before Reduction

```
(Snam (Pnam "Finance") (Empty) (Pnam "Accept") [] [])
```

(b) After Reduction

Figure 7.16: The name of the `Accept` node: (a) before and (b) after reduction.

In the names shown in Figure 7.16, prior to the application of the reduction rule, the node `Accept` is considered to be permitted to execute on a resource classified `CustomerService`. However, as one of the previously executed nodes, `PayDetail`, was executed on a resource of classification `Finance`, the reduction rule modifies the name of the node and changes the domain tuple to `Finance`.

This can be used to enforce a high water mark authorisation policy, once any node in the computation reaches a higher security level, all subsequent nodes must execute on resources that hold permissions for nodes to be executed at least at that security level. The parent WVM must apply these reduction rules to any node that is to be scheduled, and then perform a `mayExecute` check, using the high watermark policy, to determine whether child WVMs hold sufficient permissions to execute subsequent nodes.

We could also make decisions based on the type of customer using the service. If a known customer, whose details are retained by the airline, logs into the site, then the name of the node could be set to reflect this. When the payment details are required, that node would be directed to specific resources that hold saved customer financial details. $\triangle$

7.2.3 Pull and Push Access Control

The history based access control strategy works well in the case of a simple ordering of, for example, domains. With history based access control, we make access control decisions regarding the contexts that a node has gone though. We consider this strategy to be a *'pull'* authorisation strategy in that contexts are pulled from the relevant names. Authorisation decisions often have consequences that alter the possible future authorisation decisions that may occur. Consider the case where we have a mutually exclusive ordering such as with a separation of duties policy [135] or Chinese wall policies [40]. In a Chinese wall policy, once a computation executes on a particular company's resource, it should never be allowed execute on resources belonging to a competitor. With a Condensed Graph application, it is possible that multiple nodes execute in parallel, and the results from these parallel executions are integrated in the future. When we use a pull strategy to enforce a separation of concerns policy, we can encounter deadlock, where the computation can never finish due to a policy conflict. We could address this concern trivially, with a policy that dictates all computations must execute in one domain. However, such a policy will force the computation to be assigned a priori to a specific domain. This approach limits the flexibility of the computation, Instead we introduce the concept of *'push'* authorisation to address this issue.

Traditionally, when such problems are addressed dynamically, they use synchronisation between the components [19]. However, such synchronisation is often undesirable and requires extra infrastructural support. Ideally, we want to be able to identify potential conflicts and address them within the authorisation policy. Instead of pulling the information required to make a decision from the source, we instead push this information from the source to the points where it will be needed. We refer to these as "push", or future based, authorisation policies.

We can use push authorisation to force a computation to execute in the future on specific resources based on the authorisations that the computation has received in the past and on the potential conflicts that must be avoided in the future. This allows, for example, the enforcement of dynamic separation of duty policies, without an external synchronisation infrastructure. The details required to identify potential conflicts are stored in the computational context of the nodes. In WebCom, push authorisation policies are then written in terms of trust management credentials and are enforced using the existing security architecture. No additional architecture is required to support these policies, the only change is in how the names of the node are created.

Push reduction can be defined in a similar way to history based reduction. The subtle difference between them exists in the part of the execution context that is expanded, inputs for pull reduction, and outputs for push reduction.

Definition 7.3 *Push Reduction:.* A node n has a name (execution context)

```
(Snam d g f i o)
```

whose attributes provide the names of execution contexts for domain (`d`), application graph (`g`), function (`f`), inputs (`i`) and outputs (`o`) , respectively. The push reduction of the name of node n is

```
tupleElimination(Snam d g f i (map tupleReduction o]))
```

where `tupleElimination` and `tupleReduction` represent the application of tuple elimination and application reduction rules, respectively. In contrast to the history-based reduction described in Chapter 6, the `map` function applies the `tupleReduction` function to each member of the list of outputs, rather than the list of inputs. $\diamondsuit$

To make a 'push' authorisation decision about an execution context, first we expand the execution context to examine the destination contexts that will be used. Next, application specific tuple reduction rules, `tupleReduction` are used to extract the relevant details. Tuple elimination rules, `tupleElimination`, are then used, when required. Finally the reduced name is used with the authorisation mechanism.

Example 7.13 The rental company's separation of duties policy states that: "Sales and Finance Data should not be processed on the same resource". Once the node `PayDetail`, from Figure 7.17, has been scheduled to a computational context, the name of `CarModel` node must be updated so that when it is to be scheduled, it is sent to a non conflicting domain. This ensures that when the results from both of these nodes reaches the `Accept` node, it can be executed in accordance with the separation of duties policy.

Updating the `Accept` node could be achieved with pull authorisation; however updating the `CarModel` node would require communication between nodes that can execute in parallel to ensure synchronisation. Instead, when the result of the `CustDetail` node is to be integrated into the computation, the details of the computational contexts that it will be sent to is pushed to the `CarModel` node.

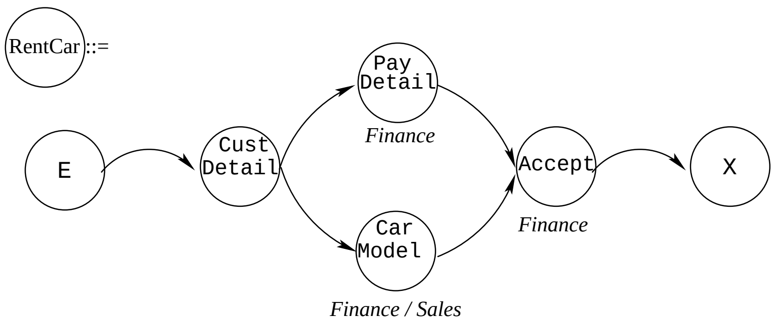


Figure 7.17: Reserving a Car specified as a Condensed Graph.

```
(Snam (Empty) (Pnam "RentCar") (Pnam "CarModel")
  [(Snam (Pnam "CustomerService") (Empty) (Pnam "CustDetail") [] [])]
  [(Snam (Empty) (Empty) (Pnam "Accept") [] [])]
)
```

Figure 7.18: Name of the `CarModel` node before the Push Authorisation decision.

Figure 7.18 shows a representation of the node's name before a push authorisation decision has taken place. In particular, the node's domain has not been specified at this point. When the `PayDetail` node is scheduled to execute, the authorisation policy ensures it is sent to the *Finance* domain. Thus, all subsequent nodes must adhere to the separation of duties policies. Once this requirement is apparent, the name of the `CarModel` node is modified to contain a specific domain, *Finance*. This push action ensures that no policy conflicts will occur.

```
pushDom  ::Name -> Name
pushDom (Snam a b c [(Snam q r s t u),(Snam v w x y z)] e)
        | v <= q  =  (Snam a b c [(Snam q r s t u),(Snam q w x y z)] e)
        | q <= v  =  (Snam a b c [(Snam v r s t u),(Snam v w x y z)] e)
```

Figure 7.19: Push reduction rule, for a node with two destination domains, q and v.

This push action is implemented in the form of a reduction rule. Figure 7.19 shows such a rule where a node's result may be sent to conflicting domains `v` and `q` . The `pushDom` reduction rule defines that where the result of this node could execute in domains `v`and `q`, then they should both be forced to be executed in the higher order domain. These rules are applied to the names of the nodes that will execute in the future and the authorisation mechanism ensures that they are scheduled to the correct domains. Figure 7.20 shows the node name after the push. △

Push authorisation allows a more dynamic control over ongoing computations and provides support for pushing computations to specific resources using the security policy. This allows the implementation of distributed separation of duty policies, without requiring ongoing synchronisation or communication between atomic nodes. Changing the names of the nodes requires no

```
(Snam (Pnam "Finance") (Pnam "CarRental") (Pnam "CarModel")
  [(Snam (Pnam "CustomerService") (Empty) (Pnam "CustDetail") [] [])]
  [(Snam (Empty) (Empty) (Pnam "Accept") [] [])]
)
```

Figure 7.20: Name of the *SelectModel* node after the authorisation decision.

changes to the existing pull-based security architecture. The same security policies are used to provide authorisation. The only change is in how the relevant information is provided to the protection mechanism.

With this push mechanism, we can provide the communication within the existing framework. However, providing a push authorisation model limits the possible contexts that a computation may execute in the future. The fact that decisions are pushed before execution means that some potential future information cannot be used when making a decision. We argue that the advantages that simplification of the architecture bring, outweigh this potential downside.

Push authorisation can also suffer from deadlock. If a node requires that it execute on a specific resource, and a push authorisation decision has excluded that resource, the computation will never complete. As with a pull deadlock condition, this would require that the computation is "rolled back" and nodes re-executed.

Push authorisations can be modelled within the pull architecture, however, this requires a centralised synchronisation mechanism. When a pull reduction takes place, the node names on parallel paths must be synchronised. Such a centralised mechanism would have to exist outside of the existing trust architecture.

7.3 Secure WebCom Software Architecture

The reference implementation of WebCom provides a software architecture that implements the security model described in Section 7.1. Currently the implementation is written in Java, using a Java s-expression library [5] to support the naming model. The reduction rules are also currently specified in Java. This section outlines the software architecture of the security systems in WebCom.

The security manager module is used in several situations to ensure that the WVM's local security policy is upheld. These situations can be described in terms of scheduling and executing actions performed by parent WVMs (WVM$_P$) and child WVMs (WVM$_C$), as shown in Figure 7.21, and enumerated below.

1. When a node is selected for execution by WebCom, it will be dispatched to the scheduler in order to find a suitable child WVM (WVM$_C$). The child is selected by the load balancing and security modules based on their policies. This authorisation can be represented as: WVM$_C$ is authorised to execute node n when `mayExecute(`WVM_C, n`)` holds. While the load

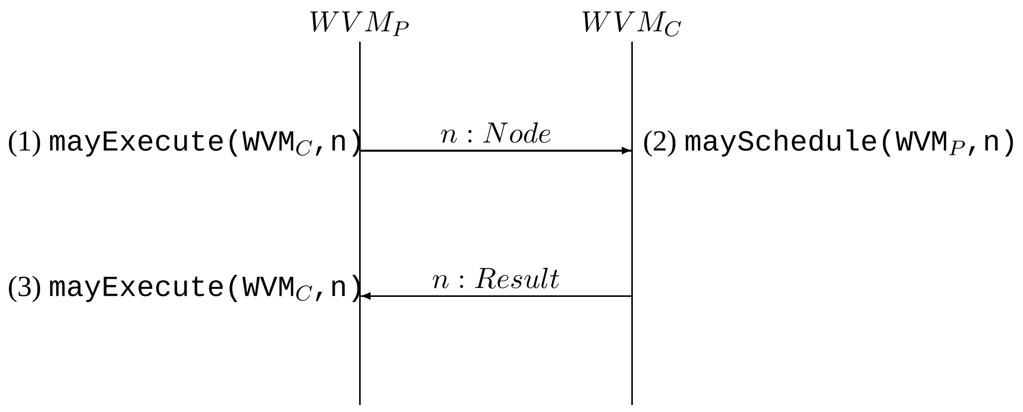

WVM_P
WVM_C
(1) mayExecute(WVM$_C$,n)
$n : Node$
(2) maySchedule(WVM$_P$,n)
(3) mayExecute(WVM$_C$,n)
$n : Result$

Figure 7.21: Authorisation Steps in Secure WebCom

balancer works with the security manager to pick suitable targets, the security manager has priority. That is, the load balancer will select its preferred option from the list of authorised children. The WVMs on this list are determined by the security manager. If no authorised children are found, then the node is rejected by the security manager for later rescheduling. Once a child is selected, the node is dispatched to that child.

2. When children receive nodes for execution, the child WVM (WVM$_C$) will first consult its security manager to ensure that the parent WVM (WVM$_P$) is authorised to schedule that Node. This authorisation can be represented as: WVM$_C$ executes node n from WVM$_P$ when `maySchedule(`WVM_P, n`)` holds. If the WVM is not authorised according to the child's local security policy, then the node is rejected and the parent informed.

3. When a willing child does execute a node and the result is returned to the parent, the result will be sent to the parent's security manager for verification. This authorisation can be represented as: WVM$_C$ is authorised to execute node n when `mayExecute(`WVM_C, n`)` holds. If the result of an execution does not correspond to the security policy, then it will be rejected and the node will be rescheduled. This authorisation check represents the reality that the `mayExecute(`WVM_C, n`)` operates on an approximation of the name of n, as not all of the information is available until the node has executed. An authorisation policy can require that results of the execution of a node fall in specific ranges. Such policies cannot be enforced before execution. For example, a clerk in a share trading firm may only be authorised to capture deals worth less than €100. If such a clerk attempts to capture a deal greater than this, then the `mayExecute` check on the result will discover it, and inform WebCom so that the result can be discarded.

These security mediations correspond exactly to the security model described in Section 7.1. However, the second `mayExecute` step is not explicitly specified in the model, as the model does not directly consider the dynamic nature of node execution. When a node executes, its name changes. Therefore, a child WVM may not have been authorised to execute the node.

7.3.1 webcom.core.security.SecurityManagerModule

The SecurityManagerModule provides the ability to control the execution of nodes and graphs in the WebCom system. Providing a different type of access control entails creating a custom security manager module. The security manager can be implemented using different enforcement mechanisms. The primary difference is in how the access control policy is specified. However, the WebCom access control model is always upheld, as every implementation of the security manager must provide a standard interface, as detailed below.

```
public boolean isAuthorised(Descriptor client, Instruction Instr);
public boolean isAuthorised(Descriptor client, Result Rslt);
public Object getAuthorisedClient(Vector clients, Instruction Instr);
public Vector getAuthorisedList(Vector clients, Instruction Instr);
```

This interface provides the ability to ask four basic access control questions:

1. `sm.isAuthorised(c,i)`: is a specific child (`c`) authorised to execute a particular node (instruction) (`i`?);

2. `sm.isAuthorised(c,r)`: is the result (`r`), that a specific child (`c`) is returning authorised to be included into the computation?;

3. `sm.getAuthorisedClient(vc,i)`: can a child be found from a list of children (`vc`) that is authorised to execute a particular instruction (`i`)?;

4. `sm.getAuthorisedList(vc,i)`:can a list of children (`vc`) be found that are all authorised to execute a particular instruction (`i`)?;

These methods target instructions (processed nodes) and the results of the execution of a node. These methods provide the implementation of both the `mayExecute` and `maySchedule` functions; `mayExecute`, when the methods are called on the parent WVM and `maySchedule`, when the methods are called by the child WVM that is to execute the node.

Security managers can be used to perform specific tasks. For example, the security manager shown in Figures 7.22 and 7.23 makes access control decisions using the KeyNote trust management system. Figure 7.22 shows how the public key is retrieved from the secure communications manager (lines 21–39). This public key is then sent with the name of the node to the `check` function (line 40), shown in Figure 7.23, which performs the actual trust management check. In this function, the tuples are extracted and used as attributes for the trust management query (lines 3–10). Next policy and user credentials are loaded into the trust management engine (25–30). Finally, the trust management engine attempts to find a link between the supplied public key and the local policy (line 32).

```
public boolean isAuthorised(Descriptor cd, Instruction I)
{
  this.backplane = getBackplane();
  if ( (currDomain == null) &&
     (getBackplane().getConMan() instanceof SecureConnectionManagerModule)){
    SSLSettings settings = ( (SecureConnectionManagerModule) getBackplane().
                            getConMan()).getSSLSettings();
    KeyNote kn = new KeyNote();

    currDomain = (kn.getPublicKeyString(kn.getKeyPair(settings.getKeyStore(),
        settings.getAlias(), settings.getPassword()))).trim();
  }
  Node currNode = I.getSourceNode();
  WebComName nodename = (WebComName) currentNode.getName();

  if (nodename == null) {
   nodename = nGenerator.generateNameFromNode(currNode, redrule, currDomain);
  }

  nodename.reduceName(getReductionRule());
  PublicKey clientkey;
  if (cd == null) {
   if (getBackplane().getConMan() instanceof SecureConnectionManagerModule){
      SSLSettings sslset = ( (SecureConnectionManagerModule) getBackplane().
                            getConMan()).getSSLSettings();
      KeyNote kn = new KeyNote();
      kn.setKeyPair(sslset.getKeyStore(), sslset.getAlias(),
                    sslset.getPassword());
      clientkey = kn.getPublicKey();
   }
   else
      clientkey = null;
  }
  else if (cd instanceof InetDescriptor) {
    String hostname = ((InetDescriptor) cd).connectedTo().getHostName();
    clientkey = ((InetDescriptor) cd).getPublicKey();
  }
  else
    return false;
  return (check(nodename, clientkey));
}
```

Figure 7.22: The Trust Management based security manager for WebCom.

7.3.2 Trust Management Based Security Manager

In Section 7.1 we described WebCom's security model. In this section, we examine a specific implementation of the security manager module. There exists implementations of security managers using both the KeyNote [29] and SPKI/SDSI [56] trust management systems. Both these security managers use Java based implementations of the requisite trust management systems, *JKeyNote* [90] and *JSDSI* [5] respectively. However, the security manager could use any of the trust management systems described in Chapter 2.

The trust management security managers work in essentially the same manner. When a trust

```
private boolean check(SecureName instrname, PublicKey ClientKey)
{
  KeyNote kn = new KeyNote();
  String Domain = instrname.getDomain();
  String Graph = instrname.getGraph();
  String Function = instrname.getFunction();
  if (Domain != null)   vL.addStringVar("Domain", Domain);
  if (Graph != null)    vL.addStringVar("Graph", Graph);
  if (Function != null) vL.addStringVar("Function", Function);

  Vector inputs = instrname.getInputs();
  for (Iterator iter = inputs.iterator(); iter.hasNext(); ) {
    String input = (String) iter.next();
    vL.addStringVar("Input", input); }
  Vector dests = instrname.getDestinations();
  for (Iterator diter = dests.iterator(); diter.hasNext(); ) {
    String destination = (String) diter.next();
    vL.addStringVar("Destination", destination); }

  knf.addVariablesList(vL);
  knf.setComplianceValues("untrusted,trusted");
  KeyNoteNavigator nav = knf.getNavigator();

  try {
    for (Iterator polsiter = pols.iterator(); polsiter.hasNext(); ) {
       String pol = (String) polsiter.next();
       trustedParser.parse(pol); }
    for (Iterator iter = creds.iterator(); iter.hasNext(); ) {
       String cred = (String) iter.next();
       untrusted.parse(cred); }

    int res1 = nav.findAuthorizer(ClientKey);
    if (res1 > 0) {
       RGLog.logFine("TMSecurityManager: Client is authorised to access "
                              + instrname);
       return true;}
    else {
       RGLog.logFine("TMSecurityManager: Client is not authorised to access "
                       + instrname);
       return false; }
  }
  catch (Exception e) {
    RGLog.logSevere("TMSecurityManager: Exception caught: " + e.toString());
    return false; }
}
```

Figure 7.23: Trust Management `Check` function used by the Security Manager.

management decision is required, the reduced name of the node is acquired from the naming manager module. This name is then used as part of the query conditions to the trust management system. Both the generation of node names, and how these names are reduced depend on the configuration of the naming manager module. Changing the reduction rules used or the name generator alter the conditions of access control decisions.

The trust management security manager extracts the information from the name provided to perform the trust management check. How this is achieved depends on the trust management system in use. For the KeyNote based system, each tuple is individually represented in the *condition* field. This allows contextual information to be separately assessed. In the SPKI/SDSI based trust management system, the s-expression is used in its entirety as part of the *tag* field. An example SPKI/SDSI credential can be seen in Figure 7.5.

7.4 Secure Authentication between WebCom Virtual Machines

WebCom communication is managed by the connection manager. Securing these communication channels is accomplished through the implementation of the connection manager module. WebCom can only securely schedule nodes when it can identify its children. This entails providing entity authentication for WebCom.

Identities in the communication manager are linked to the identities used by the security manager. For example, when selecting a child WVM to execute a job, the parent WVM must be able to associate the authorised entity with a physical IP address. Public keys are typically used to provide this link. However, any authentication token could be potentially used. For example, WebCom names can be used to provide a more contexual rich reference to a WVM.

In Chapter 2, we examined some authentication technologies, such as Kerberos [138] and the SSL [92] protocols that provide entity authentication. Using such authentication protocols in a secure connection manager allows WVMs to authenticate each other. The reference implementation of WebCom uses SSL to provide two-way authentication between WVMs. WVMs are identified by the security manager using their public keys. For example, whenever the trust management based security manager selects a child for node scheduling, the public key of that child is used as the child's identifier.

7.4.1 webcom.core.conman.SecureConnectionManager

The secure connection manager ensures that communication between one WVM and another is secure. It uses an implementation of a security protocol, for example SSL [92], to provide authentic and crytographically secure communication links. The default SecureConnectionManager class extends the standard CommunicationManagerModule class to use SSL sockets. However, in most cases, the inherited methods from the standard IPv4-based WebCom connection manager are used. The methods that are overridden provide the implementation of SSL using Sun Microsystems' Java Secure Sockets Extension (JSSE).

```
public boolean connectTo(InetAddress address)
public boolean connectTo(InetAddress address, int port)
public void setPort(int cpPort)
public void processMessage(Message msg)
```

The `scmm.connectTo(addr)` and `scmm.connectTo(addr,p)` connect to a SSL secured socket at the provided address (using the port number, when provided). These methods set up a secure connection between the local and remote WVMs. The `scmm.setPort(p)` method specifies the port number that the SSL server socket on the local WVM should be listening on. This port is normally specified in the module properties file and is loaded by the super class. This method allows the port number to be changed at runtime. Finally, the `scmm.processMessage(msg)` method adds some additional supported message types to the types supported by the superclass. These message types include the ability to query the public key provided by connecting WVMs for use by the SecurityManagerModule.

Providing secure communication also entails using a public key infrastructure to manage the keys used by WVMs. A public key infrastructure is necessary to properly determine validity of certificates, provide for certificate revocation, to provide a means to issue new and renewal certificates and to act as a repository for user certificates. There currently is no specific PKI implementation available for WebCom. Instead each WVM must have all the necessary certificates locally in order to support secure communication. The provision of a proper PKI implementation is the topic of future work. As work on WebCom is currently aimed towards using it as a Grid middleware [124], support of federated identity management [6, 117] is also an important topic of future work.

7.5 Discussion and Conclusions

In this chapter, we have introduced the access control model for WebCom. WebCom's access control model is designed to address specific threats to a distributed computing environment including illicit data and computation modification and unauthorised access to data or computations by principals. These threats are addressed by WebCom through the provision of an authorisation mechanism to ensure that computation execution and data access is only performed by authorised principals.

WebCom provides a distributed model to support access control decisions. This model uses the naming architecture described in Chapter 6 in determining the authorisation of security critical actions to be performed by WVMs. These actions consist of the scheduling and execution of condensed graph nodes.

WebCom is a closed system, in that a WVM has access to the name of every node in an application executed by that WVM. Therefore, a WVM can potentially act as a central repository of policy decisions. For example, we can use the parent WVM to ensure that an access permission granted to a particular child does not cause a conflict in a separation of duties, or Chinese wall-style policy.

Although the entire access control system can be implemented using the Haskell primitives described in this chapter, for both simplicity and efficiency reasons, the reference implementation is implemented using the Java programming language and existing access control logics, JKeyNote [90] and JSDSI [5].

In this chapter we have discussed a particular form of access control, using trust management to make security decisions: the design of WebCom supports the replacement of this system with a different enforcement mechanism. However, any implementation of the security architecture will follow the model proposed in Section 7.1 as any implementation of the security manager module *must* support the `mayExecute` and `maySchedule` checks, implemented using the `isAuthorised` function. For example, the security model could be implemented using access control lists (ACLs). In this case WebCom names would allow contextual information to be stored in the ACLs.

The naming approach to specifying security policies allows a separation of security checks from functional code. This provides a loosely-coupled architecture, where maintaining security policies does not require changing functional code. Other work has looked at codifying protection mechanisms as condensed graphs [62]. This work proposes implementing fragile, tenacious and emergent protection mechanisms using triple manager primitives that determine whether nodes should be executed in particular domains.

In the current implementation of WebCom entities are identified by their public key. A critical requirement in a distributed computing environment is the management and verification of these identities. This entails ensuring that there is a systematic way to determine the identity of both the users of the system and the resources used in the computation. Identity verification and management can be provided through the use of authorisation mechanisms.

In order to fully support identity in WebCom, some form of federated identity system is required, such as Liberty Alliance [6] or Microsoft's Passport [117]. Federation of identity allows each resource in the system to have an assembled identity that the entire distributed system can use to refer to that resource. For example, a user can be a manager in one domain and a clerk in another. Each domain refers to the user in the context they support. With a federated identity, the two domains share a context when they are referring to the user. The provision of a federated identity system is a topic of future work.

Another important topic of future research is the provision of a public key infrastructure (PKI) for WebCom. In its current form, WebCom uses an ad-hoc method to distribute both identity and authorisation certificates. It is envisioned that a PKI architecture should be constructed to provide a distribution mechanism for such certificates. PKIs also support concepts such as certificate revocation, which is also not currently addressed.

WebCom also includes a messaging architecture, where modules can send messages to other modules, both in the same or to other WVMs. The security model presented does not address this architecture. In practice, a security check is performed on all messages. Security policies can be written about messages, allowing users to control what messages are authorised to be acted upon. These policies could be specified in terms of the WVMs that a particular entity trusts. However, this only supports coarse-grained policies. More precise checks could specify the modules in a WVM that are trusted to send or receive specific messages. Such checks would require a naming system for messages. Extending the WebCom security model to represent the messaging architecture is a topic of future research.

While WebCom's messaging architecture provides several advantages, the potential for illicit behaviour using messages causes concern. It can be argued that this messaging system should not be allowed, or at least only allow a predefined set of vetted messages. However, it is possible to implement a general messaging system in a manner that lies within the security model. Messages could be implemented as condensed graph applications, and the existing security architecture used to enforce specific policies.

Chapter 8

Case Studies

We have seen, first in Chapter 5, and in more detail in Chapter 7, that WebCom is a modular and "pluggable" distributed execution system. WebCom's flexible design allows the creation of specific applications, such as the ShareTrader application described in Chapter 7. In this chapter, we will examine the advantages of Secure WebCom in more detail and describe some specific case studies that use the architecture.

WebCom's pluggable architecture allows the creation of different implementations of its core modules. Specifically, in this dissertation, we have examined the implementation of the security manager module. Changing security managers allows the enforcement of completely different security policies, while still adhering to the security model described in Chapter 7. This allows the creation of new applications that use Secure WebCom as their base.

This chapter describes applications of Secure WebCom. These applications are developed using the security architecture to support specific security requirements. In Section 8.1, we examine an early Secure WebCom prototype [65, 66] and describe how security policies from this prototype can be enforced in the current system. Section 8.2 describes a security manager for WebCom that incorporates micropayments, using the naming and security architectures to enforce pay-per-execute policies. This system uses trust management to support a credential-based payment system.

We analyse a grid administration extension for WebCom, called GridAdmin, in Section 8.3. This system uses WebCom to provide a secure administration system for computational clusters. Administration tools are provided using workflow applications that execute on demand on the machines being administrated. Applications for this system range from adding users to enforcing exclusive access to a cluster. The WebCom$_{DAC}$ case study is described in Section 8.4. WebCom$_{DAC}$ is a secure workflow system to control security policies on heterogeneous systems. We use WebCom$_{DAC}$ to view, modify and enforce a unified security policy across multiple systems and domains. Section 8.2, Section 8.3 and Section 8.4 are the result of collaborations with other members of the Centre for Unified Computing in UCC.

8.1 Classic Secure WebCom

WebCom was originally developed [65, 66] using a hybrid of the Java and C programming languages. The distributed system was implemented in Java, while the execution engine was written in C. A consequence of this design was that, as the internals of the graphs were not available for examination by the distribution system, node names were based purely on the function name of the node. The KeyNote trust management system was used to provide configurable access control. This access control mechanism was only able to make decisions based on the function of a node. However, with the development of the current WebCom architecture, this limitation no longer exists.

Security policies from the original WebCom prototype can be denoted in terms of reduction rules for the current WebCom system. This entails using tuple elimination rules to reduce a name to contain only the function tuple. This rule can be expressed in Haskell as follows:

```
retainFunc :: Name -> Name
retainFunc (Snam d g f i o) = f
```

The `retainFunc`[1] reduction rule extracts the function tuple from the provided name and discards the remaining entries. This allows traditional WebCom trust management policies to be enforced. A credential used with such a policy is shown in Figure 8.1.

```
KeyNote-Version: 2
Comment:
Local-Constants: Alice =
            "rsa-base64:MIGfMA0GCSqGSb3DQEAQUA4GNADCBiQ\
            Sr8xM9qBGuvbXG1eIZM6IcYTxQIDAQAB"
                 Bob =
            "rsa-base64:MIGfMA0GCSqGI3DQEBAQ4GNACBiQKBg\
            CKP9TXQE/zlC+poPrKHr/S7yHQIDAQAB"
Authorizer: Alice
Licensees: Bob
Conditions: App_Domain == "WebCom" &&
     (Function ==  "checkprime.isPrimeOp")
          && operation == "execute");
Signature: "sig-rsa-sha1-base64:H0YZV5yvCVNpLiVbyWWvclE\
            bmLbPdvYEzCY2nkVCX35feMasCPrOIVf+oluqjJGqY="
```

Figure 8.1: A function only KeyNote credential.

This credential defines that Alice delegates the execute right to nodes with the function name `checkprime.isPrime` to Bob. Providing such general credentials is, however, not necessarily advantageous. This credential allows Bob to execute `checkprime.isPrime` nodes in any possible graph, regardless of input or output or execution domain. This was a significant disadvantage of the original WebCom prototype. With the current implementation, using different reduction rules allows finer granularity policies to be enforced within the same access control system.

[1]We could alternatively use a combination of the tuple elimination rules described in Chapter 6. However, in this case it is simpler to use one reduction rule to retain a tuple, rather than applying four separate tuple elimination rules.

8.2 Micropayments

With the advent of computational grids, cluster owners are making their compute resources available to users outside of their own organisation. However, the provision of these resources is not without cost, both in maintenance and overheads. Owners may want to recoup these costs from their users. One solution is a micropayments system where users pay a charge based on the submitted compute job. We can develop such a system on top of Secure WebCom, using the naming and security infrastructures as the base.

Micropayments schemes are intended to support very low-value payments, in the order of €0.01, or less. Micropayment schemes [17, 35, 142] typically rely upon the notion of a digital coin that represents a fraction of a pre-agreed contract. Each coin is linked to this contract, and can only be redeemed when the coin is presented with the contract.

One means to support a micropayment system is through the use of one-way hash functions, such as MD5 [151] or SHA1 [91]. These hash functions are designed so that they are easy to compute, but computationally difficult to reverse. One way hash-based micropayment schemes [17, 37, 142] typically operate as follows: a payer (the principal making the payment) generates a fresh random seed s, and computes $h_n(s)$, where $h()$ is a cryptographic one-way hash function. If s is known only to the payer, then $([h_{n-1}(s), n-1, val] \ldots [h_1(s), 1, val])$ provides an ordered chain of micropayments, each one worth val. Initially, the payer provides a payee with $[h_n(s), n, val]$, which acts as a contract for $(n-1)$ micropayments.

As the seed is known only to the payer, and the hash function is computationally difficult to reverse, coins cannot be forged. Furthermore, as contracts are crytographically signed, presumably with strong cryptographic algorithms, they cannot be forged. Therefore, once a coin is presented for payment, the payer cannot repudiate the payment; only he could have given the payee the coin, and so the coin should be redeemed.

A payee (the principal receiving the payment) who has securely received i micropayments, $([h_{n-1}(s), n-1, val] \ldots [h_{n-i}(s), n-i, val])$, can use the hash function $h()$ to check their validity against the initial contract. Since $h()$ is a one-way hash function it is not feasible for the payee to forge or compute the next $(i+1)^{th}$ payment (before it is paid). Micropayments may be cashed in at any time; the payer keeps track of contracts issued, and any payments made, to guard against double spending.

Example 8.1 Consider an online business where a company, represented by public key K_{Comp} is providing a service to a customer, represented by key K_{Cust}. Figure 8.2 shows a simple micropayment protocol. The customer is trusted by the company to create payment contracts that the company will later redeem. K_{Comp} has received a contract from K_{Cust}. K_{Cust} sends K_{Comp} $n-i$ micropayments using this contract.

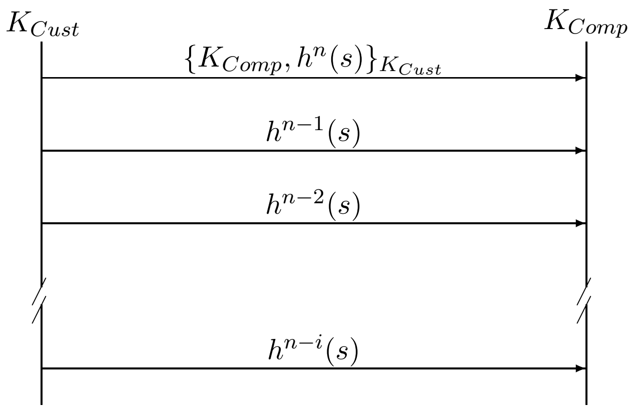


Figure 8.2: Making Micropayments

The company will later redeem these payments by presenting the contract along with the last coin received by the customer.

△

This approach to micropayments has been proposed and used in payment schemes proposed by [17, 37, 142]. For example, in [17], the payer threads digital coins (issued by a bank) through the hash chain such that each micropayment reveals an authentic digital coin that can be reimbursed by the original bank.

8.2.1 Micropayments in KeyNote

We use the KeyNote trust management system [29] to provide support for micropayment authorisation of services between public keys across networks. This architecture writes the micropayment contracts in terms of KeyNote credentials. The payer writes a contract credential for the payee, stating the terms of the contract, for example the value of each coin. This credential is used by the payee to ensure the validity of any coin received and is presented, along with the coins received, by the payee, when redeeming the contract.

Example 8.2 A company (public key K_{Comp}) expects payments for providing services X and Y. They trusts banks Bank1 or Bank2 to guarantee payments for the customers of their services up to a certain limit (€50.00). This is expressed by the KeyNote policy credential in Figure 8.3.

```
Authorizer: "POLICY"
Licensees:  "kBank1" || "kBank2"
Conditions: @Val * Num <= 50.0 &&
            Service == "X" || Service == "Y";
```

Figure 8.3: The Company's Policy

This policy credential defines the conditions under which the public keys K_{Bank1} and K_{Bank2} are trusted by the company. These conditions are defined in terms of attributes `Val` (micropayment value val), `Num` (number of micropayments n in contract) and Service. Figure 8.4 provides a micropayment contract credential that a customer (public key K_{Cust}) buys from Bank1. It is signed by the owner of public key K_{Bank1}, delegating authority over the contract to the customer.

```
Authorizer: "kBank1"
Licensees:  "kCust"
Conditions: @Val==0.01 && Num=1000 &&
   Contract == "tIyJelErKyLtYoIJQPw/bQ==";
Signature: ....
```

Figure 8.4: Customer's Contract Credential

Attribute `Contract` provides the initial contract value $[h_n(s), n, val]$ where s is the secret seed known only to the bank and customer. This particular contract is for a maximum of 1,000 micropayments valued at €0.01 each. Alternatively, the bank could decide to write a different credential that delegates authority to generate contracts directly to the customer.

When requesting service X, the customer provides $[h_{1000-i}(s), ...]$ as the i^{th} micropayment to Alice; in this case we assume that K_{Cust} has already made payments $[h_{999}(s), ...]$ to $[h_{999-i}(s), ...]$ to the company. The company uses the trust management system to determine, given the credentials, whether the customer is authorised for the particular request. Attribute bindings $[Val = 0.001; Num = 1000; Service = X; Contract = h_i(v)]$ define the circumstances of the request. The company uses the KeyNote query engine to search for a delegation chain that links the trusted key K_{Bank1} (from her policy credential) to K_{Cust} (the requester) and satisfies the circumstances of the request. $\triangle$

By casting micropayments in terms of trust management, we obtain a framework that provides flexibility in managing complex payment and service authorisation trust relationships. For example, the company might decide to out-source their support for service X to Alice by writing an appropriate KeyNote credential that delegates their authority (contract with the customer) to Alice. Similarly a bank can create contract credentials for customers, who can then delegate these contract credentials to their service providers. In such an architecture, the service provider can then invoice the bank directly, by providing both contract credentials.

Example 8.3 Figure 8.5 shows a simple payment invoicing protocol. As before K_{Comp} has received a contract from K_{Cust}. K_{Cust} sends K_{Comp} $n-i$ micropayments using this contract. K_{Comp} wishes to claim for these payments and so must invoice K_{Cust}. K_{Comp} generates an invoice detailing the last hash received, the number of payments that are being claimed, and a time-stamp of this transaction. K_{Comp} signs this invoice and sends it to K_{Cust}, with a copy of the original contract.

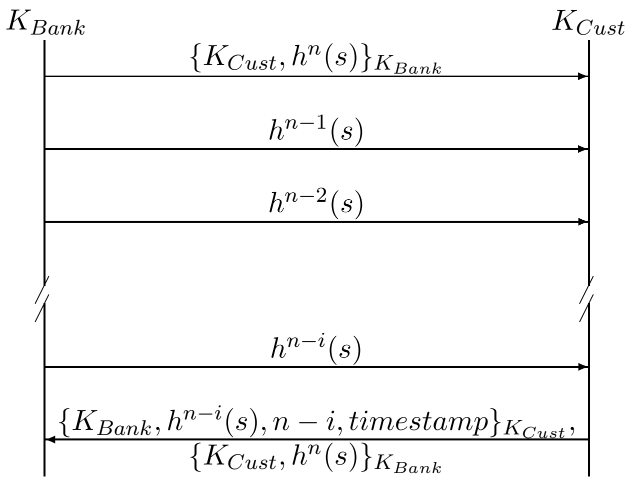

K_{Bank}
K_{Cust}
$\{K_{Cust}, h^n(s)\}_{K_{Bank}}$
$h^{n-1}(s)$
$h^{n-2}(s)$
$h^{n-i}(s)$
$\{K_{Bank}, h^{n-i}(s), n-i, timestamp\}_{K_{Cust}},$
$\{K_{Cust}, h^n(s)\}_{K_{Bank}}$

Figure 8.5: Invoicing

K_{Cust} now checks the validity of the contract, and that no claims for invoiced payments have been made before. Once K_{Cust} is happy with the transaction K_{Comp} is credited with the required payment. $\triangle$

8.2.2 Security Analysis

There are several security risks associated with the micropayment scheme outlined above. These include the risk of double spending, where a bank's customer attempts to use the same contract to pay multiple service providers; and the risk of double invoicing, where the service provider attempts to cash in the same contract multiple times.

Double spending is difficult to prevent in a distributed environment. However, this can be detected after the fact when the second service provider attempts to redeem the contract. At this point, a dispute resolution protocol will be used. In the case of a customer attempting to spend the same contract multiple times, the fact that both of the service providers hold a signed credential from the customer, delegating the contract to that provider, proves that the customer is attempting fraud.

Double invoicing is a simpler problem to address. As each invoice is presented to the bank, the bank can check to ensure that this contract has not been previously redeemed. This requires that the bank must store all redeemed contract credentials for all time. However, this cost can be reduced when the contract credentials are only valid for a fixed time period. The bank must only then store redeemed contracts that are still valid.

8.2.3 Micropayments in Secure WebCom

We can implement this scheme in Secure WebCom by modifying the trust management-based security manager so that it explicitly supports micropayments [64]. The micropayments security manager maintains a database of received coins, along with a list of current contracts. When a node is scheduled to a WVM for execution, it must be accompanied by a digital coin, called a *cyclot*. This coin forms part of the domain tuple of the node name, such as the name shown in Figure 8.6. The information contained in such names is used by the access control mechanism to enforce the

```
(Snam
  (Lnam ["kBank1","kCust","Contract 9LIwhTqls1ZuKdgfgHcSzQ=="
                  "Iter 10","Cyclot Xnd5ft1qHlEoU9k91/rf1A=="])
  (Lnam ["kBank1","kCust","kAlice","SampleAp")
  (Lnam ["kBank1","kCust","SampleAp","SampleOp"])
  [(Lnam ["kBank1","kCust","SampleAp","InputOp"])]
  [(Lnam ["kBank1","kCust","SampleAp","OutputOp"])]
)
```

Figure 8.6: A Node Name including a digital coin

pay-per-execute requirement. In this example, K_{Bank1} has created a contract for K_{Cust}, who is using this contract to pay K_{Alice} to execute application `SampleAp`.

The security manager extracts the contract, iteration and cyclot information from the name. This information could form the base of a trust management check, as described in Section 8.2.1, or using a very simple access control mechanism, calculate whether hashing the `Cyclot` value `Iter` times results in the `Contract` hash.

Using the trust management mechanism, the initial overhead incurred by a WebCom server when computing a micropayment contract for a client is offset by the subsequent computational burden that is off loaded (and paid for, task by task) to the client. Client side checking of payments carries minimal overhead, becoming part of the existing credential based authorisation check. It is possible that the child can cache the last valid coin they receive and, therefore, only need to compute one hash to ensure the next coin is valid, without requiring a trust management check.

8.2.4 Discussion

Blaze et al [35, 95] use KeyNote to manage trust for a micro-billing based payment scheme. Their scheme is similar to IBM's minipay scheme [86], whereby a credential represents an electronic cheque issued by the authoriser to the licensee. KeyNote is used by the payer (merchant) to determine whether or not an off-line payment from a particular payee (customer) should be trusted, or whether the payee should go online to validate the payment and payee. The scheme is intended for small value payments (under $1.00). Since each payment transaction requires a public key cryptographic operation it may not be practical for very small payments where the cost of processing is high relative to the value of the payment.

In [163], Shirkly argues that micropayments struggle to attain widespread usage because of their very nature. He states that users require "predictable and simple pricing" whereas micropayments "waste the user's mental effort in order to conserve cheap resources". When a user interacts with a micropayment system, they still have to make a choice: whether or not item X is worth Y. For example, imagine a website where a micropayment is required for each page viewed. A user will not automatically view each page without carefully considering whether that webpage is worth spending a tiny amount of money for. The amounts being considered are so small that this decision is wasteful. The only case where a user is willing to accept a transaction automatically is where that transaction is free.

In the system described, this is not an issue. Users do not make decisions on which transactions to accept; instead the decision to accept (trust) a transaction is performed by KeyNote. In effect the user's mental effort is represented in terms of a KeyNote policy. Another criticism of micropayment schemes is based on the ratio of setup cost versus the total cost of the transactions. Ideally, the relatively large cost of initialisation is offset by the frequent use of the low cost payment over time. With the application described in this section, this is particularly true. Relationships between payer and payee are generally long term and this setup cost is insignificant compared to the total cost of the transactions processed.

In this section we have shown a new scheme for micropayments using the KeyNote trust management system. This scheme has been implemented in the WebCom meta-computer, providing a method of rewarding clients for work completed. Our proposed scheme can be extended to provide for an evolution of trust between the server and client. The client originally does not trust the server to reimburse the micropayment, and might seek to (inefficiently) cash in each micropayment as it arrives. After n such reimbursements have been made the client might write a new KeyNote policy stating that it is safe to seek reimbursement after every m micropayments.

8.3 GridAdmin

GridAdmin [48, 147] is a WebCom-based system used to provide automated support for administrative requests, such as resource reservation and user account management. We propose using trust metrics to help judge the merits and suitability of each request. We outline how these metrics can be implemented using trust management techniques.

Grids [7, 71, 181] consist of numbers of sites cooperating to share resources. These resources are heterogeneous in nature and are maintained by a range of administrators, from full-time professionals to volunteers. The purpose of these sites is to share their resources and knowledge throughout the Grid. Grid middleware such as Globus [71, 72] facilitates the sharing of these resources and provides an identity based security infrastructure using X.509 certificates. X.509 certificates provide authentication support for users submitting jobs to remote sites. However, this does not directly address the issue of administration across the Grid: It is difficult for an administrator to decide how to react to a request from a different site, or to know whether an executable or configuration file from another site should be trusted. This is a hindrance to resource sharing.

Facilitating inter-site administration requires the definition of policies and some knowledge of each principal in the Virtual Organisation (VO). A VO is a virtual space across organisations that allows its members to interact transparently. This is a major overhead for system administrators. GridAdmin is designed so that a request from a well-trusted administrator at a different site would be approved automatically, whereas a request from an under-qualified user (for example, a student requesting a resource reservation) would require further investigation.

It is also important to consider how automation of user requests, software installation and upgrades, resource reservation, both within, and across sites should be achieved.

8.3.1 Administrating a Grid

There are several types of problems associated with grid administration. These problems form two main categories: local problems within an organisation and cross-site problems within a virtual organisation. We will examine these problems separately.

Local Problems

At any site, administrators face several basic issues. Users need accounts to operate in that site. Each user may have a different software requirement, and will need their specialised software installed on each resource they intend to use. Users may wish to request exclusive use of the resource, for example for critical timing, or simply to reserve access to the resource. The system administrator will handle these requests according to his/her knowledge of the user making the request, and within the constraints of the local policy.

Example 8.4 Site A has a policy which states that only postgraduate students and staff are allowed to make requests on their compute cluster (for accounts, software installation and for resource reservation and/or co-reservation). Priorities are assigned based on seniority of the requester and the urgency of the work. An undergraduate student could be granted an account by their supervisor, but couldn't request a cluster booking: such a request would have to be brokered by their supervisor directly. △

Cross-site Problems

When these local problems are translated to a Grid environment, they become more challenging. The same decisions must be made with less information available to the administrator.

For example, a user at site A wishes to use resources at site B, where both sites are in the same VO (Virtual Organisation) and have a functioning Grid. Currently, the procedure would be to send an email to site B's system administrator requesting that the attached executable file be installed. This would be accompanied by a configuration file used to set up the executable file. The administrator must now decide:

- does the user have a right to access resources on site B (covered under the terms of the VO agreement and handled by Globus);
- whether the user at site A is authorised to make such a request (perhaps local policy dictates that only senior staff members can make such requests);
- whether he should trust the executable file, or should the source code be consulted, and
- whether the configuration file is trustworthy.

It is obvious that the ability to trust the user would greatly improve cross-site cooperation, and facilitate resource sharing. It is easier to accommodate requests from system administrators of different sites who have established relationships, and therefore have some level of shared trust. These decisions could be taken according to the established trust relationship and in terms of the local policy.

In the absence of a trust model, it is tempting to take an authorise all, or authorise none approach, to requests from outside the administrator's domain. These approaches may hinder the spirit of the agreement and reduce inter-site cooperation.

System administrators at different sites in a Grid tend to talk to one another, even when only to exchange the bare information required to set up a Grid (for example, machine names for firewalls or user names for grid map files etc.). For this reason, some level of trust (or history) is established between them. This trust can be leveraged, along with the constraints of the local policy, and the VO agreement to interpret requests. This is rarely the case with ordinary users. It is more likely that administrators will have little or no knowledge of individual users from other sites, but will have established relationships with their administrators. The user can normally only prove their identity and their right to access the resources under the VO agreement through the Globus X.509 security system.

Trust Management is used to help automate administrative decisions rather than replacing the existing Globus security infrastructure. The contribution of our approach is to provide a framework in which Grid administration becomes more practical. In this section, we explore two approaches — explicit and "fuzzy"— to supporting Trust Management in Grid Administration. Explicit delegation of authorisation requires full authorisation details to be encoded within the Trust Management credentials. However, it does not capture the flexible nature of a real system. To this end, we propose alternative metrics to provide a means to make "fuzzy" delegations. These allow administrators to quantify the level of trust they apply to each of their users.

8.3.2 Grid Administration using WebCom

Supporting automated administrative requests on a Grid resource, requires an administrative helper “daemon” running on that resource. These agents take administrative requests, such as node reservation, and perform the low-level changes to the resources. For example, a successful request to reserve compute nodes would call the agent in charge of those nodes, and modify the system to only allow that user to log in during the reservation period.

We can represent administrative tasks as condensed graph applications [63]. The security infrastructure can then be used to decide whether these administrative actions are authorised. Thus WebCom becomes the administrative agent on the nodes. The WebCom system operates as a trusted application on the nodes, and the security manager ensures that the tasks it executes comply with the local security policy.

Example 8.5 A common problem for Grid Administrators is allowing users exclusive access on Grid resources for a period of time. Typically the Administrator must go to each resource required and prevent other users from accessing that resource for the duration of the booking. This task can be represented in an administrative condensed graph, shown in Figure 8.7.

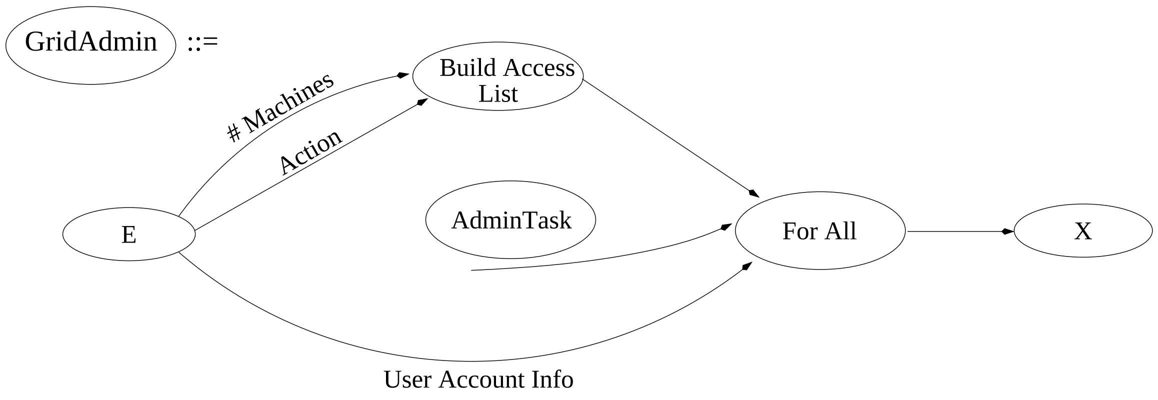


Figure 8.7: Condensed Graph Application to reserve Grid resources.

This graph specifies the sequencing of the application. First, the parameters are built up using `Build Access List` Node. This seeds the `ForAll` node with the number of machines needed and the administrative action requested. The `ForAll` node spawns copies of the `AdminTask` node, one for each resource that will be modified. The `AdminTask` node represents the administrative action to be applied on the Grid resources. In this case, it executes the exclusive access administrative request.

The policy credential shown in Figure 8.8 grants the Grid Manager the authority to assign up to 100 Grid resources exclusively to users. The Grid Manager uses this authority and signs the credential, shown in Figure 8.9, that allows the User to reserve 32 machines. The user presents this credential to WebCom to prove her authority when she wishes to reserve the resources.

```
Authorizer: "POLICY"
licensees:  "KGridManager"
Conditions: App_Domain == "WebCom" &&
            Resource == "UCC-GRID" &&
            (Graph == "GridAdmin" &&
            Function == "ExclusiveAccess" &&
            Input <= 100);
```

Figure 8.8: Policy Credential allowing the Grid Manager to assign exclusive access to up to 100 resources.

```
Authorizer: "KGridManager"
licensees:  "KUser"
Conditions: App_Domain == "WebCom" &&
            Resource == "UCC-GRID" &&
            (Graph == "GridAdmin" &&
            Function == "ExclusiveAccess" &&
            Input <= 32) &&
            _ACTION_AUTHORIZERS == KUser;
Signature: ...
```

Figure 8.9: User Credential, delegated by the Grid Manager, to allow reservation of 32 resources.

The `AdminTask` Node takes two parameters, the first is the action to be performed. This is either ‘grant’ or ‘remove’. The second parameter is the user account information. This consists of a colon delimited string with the user name, encrypted password string, UID, gui and home directory. The Node then starts or stops the NIS service as appropriate and either adds or removes the account from the machine. The UID, GID and home directory are passed in to avoid having to make changes to the NIS and NFS servers. This allows ordinary users to make major changes without the need for root access.

With GridAdmin the user need only specify a number of machines and the account information (readily available from the NIS server). To do the same task by hand, the System Administrator would have to log into each node, manually edit the password and group files to include the necessary information, and enable/disable NIS[2].

Figure 8.10 shows the amount of time taken to grant exclusive access on 1, 2, 4, 8,16 and 32 nodes of the Boole Machine. The Boole Machine, owned by the BCRI (Boole Centre for Research in Informatics) machine is a 100-cpu Beowulf cluster. It is used by over 70 researchers in 5 institutions for research into fields including Computer Science, Mathematics, Applied Mathematics, Physics, Astrophysics and Geophysics.

This time (in milliseconds) excludes the time to set up the variables, such as the user name, password, etc., for the entire run. This took 25 seconds at startup. However, this is a static per request cost. For comparison, making these changes manually was timed at 45 seconds per machine.

The advantage of this approach is apparent. The time taken for execution increases rapidly at

[2]Of course these actions could be automated with scripts.

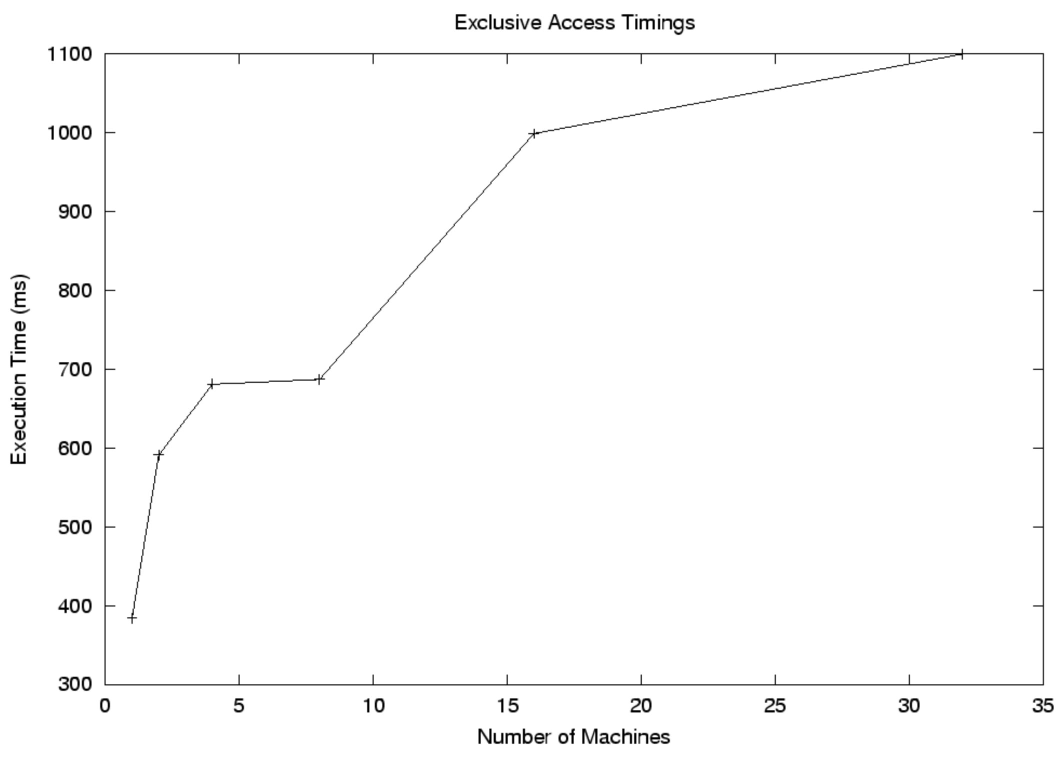


Figure 8.10: Timings to Execute Graph from Figure 8.7 on 1,2,4,8,16 and 32 Machines.

first, but as the number of client machines grows, this cost gradually flattens out. In this case, the administrator makes the request, but as the policy is enforced using trust management, this is not required. The administrator could write credentials delegating this authority to specific users. This authorisation could specify the maximum number of machines that any one request could effect. However, some external control should be applied in this case, as an unscrupulous user could avoid this restriction by attempting multiple requests. △

8.3.3 Trust Paradigms for Grid Administration

We have outlined how WebCom's Trust Management infrastructure can provide a basis for decentralised security administration in the Grid. Such a system has the ability to make authorisation decisions about user requests. Considering the problem of how to administrate islands of resources on the Grid, we can readily recognise the advantages of using Trust Management credentials to drive administrative actions.

Analysing the administrative problem further, we identify three common administrative transactions:

1. Adding a remote user to a local Grid resource. Users in remote systems often request access to local systems. The local administrators may have no personal knowledge of the remote

user, and are forced to make blind decisions regarding the user's eligibility and access level;

2. Providing the ability to book exclusive resource allocations, as described in Example 8.5. Users often require exclusive access to resources for diverse reasons. The administrator makes decisions regarding these requests based on such considerations as past behaviour. For example, when the user last had exclusive access, did they use it properly?, and

3. Providing an infrastructure for users to request custom software installation. Administrators make decisions on new software installations again based on the requesting user's past performance and the skill level of that user; An experienced user will often receive a more positive response than a novice.

There are several potential approaches to solving these problems with a Trust Management framework.

Reputation based metric

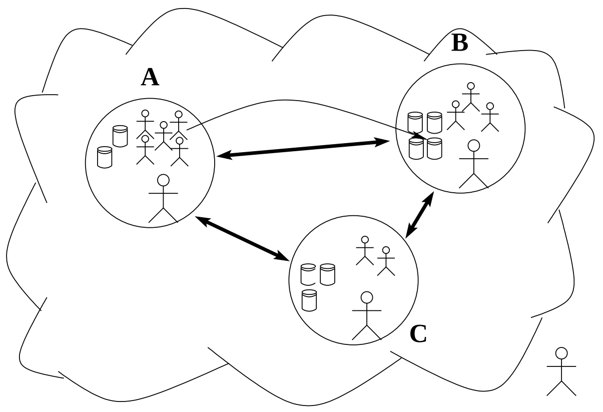


Figure 8.11: A Virtual Organisation, with three organisations sharing resources.

Administration of different domains, such as the VO shown in Figure 8.11, rely on informal relationships between the administrators of those domains. Formalising these relationships into a model would provide a more consistent outcome for user requests. An analogy can be seen in the relationships between nightclubs in a locality. In general, they are competing businesses. However, if a person misbehaves in one club, then their reputation often proceeds them to the other clubs in the area, through the "network" of doormen. This is a model well suited to the administration of Grid resources.

Using reputation based metrics for measuring trust is a well established technique [101, 156, 183]. Analysing the Grid architecture in order to use reputation to promote data integrity has previously been explored [75]. Knowing the reputation of a user can provide an insight into what access you give that user. Maintaining a measure of each user's reputation allows an administrator

```
Authorizer: "KUCC-Admin"
licensees:  "KBob"
Conditions: App_Domain == "GridAdmin" &&
            Resource == "UCC-GRID" &&
            karma = 0.52;
Signature: ...
```

Figure 8.12: Karma Credential for User kBob.

```
Authorizer: POLICY
licensees:  "KUCC-Admin"
Conditions: App_Domain == "GridAdmin" &&
            Resource == "UCC-GRID" &&
       (node_request > 5) -> (karma > 0.6);
```

Figure 8.13: Karma Policy, allowing conditional access to Compute nodes.

to make decisions about allocating the resources of the system to those users. We call this measure a user's *"karma"*. Karma is a numerical value representing the level of trust that a user has attained in the local system. This value represents the user's previous behaviour in the system. The higher the value, the greater the user's potential access. Applying a numerical weight to users allows creation of more user-understandable policies. For example, depending on the karma level of a user, automatic decisions regarding access to resources may be made.

We envision karma to be represented by a value between 0 and 1. Users could potentially receive their initial karma level $(K^u_{d_l})$ depending on their introducer's karma in the local domain $(K^i_{d_l})$ and the user's karma in the introducer's domain $(K^u_{d_i})$. An introducer is an authority in an associated domain, trusted to some level by the authorities in the local domain. For example:

$$K^u_{d_l} = K^i_{d_l} * K^u_{d_i}$$

Over time, depending on the user's behaviour, karma will rise and fall. Good behaviour, such as properly using reserved resources, is rewarded with increased karma, and therefore access. Consequently bad behaviour, such as requesting new software, but not using it, will result in reduced karma.

Karma could be encoded into trust management credentials, such as KeyNote credentials. For example, Figure 8.12 shows a karma credential for user KBob. This credential, signed by an administrator of the UCC-GRID domain, sets his karma level as 0.52. The flexibility of this system is apparent when examining the sample policy shown in Figure 8.13. This policy indicates that if a user wishes to reserve more than 5 compute nodes in the local domain, then their karma must be over 0.6. Additional conditions could be placed in this policy, such as, if a user was below 0.6, then they would need to provide a request co-signed by another user.

Other usage of karma could include assigning a karma level to machines, based on their setup. This would allow the creation of policies where machines that are very stable (high karma) would be reserved for users who also have high karma. When machines have problems, their karma drops. Consequently a high availability increases the machine's karma level.

Difficulties with a karma-based metric are in the administration of updates to the user karma values. How are changes to the user's karma level stored and enforced? Users will probably be willing to throw-away an old credential, where the replacement has a higher karma level. However, getting them to use a new credential with lower karma is more difficult. This issue can be addressed in a number of ways. Expiry dates could form part of the user karma credential, forcing the user to obtain a new credential periodically to continue using the system. Unlike Certificate Revocation Lists (CRLs), this places the burden of proof of authorisation on the user [154]. Another solution would be to store changes to each user's karma in a central *"karma server"*. This, however, introduces a single point of failure into the system, and does not have the advantages of a decentralised approach.

Furthermore, we must consider how to handle multiple introducer credentials for a given user. How do we aggregate different karma levels from different introducers? Also should changes to the introducer's karma reflect on the karma of the user? An example of such a system can be seen in [1]. Finding adequate solutions to these problems is important in order to create a useable system, and is a topic for future research.

Reputation based metric in WebCom

Supporting a reputation based metric in WebCom requires checking the requesting user's karma level, compared to the level required by the system policy for the requested administration action. If the user's karma is high enough, then the request is accepted. If not, then either a request for confirmation is made to an administrator, or, if the policy so dictates, then the request is automatically denied. These administrative actions are specified as condensed graph workflow applications.

Example 8.6 Reservation of Grid resources is specified in a condensed graph workflow application shown in Figure 8.14. When a user wishes to reserve such resources, this workflow is launched and the components executed on the relevant resources.

This security policy of the environment, in which the workflow application is running, defines that the `Book Resource` operation should be scheduled to a principal whose karma is greater than 0.4; The `Grant Request` operation is only allowed to be scheduled to a principal with karma greater than 0.6; The `Display Result` operation can be scheduled to any valid user. This policy is defined in a policy credential, such as in Figure 8.13.

KBob's credential (shown in Figure 8.12) indicates that he has sufficient karma to make a request to book a resource, however he would need someone else to approve that request, as he does not have enough karma to do so by himself. The result of the request would be displayed on KBob's machine, as he is a valid user of the system (i.e. his karma is greater than zero). △

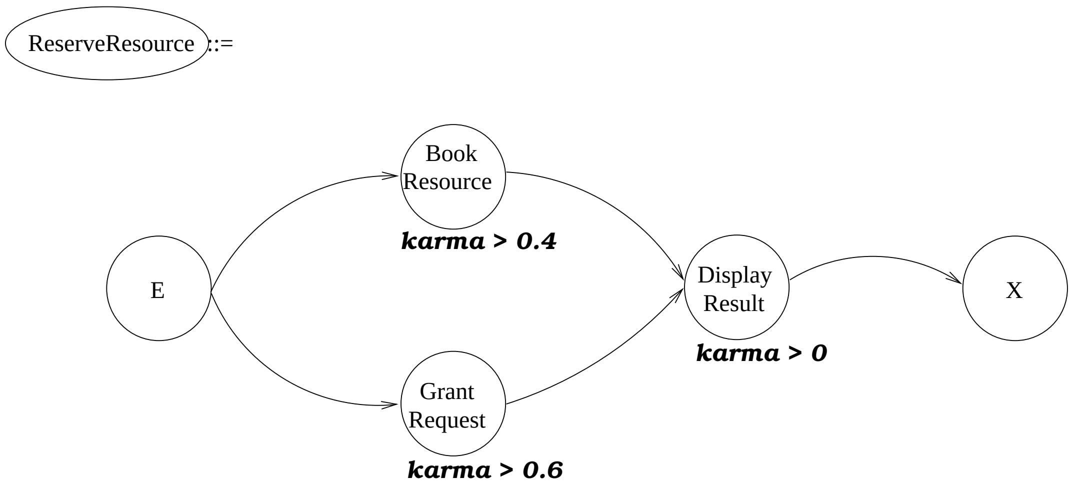


Figure 8.14: Condensed graph workflow application to reserve a resource

WebCom's naming architecture must ensure that the karma level associated with every node is retained, enabling the security manager on both the parent and child WVMs to make proper reputation decisions. The karma level is stored in the domain tuple of the node name. Furthermore, the karma levels for users may be supported by a karma user module in WebCom (see Chapter 5).

Using this combination of the workflow abilities of condensed graphs, and the security managers of WebCom we can construct a flexible automated security administrator. As the GridAdmin application, described in Section 8.3.2, already uses WebCom to schedule the administrative tasks, there is no additional overhead. Instead different credentials form part of the existing authorisation checks. However, this does not address the accounting problems with such a reputation metric.

Assurance based metric

Using money as a trust metric has been growing in popularity in recent years. In Section 8.2, we described how we can use micropayments within WebCom to pay for the execution of nodes. We can use a similar technique towards paying for administrative actions in GridAdmin. Applying monetary or assurance [119, 150, 162] terminology to trust decisions is appealing as the stakes involved in a system are readily understandable. Unlike the reputation based metric, a monetary based metric requires no storage of the changes in each user's fortunes: users take care of their own money.

In a monetary based system, money is exchanged between principals. To use a resource, an agreed sum must be paid to the owner of the resource. It is important that the trust mechanism has a low computational and administrative cost, and also that contracts between users must be both verifiable and subject to conflict resolution. Such a system can be implemented using a trust management system [35, 59, 64]. These systems work on the basis that either the payments act as electronic cheques, that are reimbursed later, or are used as a closed currency. In a closed currency system, payments take the form of coupons, traded for resources. Ideally principals must either "save up", or several principals must combine, to request an "expensive" resource. Such a system discourages bad behaviour, as the abuser will lose money in the transaction.

A difficulty with such a metric becomes apparent when considering problems experienced in economics. For example in [108], Krugman introduces the problems with Babysitter clubs, that are common in the US. In these clubs, each set of parents are initially issued a fixed number of seed coupons. These coupons are used to pay other parents when a babysitter is required. When a parent wants a night out, they spend a coupon and another parent babysits for them. However over time, the system collapses due to hoarding of coupons by parents "saving up" for a special occasion. Other parents noticing the lack of babysitting jobs also stay in, preferring to save their coupons for emergencies. Applying this behavioural result to the proposed metric, leads to the conclusion that similar problems may well be experienced.

Instead of a coupon based metric, consider instead a deposit based system. In such a system, a "promissory note" is signed by the principal requesting the resource. If they behave properly, then the contract is returned after some period. However, if abuse of the resource takes place, then the owner of that resource cashes in the contract, reducing the future purchasing power of the principal. This is analogous to an insurance policy. This is in effect an "Assurance" policy. If the user abuses the resource, then the assurance policy is invoked, and compensation is paid.

Seeding the system requires that a trusted source, for example a bank, must set limits on all the principals using the system. This can be achieved using a trust management system. Initially authorities in a Virtual Organisation are delegated a certain amount of credit. These authorities can then pass on portions of this credit to their local users.

Example 8.7 In the Virtual Organisation shown in Figure 8.11, there are three component domains, A, B and C. The KeyNote policy shown in Figure 8.15 assigns a credit of 1000 to KAngela, the Administrator of domain A's key.

```
Authorizer: POLICY
licensees:  "KAngela"
Conditions: App_Domain == "GridAdmin" &&
            Resource == "UCC-GRID" &&
            Credit = 1000 &&
            Validity <= 200404312359;
```

Figure 8.15: Administrator Angela is delegated a credit of 1000.

These credentials have a validity date, up to when the credentials are valid. These validity dates allow the legitimate reuse of the credit that a user holds, without requiring the administrator to explicitly return the deposits. KAngela can then delegate parts of this total to users in her domain. Such a delegation is shown in Figure 8.16.

This credential delegates a credit of 100 to KBob. KBob could now use this credential to generate a contract, guaranteeing good behaviour when requesting a resource in domain B. As each of

```
Authorizer: "KAngela"
licensees:  "KBob"
Conditions: App_Domain == "GridAdmin" &&
            Resource == "UCC-GRID" &&
            Credit = 100 &&
            Validity <= 200404142359;
Signature: ...
```

Figure 8.16: Administrator Angela delegates a credit of 100 to user KBob

the domains would trust administrators in the other domains, such a contract would be honoured in domain B. △

The metric outlined is essentially the opposite of a reputation based metric. Good behaviour simply guarantees continual access to resources. Bad behaviour would result in default of the contract, reducing the amount of money available in the future. If a principal misbehaves, then a conflict resolution process would be enacted. Using this process, the complainant would furnish the contract credential, and some proof of the bad behaviour. If the complaint is upheld, at the start of the next renewal period for the user credit credentials, then the credit of the misbehaving principal would be reduced, and the credit instead issued to the complainant. If principals can show good behaviour in terms of contracts successfully completed, then their issuing authority could choose to raise their credit limit. This is comparable to a credit card company increasing the credit limit of a good customer.

There is a potential problem with such a metric. Due to the decentralised nature of the proposed system, double spending, or promising the same deposit to more than one principal, becomes possible. A principal could make guarantees in domains B and C using the same collateral. However, we propose that this is, in fact, a desirable characteristic. If a principal acts properly in both domains, then the double spending will never become apparent. However, when a default occurs in both domains, the digital signatures will prove the guilty principal, and a conflict resolution process would take over. Such a system will reward a principal who takes more risks, yet whose behaviour is good. Good behaviour is likely to be increased, as principals are risking potential disaster when discovered.

Assurance based metric in WebCom

Alternatively, supporting a assurance based metric in WebCom requires a different type of credential infrastructure. When a user wishes to make an administrative request, they create and sign a contract credential. This credential is then sent to the Administrator of the resource requested. If the Administrator accepts the contract, then the request is granted. These decisions are taken based on the local policy of the resources requested. For example, if the policy stated the cost per minute of reserving a node, then the user would have to offer at least this amount for the request to succeed. Even though this metric is in practice the opposite of the reputation metric: principals must prove their worth, the system is not required to maintain state; these administrative requests can be specified in the same form as those used with the reputation metric.

Example 8.8 Principal KClare wants to reserve 15 compute nodes for 10 hours in order to generate some accurate results. In order to achieve this, she creates a contract credential, shown in Figure 8.17.

```
Authorizer: "KClare"
licensees:  "KUCC-Admin"
Conditions: App_Domain == "GridAdmin" &&
            Resource == "UCC-GRID" &&
            Request == "BookResource" &&
            Nodes = 15 &&
            Time = 600 &&
            Deposit = 100 &&
            Validity <= 200404142359 &&
            [...];
Signature: ...
```

Figure 8.17: KClare contract for reserving 15 compute nodes for 10 hours.

This contract credential allocates a deposit of value 100 to KUCC-Admin to guarantee KClare's good behaviour while using the requested compute nodes. For this request to be successful, KClare would have to provide a credential from a source trusted by KUCC-Admin, giving her the right to create such a contract credential. Figure 8.18 shows a credential fulfilling these requirements.

```
Authorizer: "KUCC-Finance"
licensees:  "KClare"
Conditions: App_Domain == "GridAdmin" &&
            Deposit <= 250 &&
            Validity <= 200404312359;
Signature: ...
```

Figure 8.18: Credit Credential from UCC's Finance Department, giving KClare's Credit limit.

This credential, signed by a key belonging to the Finance department in UCC, gives KClare the right to sign contracts up to value 250, in the GridAdmin application. Finally, KUCC-Admin's local policy must declare what price the Administrator is willing to accept for reservation of nodes. The policy must also trust the KUCC-Finance key for this request to be successful.

Figure 8.19 shows such a policy. In this policy credential the administrator has defined the conditions under which certain administrative requests are acceptable. Specifically, in order to reserve nodes, principals must provide a deposit based on the number of nodes required and the length of time (in minutes), they are required for. △

```
Authorizer: POLICY
licensees:  "KUCC-Finance" ||\
            "KNUIG-Finance" ||\
            "KTCD-Finance"
Conditions: App_Domain == "GridAdmin" &&
     ((Request == "BookResource" &&
       (Deposit >= Time * Nodes * 0.01)) ||
      (Request == "InstallSoftware" &&
       (Deposit >= Nodes * 100)));
```

Figure 8.19: KUCC-Admin's policy, trusting the keys of several Finance departments to assign credit limits. It also dictates the terms acceptable to the Administrator.

This system can be extended to encompass all the administrative actions concerning the administrator. Placing a monetary value on the actions allows the administrator to discourage certain actions, without outright refusal. For example in Figure 8.19, the administrator has defined the value 100 as the price to install a new piece of software on each node. These conditions can be as fine-grained as the administrator requires. For example, reserving an SMP machine could be much more expensive than a uni-processor node.

Additionally, using the architecture of the WebCom system, we can implement the *"Nightclub"* model previously discussed. Using the communication capabilities of WebCom, advisory credentials, written by administrators, could be distributed throughout the system and integrated into the trust management decision. These credentials could specify that a higher "Entrance fee" is required from users who have misbehaved on other systems. This provides a means to instantly reduce the purchasing power of individual users, without waiting for the renewal of credit credentials. This concept is similar to the idea of Certificate Cancellation Notices (CCN) in SPKI [94]. CCNs are an informal version of Certificate Revocation Lists (CRLs) with most of the benefits, but at reduced cost.

These decisions take place in a fully decentralised manner. Different administrators have different priorities, and so the policies will vary from domain to domain. Another advantage of this decentralised architecture is the ability to "sub-contract" work. It would be possible that KUCC-Admin decides to sub-contract some work to another domain. This is achieved by the creation of a new contract credential by KUCC-Admin to another administrator, delegating the deposit from the received contract credential. The original contract credential would be passed along to preserve the delegation chain.

8.3.4 Discussion

In this section we have introduced GridAdmin, an automated administrator, empowered to handle the tedious administrative requests, such as the reservation of compute nodes, common in grids today. This system provides a "value-added" service to Grid administration, sitting on top of the existing Grid architecture rather than replacing the existing security architecture. For example, the cryptographic keys used to sign credentials are the same keys used by the principals to authenticate themselves to the Grid management software, such as Globus.

Section 8.3.2 described how we use the WebCom system to provide automated administration of Grid resources. Our experimental results show the value of this approach, dramatically decreasing the amount of time required to perform common administrative tasks.

However, this implementation does not capture the flexible needs of a real users. To this end, Section 8.3.3 proposes alternative trust metrics that provide a "fuzzier" notion of trust. Each approach was found to have advantages and disadvantages, such as problems of aggregation in the karma system and conflict resolution in the assurance system.

The metrics proposed encompass alternative ends of the possible design of such a system. However we believe that the assurance metric provides an interesting, yet useful simulation of the real-life situations administrators find themselves in. Often we ask ourselves: "what's in this for me?; what guarantees do I have that this will not break our system?" These questions are addressed using an assurance system, with the cost/benefit analysis being readily understandable.

We are in the process of deploying such an automated system in regards to the Cosmogrid [10] project. This will reduce the time required to administrate, and increase both flexibility and sharing of resources between the component sites. We have implemented a prototype system on the Boole machine in UCC. Users have the ability to perform automated resource reservations.

In the future, we intend analysing both the usability of the GridAdmin architecture, and the suitability of the proposed metrics over time. The flexibility of a trust management based approach allows each site to alter their policies to suit local conditions, while providing a consistent infrastructure throughout the sites. Integrating some of the features of both proposed metrics into a combined metric also may provide some interesting results. More research into these metrics is required.

8.4 WebCom$_{DAC}$

WebCom$_{DAC}$ [63, 134] is a security orientated heterogeneous administrative tool based on the WebCom architecture. WebCom$_{DAC}$ (WebCom with dynamic administrative coalitions) acts as an administrative interface to an organisation's heterogeneous systems. Dynamic administrative coalitions are virtual administrative spaces where administration tasks are carried out for users. These tasks are specified as condensed graphs and are coordinated by trusted instances of WebCom, called DAC coordinators.

WebCom$_{DAC}$ provides an activity-centred model for structuring and organising administrative workflows on a heterogeneous network. These administrative workflows are described as *activity sets* [60, 61]. These activity sets are then translated into condensed graphs and executed by WebCom. Activities can be represented visually, as shown in Figure 8.20. This shows the template for an activity set. Activities are made up of a number of actions: `Start` to start the activity; `Finish` to conclude the activity; `Join` to join the activity; `Leave` to resign from the activity; and `Do` to perform some action. Activities support multiple actions of each of the different action types. In the implementation of activity sets, only join, leave and do actions are supported.

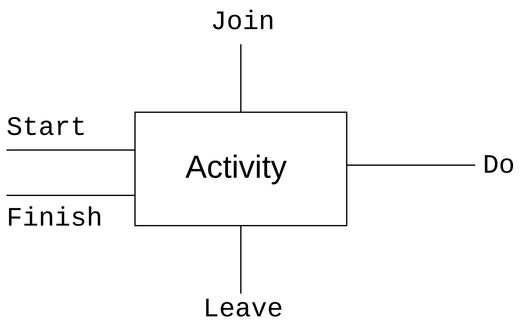


Figure 8.20: The Template for an Activity.

Activities can be linked together, for example, a `Do` action in one activity could be linked to a `Join` or `Start` action in another activity. Activity sets can represent business rules; each activity represents a different principal's duties. Example 8.9 describes a share trading activity using a number of linked activity sets.

Example 8.9 Figure 8.21 shows a share trading application cast as a number of activities. Each activity set corresponds to a user's view of their actions. For example, the CEO can appoint trading managers, sales managers, and can resign. The join action for the CEO's duties starts the activity by the appointment of the CEO.

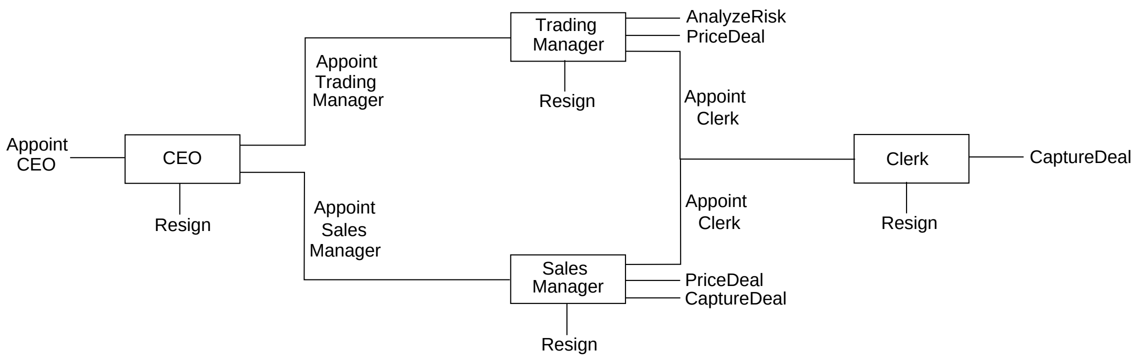


Figure 8.21: A Share Trading Activity Set.

The CEO's `do` actions entail appointing trading and sales managers. Both of these mangers can appoint clerks. Their other duties include analysing risks and pricing deals for the trading manager and pricing deals and capturing deals for the sales manager. Clerks can capture deals. △

The aim of the WebCom$_{DAC}$ system is to implement this activity model using condensed graphs to support the sequencing constraints. We use activity sets to specify the business rules for an organisation. These rules are then automatically translated into condensed graph workflow applications [134]. We securely execute these workflows using WebCom.

8.4.1 WebCom$_{DAC}$ Architecture

We have examined how administrative workflows are specified for WebCom$_{DAC}$. However, this is but one aspect of the system. In order to provide administrative support in WebCom we also require the ability to view and modify authorisation policies on the resources of the system. DAC policies are specified and controlled by the DAC policy tool.

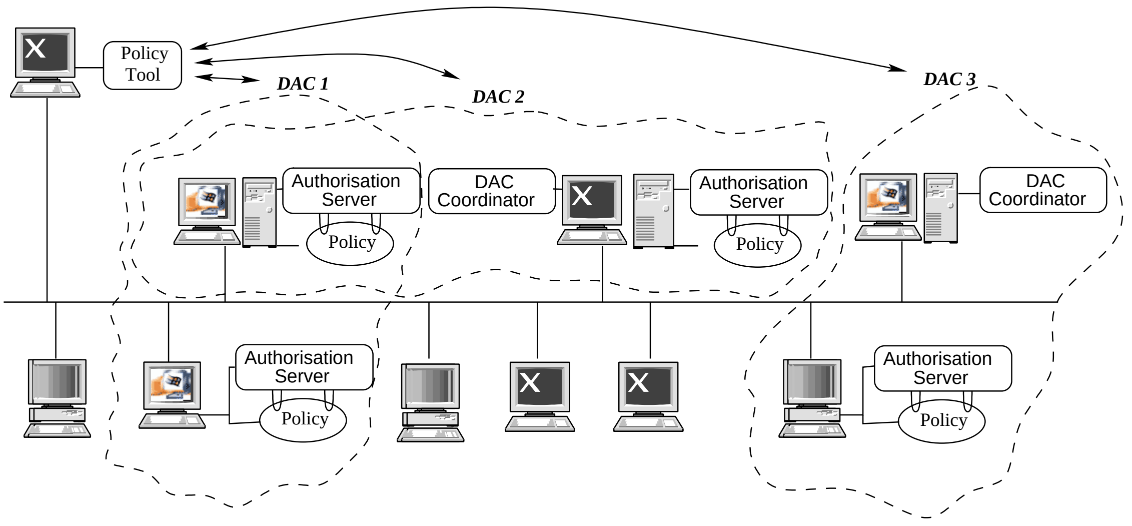


Figure 8.22: The DAC Architecture

The DAC system operates though the use of DAC coordinators to schedule administrative tasks, and authorisation servers to perform those tasks on their corresponding servers. Figure 8.22 shows a representation of a network containing a number of administrative coalitions. Implementation of authorisation servers exist for a number of architectures, including DCOM/.Net (KeyCOM), Linux (KeyLin) and Enterprise Java Beans (KeyBean). We examine these authorisation servers later.

The DAC coordinators are implemented as trusted instances of WebCom, called WebCom$_{DAC}$. WebCom$_{DAC}$ is used by the DAC policy tool to coordinate administrative workflows on legacy systems. These systems provide executable components that form part of applications. The authorisation servers can be used to view and/or update the authorisation policies for these components.

DAC domains can overlap, for example *DAC1* and *DAC2* in Figure 8.22. This could potentially cause an issue where an administrative update on one authorisation server by one DAC conflicts with the requirements of another DAC. We do not believe this is a significant problem as WebCom$_{DAC}$ is a closed system and a single workflow is executing at any time. If a change is made to a authorisation server policy, that policy update is reflected in the DAC policy tool.

8.4.2 Implementing WebCom$_{DAC}$

The WebCom$_{DAC}$ architecture is shown in Figure 8.23. It is broken down into a number of components. The middleware systems, such as COM or EJB, provide the components that can be used by WebCom applications. Both the components and the security policy are discovered by *interrogators*. These interrogators determine the services that are available on a middleware server and input this information into a database. This database is used by the WebCom integrated development environment (IDE) to allow developers to create applications using these services. During component interrogation, the interrogators also extract the security policy for these middleware components and also store this information in a database. This information is represented in a palette in the WebCom IDE.

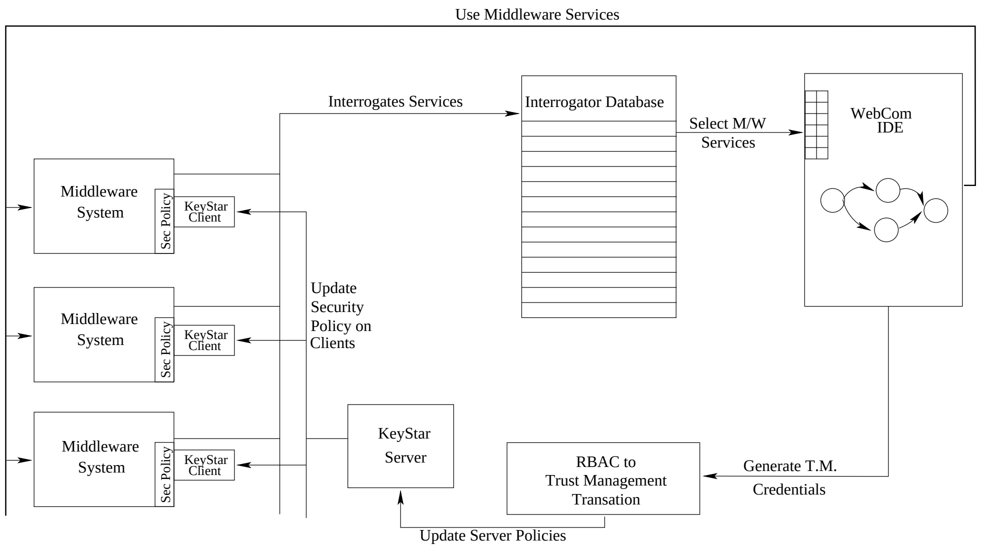


Figure 8.23: The WebCom$_{DAC}$ Architecture.

Once an application has been created, an access control policy must also be created for this application. Typically, this takes the form of a trust management policy. This may require changes to the authorisation policies of the middleware systems. In this case, the *KeyStar* subsystem is used to make these updates.

Middleware systems typically use a type of role based access control (RBAC) for authorisation. RBAC-like policies can be encoded in terms of equivalent cryptographic certificates/policies [109, 152]. In addition to supporting ad hoc KeyNote policies, WebCom$_{DAC}$ supports middleware RBAC-like security policies within KeyNote authorisation credentials. This is unlike [109], where authorisation certificates are only integrated as part of the lower-level middleware system. WebCom$_{DAC}$ uses KeyNote to determine whether it is safe to execute a middleware component.

A WebCom$_{DAC}$ environment can automatically convert middleware RBAC policies to their equivalent KeyNote policies/credentials, and vice-versa. This provides a high degree of policy interoperability, both between the middleware and trust management layers, and within different Middlewares. In addition to providing a uniform way of specifying RBAC policies for different middleware systems, it also becomes possible to enforce standardised RBAC middleware policies across middleware systems that do not have a configured RBAC policy.

Role-based access control (RBAC) [158] is widely used to provide access control in Database management systems, operating systems and Middleware architectures. In RBAC, access rights (permissions) are associated with roles, and users are members of these roles. When a user is assigned to a role, they gain all the permissions of that role in the system. This allows an organisation to model its security infrastructure along the lines of its business. For the purposes of this dissertation we extend the conventional RBAC model of *Users*, *Roles* and *Permissions*, to include *Domain*.

- *Permission*: represent actions, capabilities, applications or any other active behaviour that can be "performed" and, to which, we intend to control authorisation.
- *Domains*: administrative boundaries that group permissions and manage their underlying resources. In general, domains do not intersect in their underlying permissions.
- *Roles*: roles are logical groupings of permissions that reflect a particular task that can be assigned to some user. We assume that roles do intersect in their underlying permissions.
- *Users*: include humans or any other entity that can be assigned a role.

An RBAC policy is defined in terms of the following relations.

$$\begin{aligned} RolePerm &: (Domain \times Role) \leftrightarrow Permission \\ RoleUser &: (Domain \times Role) \leftrightarrow User \end{aligned}$$

where $RolePerm((d, r), p)$ means that the role r (in domain d) holds permission p (on some object), and $RoleUser(d, r, u)$ means that user u is assigned to domain-role pair (d, r). Table 8.1 uses this model to provide a uniform interpretation of basic COM+, EJB and CORBA middleware RBAC policies.

There are a variety of approaches to supporting roles in KeyNote. Encoding the fixed relationships from the Domain-Role-Permission table as a single KeyNote credential provides a simplistic representation of the RBAC policy. Individual credentials are then issued, associating users to roles.

Example 8.10 The ShareTrader Domain-Role-Permission table can be encoded as the following policy credential.

Type	*Domain*	*Role*	*User*	*Permission*
EJB	Combination of Host, EJB Server, relevant bean container.	Application Specific for each server.	Exist globally on each server, can be members of different roles.	Method calls (of an object type) that roles are permitted to make.
COM	Windows NT Domains.	Unique to Domains.	Windows Users. Unique to each Domain.	Considering only `Launch`, `Access` and `RunAs`.
CORBA	Machine name and ORB server name.	Unique to Domains.	Can be members of different roles, unique to each server.	Relate to method calls on objects of the given object type.

Table 8.1: Interpretation of Middleware RBAC Models

```
Authorizer: POLICY
Licencees: "Kwebcom"
Conditions: app_domain="ShareTrader" &&
  (Domain=="mgmt"&&(role=="TraderMgr") ->
    (perm=="setlimit"||perm=="analyzerisk"||...);
   ....
  (Domain=="staff"&&(role=="sales") ->
    (perm=="pricedeal"||perm=="capturedeal");
```

This specifies that the WebCom administration key `Kwebcom` is authorised to administer rights in connection with this policy.

```
Authorizer: "Kwebcom"
Licencee:   "Kjoe"
Condition: app_domain=="ShareTrader" &&
           role=="Trader";
```

This credential authorises Joe as a Trader. △

The above approach promotes a more centralised policy administration, with the WebCom environment (administrator) managing delegation and is comparable to the conventional middleware approach.

Alternatively, the Domain-Role-Permission table can be decentralised and spread across a number of credentials and additional authorisations and role memberships delegated to other keys. A common strategy is to represent roles (from domains) in terms of public keys; delegation is used to create the role-permission and role-user relationships. In practice, roles are best supported using SDSI-like local names [152], however, we can approximate the role membership effect in KeyNote as follows.

Example 8.11 Public keys `KRtrader` and `KRsales`, etc., are used to represent roles. Credentials associate authorisations to the roles. For example,

```
Authorizer: KRtrader
Licencee:   KRsales
Condition: app_domain=="ShareTrader" &&
    perm=="pricedeal"||perm=="capturedeal";
```

Sally is assigned this role using a credential, signed by `KRtrader`, authorising `Ksally`. In practice, if Joe is a member of the `KRtrader` and is permitted to further delegate the associated permissions, then Joe could authorise Sally to be in the `KRsales` role. △

A disadvantage of this more flexible and decentralised approach is that, in giving administration authority to individual users, it provides only limited control of how these users subsequently delegate their authority; trading manager, Mandy, can decide to directly authorise salesperson, Sally, to setlimit, regardless of the intended role hierarchy. In [63], we describe how distributed workflow rules supported by WebCom are used to place constraints on the delegation actions of such users.

8.4.3 KeyStar

KeyStar is the administrative update system for WebCom$_{DAC}$ and is shown in Figure 8.24. It is implemented using a client/server architecture. The middleware systems that support KeyStar implement a simple client API that listens for update orders from the KeyStar server.

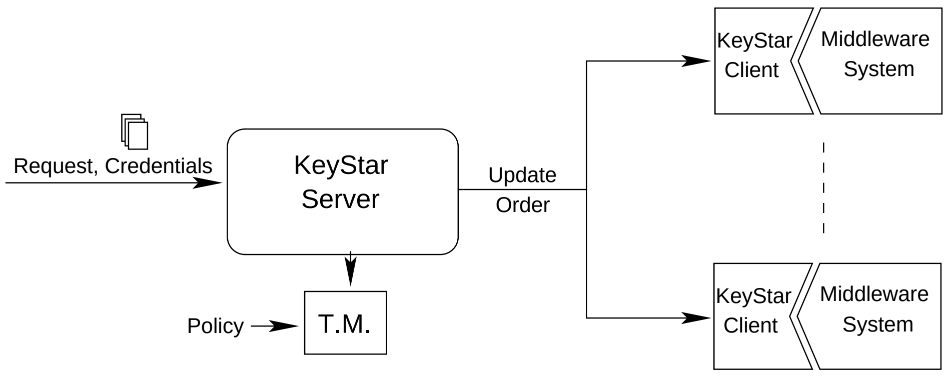


Figure 8.24: The KeyStar Architecture

The KeyStar server listens for administrative requests from WebCom$_{DAC}$. When a request is received, it is accompanied by trust management credentials that should authorise the request. These credentials are used as part of a query to the trust management system to ensure the request is authorised according to KeyStar's policy. If the request is authorised, then an update order is sent to the relevant KeyStar client(s). When this order is received, the middleware authorisation policy is updated to reflect the order. For example, the KeyNote credential shown in Figure 8.25, authorises the user `jsmith` to add new users on the server `ceres.ucc.ie`.

```
KeyNote-Version: 2
Comment: Authorises jsmith to Add Users on Ceres
Local-Constants: manager =
            "rsa-base64:MGA0GCSGSIb3DQEAQUAA4ADCBiQKBg\
            Tk162GCQWJc5gyAOuZrXaHZp2QIDAQAB"
                 jsmith =
            "rsa-base64:MIGfMA0GCSqIb3DQEBAQUAGNADCBiQ\
            YhDrrHn/eJqQXFRYY8h0BdNfJQIDAQAB"
Authorizer: manager
Licensees: jsmith
Conditions: App_Domain == "KeyStar" &&
            (Domain == "ceres.ucc.ie/BUILTIN" &&
             Task == "AddUser");
Signature: ...
```

Figure 8.25: A KeyNote credential used by KeyStar.

Implementations of KeyStar clients exist for a number of middleware systems, including Microsoft's DCOM/.Net (KeyCOM) and EJBs (KeyBean). There are a number of standard RBAC-based administrative requests that are supported including:

- adding new users to the system;
- adding new roles to the system;
- adding a user to a role;
- assigning a permission to a role.

These administrative requests are used to support administrative workflows that are coordinated by WebCom$_{DAC}$.

8.4.4 Stacked Authorisation

A Secure WebCom environment uses KeyNote to help manage trust relationships with other Secure WebCom environments. This approach requires the WebCom environment to be trusted in the sense that the security mediation (authorisation) is done by the WebCom environment and not the underlying operating system. An advantage of this approach is that, since it is independent of the security architecture of the underlying system, then it provides a better opportunity for interoperation between heterogeneous platforms that run the WebCom environment. However, since it does not rely on the underlying operating system and/or middleware authorisation mechanisms, a result is that it increases the software in the trusted computing base.

In this section we address this issue by considering how the security mechanisms of the underlying middleware and/or operating system can be used to provide the basis of security mediation and form a part of the overall WebCom security architecture. This provides a stack of security layers, as depicted in Figure 8.26. Note that Level 3 security corresponds to mechanisms encoded within the condensed graph that is used to coordinate the application components. This is examined in Chapter 7.

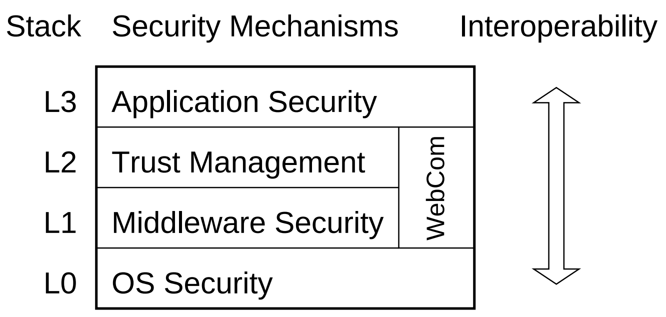


Figure 8.26: Stacked Security Architecture in WebCom$_{DAC}$

These stacked layers of secure WebCom are 'pluggable' in the sense of [96, 157]; for example, in the absence of CORBASec support for a particular ORB, a WebCom environment could be configured so that authorisation is based only on a combination of KeyNote (trust management) and underlying operating system policy. A Secure WebCom environment can automatically convert middleware RBAC policies to their equivalent KeyNote policies/credentials, and vice-versa. This provides a high degree of policy interoperability, between the middleware and trust management layer, and also within the different middleware. In addition to providing a uniform way of specifying RBAC policies for different middleware systems, it also becomes possible to enforce standardised RBAC middleware policies across middleware systems that do not have a configured RBAC policy.

As one may not have access to the source code of these legacy systems, it is not always possible to change their security policies. WebCom provides the ability to enforce a different security policy at a higher layer. The same argument can be made for fault tolerance and load balancing policies. This architecture provides a clear separation of functional and control code.

8.4.5 Discussion

WebCom$_{DAC}$ provides a framework to support automated administration using workflows. In Section 8.3, we examined GridAdmin, an administration toolkit for Grids. In effect, GridAdmin is an extension of the WebCom$_{DAC}$ system. We could implement these administrative tasks using KeyStar, specifically "*KeyGrid*".

The WebCom$_{DAC}$ framework allows the development of business rules in an activity-centric manner. First the activities of a user are specified and the interaction between users noted. These sets of activities are then translated into condensed graph workflows. These workflow applications can be then executed (or coordinated) by WebCom.

WebCom$_{DAC}$ is used to provide a central view of different legacy systems. As each legacy system's security policy is specified, modified and enforced uniquely to that system, interoperability between these systems is a difficult prospect. WebCom$_{DAC}$ provides a higher-level view of these systems. As security policies are interrogated and a representation of those policies are formed, it is possible to use the WebCom security mechanisms to emulate the security policy of a legacy system. In this way we can take an application security policy from one system and enforce an approximation of that policy on a different system, using trust management as an intermediate language.

Naturally, this approximation is not perfect. However, providing even simple translations, such as the creation of the same users, roles and permissions on a new system is useful. The WebCom$_{DAC}$ framework provides the tools to view, modify and enforce cross-platform security policies using a single mechanism.

In [23], the authors propose a method to constrain the delegation of authority using regular expressions that are embedded into the authorisation credentials. This allows the chain of delegations to be restricted, in the sense that the ability to delegate doesn't necessarily imply the holding of the authorisation. WebCom$_{DAC}$ provides similar functionality using workflow to sequence the delegation operations. In this way, the delegation operations are performed in a well defined and controlled manner. However, this approach requires a distributed architecture unlike the credential based system described in [23].

8.5 Discussion and Evaluation

This chapter discussed several application case studies using Secure WebCom. We have described how the security architecture of WebCom can be used to support specific applications, such as micropayments, administrative workflows and Grid administration. We have examined the extensibility of WebCom and in particular the naming and security architectures.

As WebCom is designed as a modular system, it can be extended to support different application requirements. This has the advantage that specific requirements can be quickly implemented using the security system. For example, the micropayment security manager can be used by any application to provide a payment system where every execution must be paid for. This could be used as an auditing tool, where we do not attempt to recover payment, but instead analyse where and how many nodes have executed in any given domain.

Recall in Chapter 5, we argued that the WebCom architecture is inherently loosely coupled, that is, functional and security requirements are implemented separately. However, experience gained developing these case studies has shown that this is not entirely true. While in theory, information can be abstractly represented in the name of a node, in practice, it is often easier to embed this information within the implementation of the security manager. Therefore, the case studies demonstrate that custom security manager implementations are required for specific applications.

The case studies presented in this dissertation serve to evaluate the effectiveness of WebCom's security model and the software architecture. Several conclusions can be drawn from experience gained through the development and implementation of these applications. For example, representing hash-based micropayments requires implementing a specific micropayment security manager for WebCom. This security manager handles the contracts and provides the logic to ensure that the hash coins are valid. However, an extension of the KeyNote trust management systems to support a hashing function has been proposed [59]. This would remove the need for the hashing logic in the security manager. Regardless, a separate contract mechanism would have to be retained.

Interacting with Grid resources is also not as straightforward as initially believed. As Grid security is based upon the identity of the users submitting jobs. These identities allow Grid resources to determine whether, and indeed when, to execute user jobs. As WebCom identifies users in terms of public keys, user jobs can at best be identified to the granularity of the WVM submitting the Grid job. While the naming architecture is technically capable of embedding user certificates as part of a node's name, implementing such a system would again require a specific "grid job" security manager. The provision of a federated identity system [6] would help greatly towards addressing this particular problem.

This experience has resulted in the belief that any non-trivial security application would require the creation of a specific security manager for WebCom. However, WebCom supports the use of several implementations of any module. For example, the micropayment security manager could be used in conjunction with the standard trust management based security manager so that the micropayments were processed, while a standard trust management policy could also be enforced.

Part IV

Discussion and Conclusions

Chapter 9

Conclusions

In this dissertation, we have introduced the WebCom distributed computation environment, and the condensed graph computation model that it utilises. The WebCom architecture is a distributed computation environment provides the basis for secure, fault tolerant, load balanced distributed applications. The pluggable nature of the WebCom architecture allows the development of modular components. The reference implementations of the core modules can be replaced, allowing different implementations of those modules to be used.

Applications in WebCom are specified as condensed graphs, where the nodes in the graphs represent atomic actions. WebCom's modules control the scheduling of these nodes. This provides a clear separation of function and control code. This separation of concerns is not security specific; every module in WebCom enjoys the same advantage. This allows policies to be written independently of the functional code.

From a security standpoint, the architecture of WebCom prompts some interesting challenges. The security requirements of WebCom are managed by the Security Manager Module. Different implementations of the security manager can be used to enforce different types of access control. As WebCom is a distributed environment, enforcement of the security policy must also be distributed. WebCom can exist outside of the control of a single administrator. Instances of WebCom running on different resources can have different administrators. The security system must support this type of architecture.

Developing a security architecture for WebCom necessitated the investigation of the nature of condensed graph applications. In this dissertation, we have argued that in order to properly specify security policies for nodes in a condensed graph, we first need to properly name these nodes.

Naming condensed graph nodes requires capturing the contextual details that describe these nodes. This context includes details such as the function of the node, the domain where it is executing, and the history of the node. Using this context, naming policies can be constructed that can be used by enforcement mechanisms to help direct the scheduling and execution of these nodes.

Chapter 6 explored naming issues and developed a model for naming in WebCom.

The WebCom naming model can also be used to support the other WebCom modules. For example, using WebCom names in the fault tolerance module to refer to WVMs would allow more contextual information about these WVMs to be provided. As WVMs were referenced by their IP address in the original implementation of WebCom, only one instance of WebCom on a single resource was permitted. The greater contextual information available within WebCom names allows multiple instances of WVMs on a single resource.

Reduction rules are used to take complex names and remove unnecessary information. We use reduction rules to help create specific policies, such as history-based policies. For example, we can use history-based policies to store the names of the domains that the nodes have executed during the computation.

We investigated WebCom's security architecture in Chapter 7. Secure WebCom provides the ability to specify access control policies in terms of WebCom names. These security policies are enforced by WebCom's security manager. We argue that if sufficient context is provided within the names, then a wide variety of authorisation requirements can be captured as access control policies.

Finally, we explored some extensions to WebCom in Chapter 8. These case studies demonstrate the extensibility of the Secure WebCom architecture and provide some practical examples of applications that can be developed using WebCom.

9.1 Results and Contributions

There are four main contributions contained within this dissertation. We have defined a naming architecture for condensed graphs, that specifies the contextual detail required to properly name a distributed component. Using this naming architecture, we have developed an access control-based security architecture for WebCom that allows application developers to specify security constraints regarding their applications. This architecture has been implemented in terms of a software architecture that support names in practice. Finally, we have developed a number of case studies that examine the capabilities of WebCom and explore some of the advantages of WebCom's security architecture.

9.2 Limitations and Future Work

The naming architecture described in this dissertation suffers a number of limitations. Primarily, as the naming model is not formally specified, no consistency guarantees can be made. Furthermore, we have seen in Chapter 6 that when multiple reduction rules are created and used in conjunction can contradict the goal of the policy, even though they operate correctly when used separately. The order that reduction rules are applied to names must be carefully controlled by the developer. There is no current means to address reduction rule inconsistencies.

A topic of future research would be to develop a formal model for WebCom names. This model could provide proofs of name consistency and completeness. Such a model could provide assurances about the application of reduction rules and the order that they are applied.

Identity management is another limitation in the current Secure WebCom prototype. Each WebCom domain has its own view of the names of entities. Federation of identity allows entities in different domains to have a common point of reference. Implementing some form of federated identity management [6] would help support large, cross-domain applications, particularly when considering Grid applications for WebCom.

Federation of identity allows each resource in the system to have an assembled identity that the entire distributed system can use to refer to that resource. For example, a user can be a manager in one domain and a clerk in another. Each domain refers to the user in the context they support. With a federated identity, when the two domains are discussing that user, they both know which user the other is referring to.

While the Naming architecture described in Chapter 6 provides a framework that could be used to store the necessary contextual detail to identify resources, it does so from a local perspective. Identity federation allows the linking of these local perspectives into a global identity. WebCom names could be used to provide this link. It is envisioned that WebCom will be used on the Grid [123]. Supporting an existing federation scheme would promote integration with existing infrastructures and helps gain acceptance within the global community.

WebCom provides a messaging service that is independent of the condensed graph model. This messaging service provides the ability for both WVMs and the component modules to communicate and perform tasks on request. Messaging is possible both internally to a WVM and between WVMs.

Each module in WebCom defines the messages it can handle. For example, the security manager module could define a message that allows remote querying of its security policy. Remote querying has the advantage that a parent can ask its children that, in the case they were sent a particular node, would they allow it to be executed.

However, this can have security implications. For example, if a WVM wanted to surreptitiously execute a dangerous node, it could ask each of its children until it received a positive reply. The security manager should provide the capability to vet these messages so that dangerous messages can be prevented from reaching their destinations. This aspect of WebCom is not addressed in the current access control model. It is foreseen that some form of `mayCommunicate` check could be defined in terms of the WebCom access control model. This is a potential avenue for future research.

Another important topic of future research is the provision of a public key infrastructure (PKI) for WebCom. In its current form, WebCom uses an ad hoc method to distribute both identity and authorisation certificates. It is envisioned that a PKI architecture should be constructed to provide a distribution mechanism for such certificates. PKIs also support concepts such as certificate revocation, which is also not currently addressed.

We suggest that a PKI system could be constructed on top of WebCom. A WebCom could implement a certificate query system. This system could use, for example, a Chord [168] like peer-to-peer structure to maintain the certificates across the distributed network. WebCom's messaging system could be utilised to support the storage and retrieval of the certificates. PKI queries could be expressed in terms of condensed graph applications. However, developing a PKI infrastructure for any architecture is a significant task.

Part V

Appendices

Appendix A

WebCom Names XML Definition

The following the the XML definition of a WebCom name. These representations are used to define names for nodes a priori. The Naming Manager reads in these names and uses them during graph execution.

```
<!-- cg.dtd - an XML Document Type Declaration for XML documents           -->
<!-- describing sets of Condensed Graphs.                                  -->
<!-- Copyright (C) 2003  The Centre for Unified Computing                  -->

<!-- DOCUMENT HISTORY -->
<!-- Date         Person              Action                               -->
<!-- 06/3/2004    Philip Healy        Created                              -->

<!-- secname:securename element -->
<!ELEMENT secname:securename
          (secname:domain?, secname:graph?, secname:operand*,
   secname:operator?, secname:destination*)>
<!ATTLIST secname:securename
xmlns:secname CDATA #FIXED "http://cuc.ucc.ie/xml/secname">

<!-- secname:domain element -->
<!ELEMENT secname:domain EMPTY>
<!ATTLIST secname:domain name CDATA #REQUIRED>

<!-- secname:graph element -->
<!ELEMENT secname:graph EMPTY>
<!ATTLIST secname:graph name CDATA #REQUIRED>

<!-- secname:operand element -->
<!ELEMENT secname:operand EMPTY>
<!ATTLIST secname:operand name CDATA #REQUIRED>
```

```
<!-- secname:operator element -->
<!ELEMENT secname:operator EMPTY>
<!ATTLIST secname:operator name CDATA #REQUIRED>

<!-- secname:destination element -->
<!ELEMENT secname:destination EMPTY>
<!ATTLIST secname:destination name CDATA #REQUIRED>
```

Appendix B

Naming System for the ShareTrader Application

B.1 Generating Names for ShareTrader Nodes

```
public class ShareTraderNameGenerator extends NameGenerator
{
  public SecureName generateNameFromNode(CondensedGraph cg, Node node,
                                         ReductionRule reduxrule,
                                         String Domain) {
    SecureName nodename = new SecureName();
    CGExaminer examiner = new CGExaminer(cg);
    int NodeID = examiner.getNodeID(node);

    nodename.setDomain(Domain);
    nodename.setFunction(examiner.getOperator(NodeID));
    nodename.setGraph(examiner.getName());

    int destports = examiner.getNumDestinationPorts(NodeID);
    for (int i = 0; i < destports; i++) {
      int[] destids = examiner.getNodeDestinationIDs(NodeID);
      for (int j = 0; j < destids.length; j++) {
        String operator = examiner.getOperator(destids[j]);
        nodename.addDestination(operator);
      }
    }

    Vector NodeOperands = examiner.getNodeOperands(NodeID);
    int[] inpnodeids = new int[NodeOperands.size()];
    int pos = 0;
    for (Iterator iter = NodeOperands.iterator(); iter.hasNext(); ) {
      Object[] item = (Object[]) iter.next();
      if (item[0] instanceof String) {
        String type = (String) item[0];
        if ( (type.equals("NodeID")) && (item[1] instanceof Integer))
          inpnodeids[pos] = ( (Integer) item[1]).intValue();
        else if (type.equals("Value"))
          inpnodeids[pos] = -1;
        else {
          throw new webcom.cgengine.InvalidIDException(
              "Invalid Operand type in Node ID " + NodeID +
              " Operand " + pos);
        }
        pos++;
      }
    }

    if (inpnodeids.length > 0) {
      for (int j = 0; j < inpnodeids.length; j++) {
        if (inpnodeids[j] == -1) // Node is a value. {
          Object[] item = (Object[]) NodeOperands.elementAt(j);
          String type = (String) item[0];
          if (type.equals("Value")) {
            if (item[1] instanceof Trade) {
              Trade trd = (Trade) item[1];
              nodename.addInput("" + trd.getTotal());
            }
            nodename.addInput(item[1].toString());
          }
          else
            throw new webcom.cgengine.InvalidIDException(
                "Invalid Node ID " + inpnodeids[j] + " in Node ID "
                + NodeID + "’s Operand " + j);
        }
        else
          nodename.addInput(examiner.getOperator(inpnodeids[j]));
      }
    }
    nodename.reduceName(reduxrule); // Apply the Reduction Rule specified.
    return (nodename);
  }
}
```

B.2 Reduction Rules for ShareTrader Nodes

```
public class ShareTraderReductionRule extends ReductionRule {
  public SexpList reduce(SexpList Name) {
    SexpList Function = null;
    SexpList Inputs = null;

    if (Name == null)
      return null;
    int pos = 0;
    for (Iterator i = Name.iterator(); i.hasNext(); ) {
      if (pos == 0) {
        pos++;
        i.next();
      }

      SexpList part = (SexpList) i.next();
      if ( (part.getType()).compareTo("Function") == 0)
        Function = part;
      else if ( (part.getType().compareTo("Inputs") == 0))
        Inputs = part;
      pos++;
    }

    Sexp list[] = new Sexp[Function.size() + Inputs.size()];
    pos = 0;
    for (Iterator iter = Function.iterator(); iter.hasNext(); ) {
      SexpString item = (SexpString) iter.next();
      list[pos] = item;
      pos++;
    }
    for (Iterator iter = Inputs.iterator(); iter.hasNext(); ) {
      Sexp item = (Sexp)iter.next();
      list[pos] = item;
      pos++;
    }
    SexpList newname = new SexpList(new SexpString("WebComName"), list);
     return (newname);
  }
}
```

Afterword

The story of the writing of this Ph.D. thesis is rather bizarre. As I mentioned in the dedication, my maternal Grandfather passed away during the writing of this thesis. I spend many interesting days sitting by his hospital bed in the weeks before his death. He had some wonderful stories, and many of my funniest stories involve him. He interrogated me on many occasions about when he would see this thesis.

After he passed away, on August 23^{rd} 2005, I returned to the full time writing of this Thesis. One month later, during the anniversity service, my parent's house in Galway was broken into. The thieves were disturbed during the theft, and only made off with some cash, a playstation and the laptop I was writing this thesis upon. They also took all of the backups. These included two usb-pendrives, a usb harddrive and a recent printout. As I had been living in Galway for nearly three months at this point, I was only backing up the thesis locally, in case of hardware failure. The most recent remote copy I was left with was a three months old copy in Cork.

The Gardaí were not hopeful about the return of the laptop. Not willing to give up, we contacted the local radio station and they kindly made an announcement requesting the return of the laptop. A local paper, the Galway Sentinal, called and asked if they could help. They wrote an article about the theft, and left a contact number where we could be reached. The following day, someone called, stating they had "purchased" the laptop for €1000 and would "sell" it back to us for that price.

My mother, Áine, haggled with the thieves and brought the price down significantly. Eventually, after many phonecalls, a meet was arranged where the money was to be swapped for the laptop. As I was in Cork at the time, my mother and brother, Cormac, went to a graveyard to meet the thieves. The laptop was retrieved and given to the Gardaí for fingerprinting. I have some advice in this regard: Never let your LCD screen be fingerprinted, the dust is next to impossible to remove! I am extremely grateful to both my mother and my brother for their efforts on my behalf. At this point my mother believes that she deserves the Ph.D. more than I do!

Bibliography

[1] *Advogato's trust metric*. http://www.advogato.org/trust-metric.html.

[2] Apache-ssl release version 1.3.6/1.36. Open source software distribution. `http://www.apache.org`.

[3] *ClimatePrediction.net*. http://www.climateprediction.net/.

[4] *Distributed.net*. http://www.distributed.net/.

[5] *The JSDSI Project*. http://jsdsi.sourceforge.net/.

[6] *Liberty Alliance*. http://www.projectliberty.org/.

[7] *The Message Passing Interface (MPI) standard*. http://www-unix.mcs.anl.gov/mpi/.

[8] The Object Management Group. http://www.omg.org.

[9] *Parallel Virtual Machine (PVM)*. http://www.csm.ornl.gov/pvm/pvm_home.html.

[10] *The Cosmogrid Project*, 2004. http://www.cosmogrid.ie/.

[11] M. Abadi, A. Birrell, and T. Wobber. Access control in a world of software diversity. In *Proceedings of the Tenth Workshop on Hot Topics in Operating Systems*. USENIX, June 2005.

[12] D. A. Adams. *A computational model with dataflow sequencing*. PhD thesis, Stanford, California, 1968. TR/CS-117.

[13] Formula One Administration. Formula 1 live timing java applet. http://www.formula1.com/archive/grandprix/livetiming/popup/757/8.html.

[14] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

[15] P. Ammann and R. S. Sandhu. The extended schematic protection model. *Journal of Computer Security*, 1(3–4):335–384, 1992.

[16] M. Blaze and. The role of trust management in distributed systems security. In *Secure Internet Programming: Issues in Distributed and Mobile Object Systems*. Springer-Verlag Lecture Notes in Computer Science, 1999.

[17] R. Anderson, H. Manifavas, and C. Sutherland. Netcard - a practical electronic cash system. In *Cambridge Workshop on Security Protocols*, 1995.

[18] Arvind and K. P. Gostelow. A computer capable of exchanging processors for time. In *Proceedings of IFIP Congress 1977*, pages 849–853, Toronto, Canada, August 1977.

[19] V. Atluri, S. Ae Chun, and P. Mazzoleni. Chinese wall security for decentralized workflow management systems. *Journal of Computer Security*, 12(6):799–840, 2004.

[20] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641., August 1978.

[21] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. A domain and type enforcement UNIX prototype. In *Proceedings of the Fifth Usenix UNIX Security Symposium*, Salt Lake City, Utah, USA., June 5–7 1995. USENIX.

[22] D. Balenson. Privacy enhancement for internet electronic mail: Part III: Algorithms, modes and identifiers. Request for Comment (RFC) 1423, Internet Engineering Task Force, February 1993.

[23] O. Bandmann, M. Dam, and B. S. Firozabadi. Constrained delegation. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 131–140, Oakland, CA, USA, May 2002. IEEE.

[24] A. Baratloo, M. Karul, Z. Kedem, and P. Wyckoff. Charlotte: metacomputing on the web. In K. Yetongnon and S. Hariri, editors, *9th International Conference on Parallel and Distributed Computing Systems*, Dijon, France, September 25–27 1996.

[25] D. E. Bell and L. J. La Padula. Secure computer system: unified exposition and MULTICS interpretation. Report ESD-TR-75-306, The MITRE Corporation, March 1976.

[26] E. Bertino, B. Catania, G. Guerrini, M. Martelli, and D. Montesi. A bottom-up interpreter for a database language with updates and transactions. In M. Alpuente R. Barbuti and I. Ramos, editors, *1994 Joint Conference on Declarative Programming*, volume II, pages 206–220, Peniscola, Spain, September 1994.

[27] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153 Rev 1 (ESD-TR-76-372), MITRE Corp Bedford MA, 1976.

[28] B. Blakley. *Corba Security. An Introduction to Safe Computing with Objects.* Object Technology Series. Addison-Wesley, 2000.

[29] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote trust management system version 2. September 1999. Internet Request For Comments 2704.

[30] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. Using the KeyNote trust management system. `http://www.crypto.com/trustmgt`, December 1999.

[31] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The role of trust management in distributed systems security. In Jan Vitek and C.tI. Jensen, editors, *Security Issues for Mobile and Distributed Objects*, Fourth International Workshop, MOS'98, Brussels, Belgium, July 1998. Springer-Verlag Inc.

[32] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the Symposium on Security and Privacy*. IEEE Computer Society Press, 1996.

[33] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance checking in the policymaker trust management system. In *Proceedings of the 2nd Financial Cryptography Conference*. Springer Verlag LNCS, 1998.

[34] M. Blaze, J. Ioannidis, and A. D. Keromytis. Trust management and network layer security protocols. In *Security Protocols International Workshop*. Springer Verlag LNCS, 1999.

[35] M. Blaze, J. Ioannidis, and A. D. Keromytis. Offline micropayments without trusted hardware. In *Financial Cryptography*, Grand Cayman, February 2001.

[36] W.E. Bobert and R.Y. Kain. A practical alternative to hierarchical integrity properties. In *Proceedings of the National Computer Security Conference*, pages 18–27, 1985.

[37] J. Boly, A. Bosselaers, R. Cramer, R. Michelsen, S. F. Mjolsnes, F. Muller, T. P. Pedersen, B. Pfitzmann, P. de Rooij, B. Schoenmakers, M. Schunter, L. Vallee, and M. Waidner. The ESPRIT project CAFE - high security digital payment systems. In *ESORICS*, pages 217–230, 1994.

[38] M. Branchaud. A survey of public key infrastructures. Master's thesis, McGill University, Montreal, Quebec, Canada., 1997.

[39] S. A. Brands. *Rethinking Public Key Infrastructures and Digital Certificates; Building in Privacy*. The MIT Press, Cambridge, Massachusetts, 2000.

[40] D. F. C. Brewer and M. J. Nash. The Chinese Wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214. IEEE Computer Society Press, May 1989.

[41] CCITT Draft Recomendation. *The Directory Authentication Framework, Version 7*, November 1987.

[42] Y. Chu. Trust management for the world wide web. Master's thesis, Massachusetts Institute of Technology, June 1997.

[43] Y. Chu, P. DesAutels, B. LaMacchia, and P. Lipp. PICS signed labels (dsig) 1.0 specification. Technical report, World Wide Web Consortium, May 1998. http://www.w3.org/TR/REC-DSig-label.

[44] Y. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. Referee: Trust management for web applications. In *Sixth International World Wide Web Conference*, Santa Clara, California, USA, April 1997. http://www.farcaster.com/papers/www6-referee/www6-referee.htm.

[45] M. J. Ciaraldi, D. Finkel, and C. E. Wills. Risks in anonymous distributed computing systems. In *Proceedings of the International Network Conference*, Plymouth, UK, July 2000.

[46] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security models. In *Proceedings Symposium on Security and Privacy*, pages 184–194. IEEE Computer Society Press, April 1987.

[47] D. Clarke, J. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, September 2001.

[48] B. C. Clayton, T. B. Quillinan, and S. N. Foley. Automating security configuration for the grid. *Journal of Scientific Programming*, 13(2):113–125, 2005.

[49] S. Cluet, O. Kapitskaia, and D. Srivastava. Using LDAP directory caches. In *PODS '99: Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems*, pages 273–284, New York, NY, USA, 1999. ACM Press.

[50] Microsoft Corporation. Microsoft Kerberos specification. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secauthn/security/microsoft_kerberos.asp, July 2005.

[51] N. Dershowitz. A taste of rewriting. In P. Lauer and J. Zucker, editors, *Functional Programming, Concurrency, Simulation and Automated Reasoning.*, volume 693, pages 199–228. Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1993.

[52] M. Donnelly. *An Introduction to LDAP*, April 2000. http://www.ldapman.org/articles/intro_to_ldap.html.

[53] E. Dulaney, V. Sankar, and S. E. Sankar. *Integrating Unix and NT Technology*. 29th Street Press, June 1999.

[54] C. Ellison. SPKI requirements. Request for Comment (RFC) 2692, Internet Engineering Task Force, September 1999.

[55] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI examples. Internet draft, Internet Engineering Task Force, 1998.

[56] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. Request for Comment (RFC) 2693, Internet Engineering Task Force, September 1999.

[57] L. Demailly et. al. *Safe-TCL*. Sun Microsystems Inc. http://www.demailly.com/tcl/plugin/safetcl.html.

[58] S. N. Foley. Building Chinese Walls in standard Unix. *Computers and Security Journal*, 16(6):551–563, December 1997.

[59] S. N. Foley. Using trust management to support transferable hash-based micropayments. In *Proceedings of the 7th International Financial Cryptography Conference*, Gosier, Guadeloupe, FWI, January 2003.

[60] S. N. Foley and J. L. Jacob. Specifying security for CSCW systems. In *Proceedings of the Computer Security Foundations Workshop*, pages 136–145, Kenmare, Co. Kerry, Ireland, June 1995. IEEE Computer Society.

[61] S. N. Foley and J.L. Jacob. Specifying security for computer supported collaborative working. *Journal of Computer Security*, 3(4):233–254, 1994/1995.

[62] S. N. Foley and J.P Morrison. Computational paradigms and protection. In *ACM New Computer Security Paradigms*, Cloudcroft, NM, USA, 2001. ACM Press.

[63] S. N. Foley, B. P. Mulcahy, and T. B. Quillinan. Dynamic administrative coalitions with webcom *DAC*. In *WeB2004 The Third Workshop on e-Business*, Washington D.C., USA, December 2004.

[64] S. N. Foley and T. B. Quillinan. Using trust management to support micropayments. In *Proceedings of the Second Information Technology and Telecommunications Conference*, pages 219–223, Waterford Institute of Technology, Waterford, Ireland., October 2002. TecNet.

[65] S. N. Foley, T. B. Quillinan, and J. P. Morrison. Secure component distribution using webcom. In *Proceeding of the 17th International Conference on Information Security (IFIP/SEC 2002)*, Cairo, Egypt, May 2002.

[66] S. N. Foley, T. B. Quillinan, J. P. Morrison, D. A. Power, and J. J. Kennedy. Exploiting KeyNote in WebCom: Architecture neutral glue for trust management. In *Proceedings of the Nordic Workshop on Secure IT Systems Encouraging Co-operation*, Reykjavik University, Reykjavik, Iceland, October 2000.

[67] S. N. Foley, T. B. Quillinan, M. O'Connor, B. P. Mulcahy, and J. P. Morrison. A framework for heterogeneous middleware security. In *Proceedings of the 13th International Heterogeneous Computing Workshop*, Santa Fe, New Mexico, USA., April 2004. IPDPS.

[68] S. N. Foley and H. Zhou. Authorisation subterfuge by delegation in decentralised networks. In *Proceedings of International Security Protocols Workshop*. Springer Verlag LNCS, April 2005.

[69] B. Fonseca. VeriSign issues false Microsoft digital certificates. http://www.infoworld.com/articles/hn/xml/01/03/22/010322hnmicroversign.html, March 2001. Infoworld.

[70] Internet Engineering Task Force. Public key infrastructure (x.509) [PKIX]. http://www.ietf.org/html.charters/pkix-charter.html.

[71] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[72] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *5th ACM Conference on Computer and Communications Security*, 1998.

[73] International DOI Foundation. Digital object identifiers. http://www.doi.org/.

[74] L. J. Fraim. Scomp: a solution to the multilevel security problem. *IEEE Computer*, 16(7):126–143, July 1983.

[75] A. Gilbert, A. Abraham, and M. Paprzycki. A system for ensuring data integrity in grid environments. 2004.

[76] D. Gollmann. What do we mean by entity authentication? *Proceedings of the Symposium on Security and Privacy*, pages 46–54, May 1996.

[77] D. Gollmann. *Computer Security*. Wiley, 1^{st} edition, 1999. ISBN: 0-471-97844-2.

[78] L. Gong. *Inside Java^{TM} 2 Platform Security*. The Java^{TM} Series. Addison Wesley, June 1999. ISBN: 0-201-31000-7.

[79] L. Gong et al. Going beyond the sandbox: An overview of the new security architecture in the java development kit 1.2. In *USENIX Symposium on Internet Technologys and Systems*, pages 103–112, 1997.

[80] T. Grandison and M. Sloman. A survey of trust in internet applications. *IEEE Communications Surveys*, December 2000. http://www.comsoc.org/livepubs/surveys/public/2000/dec/grandison.html.

[81] The Object Management Group. Common object request broker architecture (corba/iiop). Technical report, The Object Management Group, December 2002. http://www.omg.org/technology/documents/formal/corba_iiop.htm.

[82] C. A. Gunter and T. Jim. Design of an application-level security infrastructure. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, September 1997.

[83] C. A. Gunter and T. Jim. Policy-directed certificate retrieval. *Software: Practice & Experience*, 30(15):1609–1640, September 2000.

[84] M. Harrison, W. Ruzzo, and J. Ullman. Protection in operating systems. *Communications of the ACM*, 19:461–471, 1976.

[85] P. D. Healy. *Architecture and Implementation of a Distributed Reconfigurable Metacomputer*. PhD thesis, University College Cork, April 2006.

[86] A. Herzherg and H. Yochai. Mini-pay: Charging per click on the web. In *Sixth International World Wide Web Conference*, Santa Clara, California, USA, April 7–11 1997.

[87] R. Housley et al. Internet X.509 public key infrastructure certificate and CRL profile. January 1999. Internet Engineering Task Force, Request for Comments 2459.

[88] J. R. Howell. *Naming and sharing resources across administrative boundaries*. PhD thesis, Dartmouth College, Hanover, New Hampshire, May 2000.

[89] T. Howes, S. Kille, W. Yeong, and C. Robbins. The string representation of standard attribute syntaxes. Request for Comment (RFC) 1778, Internet Engineering Task Force, March 1995.

[90] E. Huggard. JKeyNote. Fourth year computer science project, University College Cork, Ireland, April 2003. http://kargellan.ucc.ie/JKeyNote.

[91] D. Eastlake III and P. Jones. US secure hash algorithm 1 (SHA1). Request for Comments (RFC) 3174, Internet Engineering Task Force, September 2001.

[92] Netscape Inc. Secure sockets layer website. Technical Brief: http://home.netscape.com/security/techbriefs/ssl.html.

[93] INRIA Rocquencourt, projet Cristal. The Caml homepage. http://pauillac.inria.fr/caml/index-eng.html.

[94] Internet Engineering Task Force. Simple public key infrastructure (SPKI). http://www.ietf.org/html.charters/spki-charter.html.

[95] J. Ioannidis et al. Fileteller: Paying and getting paid for file storage. In *Proceedings of Financial Cryptography*, March 2003.

[96] N. Itoi and P. Honeyman. Pluggable authentication modules for Windows NT. In *Proceedings of the 2nd USENIX Windows NT Symposium*, pages 97–108, Seattle, Washington, August 1998.

[97] T. Jim. Sd3: a trust management system with certified evaluation. In *Proceeding of the IEEE Symposium on Security and Privacy*, May 2001.

[98] M. P. Jones. Hugs 1.3, the Haskell user's Gofer system: User manual. Technical Report NOTTCS-TR-96-2, Department of Computer Science, Nottingham University, Nottingham NG7 2RD, UK, 1996.

[99] B. S. Kaliski Jr. A layman's guide to a subset of ASN.1, BER, and DER. Technical report, RSA Laboratories, November 1993. ftp://ftp.rsa.com/pub/pkcs/ascii/layman.asc.

[100] B. Kaliski. Privacy enhancement for internet electronic mail: Part IV: Key certification and related services. Request for Comment (RFC) 1424, Internet Engineering Task Force, February 1993.

[101] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The eigentrust algorithm for reputation management in P2P networks. In *In Proceedings of the Twelfth International World Wide Web Conference (WWW2003)*, Budapest, Hungary, May 20-24 2003. ACM Press.

[102] R.M. Karp and R.E. Miller. Properties of a model for parallel computations:determinacy, temination, queueing. *SIAM Journal of Applied Mathematics*, 14(6):1390–1411, November 1966.

[103] M. Karul. *Metacomputing and resource allocation on the world wide web*. PhD thesis, New York University, May 1998.

[104] J. J. Kennedy. *Design and Implementation n-tier Metacomputer with Decentralised Fault Tolerence*. PhD thesis, University College Cork, Ireland, 2004.

[105] S. Kent. Privacy enhancement for internet electronic mail: Part II: Certificate-based key mangement. Request for Comment (RFC) 1422, Internet Engineering Task Force, February 1993.

[106] V. Kessler. On the Chinese Wall model. In *European Symposium on Research in Computer Security*, pages 39–54. Springer Verlag, LNCS 875, 1992.

[107] S. Kille. String representation of distinguished names. Request for Comment (RFC) 1779, Internet Engineering Task Force, March 1995.

[108] P. Krugman. *The return of Depression Economics*. WW Norton & Co, 1999. 176 pages.

[109] T. Lampinen. Using SPKI certificates for authorization in CORBA based distributed object-oriented systems. In *4th Nordic Workshop on Secure IT systems (NordSec '99)*, pages 61–81, Kista, Sweden, November 1999.

[110] B. Lampson. Protection. *ACM Operating Systems Review*, 8, 1974.

[111] U. Lang. *Access Policies for Middleware*. PhD thesis, University of Cambridge, Wolfson College, Cambridge, UK., May 2003.

[112] N. Li. Local names in SPKI/SDSI. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 2–15. IEEE Computer Society Press, July 2000.

[113] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust-management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, number ISSN: 1081-6011 ISBN: 0-7695-1543-6, pages 114–130, Oakland, CA, USA, 2002. IEEE.

[114] J. Linn. Privacy enhancement for internet electronic mail: Part I: Message encryption and authentication procedures. Request for Comment (RFC) 1421, Internet Engineering Task Force, February 1993.

[115] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track of the 2001 USENIX Annual Technical Conference.*, Boston, MA, USA., June 2001.

[116] N. McBurnett. PGP web of trust statistics, 1997. http://bcn.boulder.co.us/ neal/pgpstat/.

[117] Microsoft Corporation. *Microsoft Passport*. http://www.passport.net/.

[118] Microsoft Corporation. *Microsoft Platform SDK. The COM Library. Microsoft Developer Network.*, 0.9 edition, October 1995. http://www.msdn.microsoft.com.

[119] J. K. Millen and R. N. Wright. Reasoning about trust and insurance in a public key infrastructure. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW'00)*, pages 16–23, Cambridge, England, July 03–05 2000.

[120] J. G. Mitchell, J. J Gibbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia. An overview of the Spring system. In *Compcon Spring '94, Digest of Papers.*, pages 122–131, 28th February – 4th March 1994.

[121] P. Mockapetris. Domain names - concepts and facilities. Request for Comment (RFC) 1034, Internet Engineering Task Force, November 1987.

[122] J. P. Morrison. *Condensed Graphs: Unifying Availability-Driven, Coercion-Driven and Control-Driven Computing*. PhD thesis, Eindhoven, 1996.

[123] J. P. Morrison, B. Clayton, A. Patil, and S. John. The information gathering module of the WebCom-G operating system. In *Proceedings of the Second International Symposium on Parallel and Distributed Computing (ISPDC03)*, Ljubljana, Slovenia, October 2003.

[124] J. P. Morrison, B. Clayton, D. A. Power, and A. Patil. s: Grid enabled metacomputing. *The Journal of Neural, Parallel and Scientific Computation*, Special Issue on Grid Computing, 2004.

[125] J. P. Morrison and R. Connolly. Facilitating Parallel Programming in PVM using Condensed Graphs. Proceedings of EuroPVM'99: Universitat Autonoma de Barcelona, Spain. 26-29 Sept 1999.

[126] J. P. Morrison et al. Architectural neutral glue for COM objects. Internal Note, Center for Unified Computing, University College, Cork, Ireland, 2000.

[127] J. P. Morrison and P. D. Healy. Implementing the WebCom 2 distributed computing platform with XML. In *IEEE Proceeding of the International Symposium on Parallel and Distributed Computing*, 2002.

[128] J. P. Morrison, P. D. Healy, and P. J. O'Dowd. Architecture and implementation of a distributed reconfigurable metacomputer. In *Proceedings of the Second International Symposium on Parallel and Distributed Computing (ISPDC 2003)*, pages 153–158, Ljubljana, Slovenia, October 2003.

[129] J. P. Morrison, P. D. Healy, D. A. Power, and K. J. Power. The role of XML within the WebCom metacomputing platform. *Scalable Computing: Practice and Experience*, 6(1), 2005.

[130] J. P. Morrison and D. A. Power. Master promotion and client redirection in the webcom system. In *PDPTA, Las Vegas USA*, 2000.

[131] J. P. Morrison, D. A. Power, and J. J. Kennedy. A Condensed Graphs Engine to Drive Metacomputing. Proceedings of the international conference on parallel and distributed processing techniques and applications (PDPTA '99), Las Vegas, Nevada, June 28 - July1, 1999.

[132] J. P. Morrison, D. A. Power, and J. J. Kennedy. WebCom: A Web Based Distributed Computation Platform. Proceedings of Distributed computing on the Web, Rostock, Germany, June 21 - 23, 1999.

[133] J. P. Morrison, K. Power, and N. Cafferkey. Cyclone: A cycle brokering system to harvest wasted processor cycles. In *Parallel and Distributed Computing Techniques and Applications*, Las Vegas, NV, USA, June 2000.

[134] B. P. Mulcahy, S. N. Foley, and J. P. Morrison. Cross cutting condensed graphs. In *Proceedings of the 2005 International Conference on Parallel and Distributed Processing Techniques and Applications*, Monte Carlo Resort, Las Vegas, Nevada, USA, June 27-30 2005. PDPTA.

[135] M. J. Nash and K. R. Poland. Some conundrums concerning separation of duty. In *Proceedings of the Symposium on Security and Privacy*, pages 201–207, Oakland, CA, May 1990. IEEE Computer Society Press.

[136] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[137] M. N. Nelson and S. R. Radia. A uniform name service for spring's unix environment. In *Proceedings of the Winter 1994 USENIX Conference*. USENIX, January 1994.

[138] B. C. Neumann and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, September 1994.

[139] Massachusetts Institute of Technology (MIT). Project athena. http://web.mit.edu/release/www/index.html.

[140] Object Management Group (OMG). Naming service specification. Technical Report Version 1.3, OMG, October 2004. http://www.omg.org/cgi-bin/doc?formal/04-10-03.

[141] S. Ó'Tuairisg, M. Browne, J. Cunniffe, A. Shearer, J. Morrison, and K. Power. WebCom-G: Implementing an astronomical data analysis pipeline on a Grid-type infrastructure. In P. L. Shopbell, M. C. Britton, and R. Ebert, editors, *ASP Conf. Ser. Astronomical Data Analysis Software and Systems XIV*. Publications of the Astronomy Society of the Pacific, 2005.

[142] T. P. Pedersen. Electronic payments of small amounts. In *Security Protocols Workshop*, pages 59–68, 1996.

[143] K. Pingali and Arvind. Efficient demand-driven evaluation. part 1. *ACM Transactions on Programming Languages and Systems*, 7(2):311–333, April 1985.

[144] D. A. Power. *A Framework for: Heterogeneous Metacomputing, Load Balancing and Programming in WebCom*. PhD thesis, University College Cork, Ireland, 2004.

[145] D. A. Power, A. Patil, S. John, and J. P. Morrison. WebCom-G. In *Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03)*, Las Vegas, Nevada, June 2003. CSREA Press.

[146] K. Power. *ComPeer: A Scalable, Self-organizing, peer-to-peer MetaComputer*. PhD thesis, University College Cork, Ireland, 2004.

[147] T. B. Quillinan, B. C. Clayton, and S. N. Foley. GridAdmin: Decentralising grid administration using trust management. In *Proceedings of the Third International Symposium on Parallel and Distributed Computing (ISPDC04)*, Cork, Ireland, July 2004.

[148] T. B. Quillinan and S. N. Foley. Security in WebCom: Addressing naming issues for a web services architecture. In *Proceedings of the 2004 ACM Workshop on Secure Web Services (SWS).*, Washington D.C., USA., October 2004. ACM.

[149] S. Radia. Naming policies in the Spring system. In *Proceedings of the 1st International Workshop on Services in Distributed and Networked Environments.* Sun Microsystems, Inc., IEEE, 1994.

[150] M. K. Reiter and S. G. Stubblebine. Path independence for authentication in large-scale systems. In *Proceedings of the 4th ACM conference on Computer and communications security (CCS97)*, pages 57–66. ACM Press, 1997.

[151] R. Rivest. The MD5 message-digest algorithm. Request for Comments (RFC) 1321, Internet Engineering Task Force, April 1992.

[152] R. Rivest and B. Lampson. SDSI - a simple distributed security infrastructure. In *DIMACS Workshop on Trust Management in Networks*, 1996.

[153] R. L. Rivest. S-expressions. Technical report, Network Working Group, Internet Engineering Task Force, May 1997. Internet Draft: http://theory.lcs.mit.edu/ rivest/sexp.txt.

[154] R. L. Rivest. Can we eliminate certificate revocation lists? In Rafael Hirschfeld, editor, *Proceedings of Financial Cryptography '98*, number 1465, pages 178–183. Springer Lecture Notes in Computer Science, February 1998.

[155] A. D. Rubin and D. E. Geer Jr. Mobile code security. *Internet Computing*, 2(6):30 – 34, November/December 1998. ISSN: 1089-7801.

[156] J. Sabater and C. Sierra. Social regret, a reputation model based on social relations. *SIGecom Exch.*, 3(1):44–56, 2002.

[157] V. Samar and R. Schemers. Unified login with pluggable authentication modules (PAM). Request for Comments 86.0, Open Software Foundation, October 1995.

[158] R. Sandhu et al. Role based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[159] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[160] R. S. Sandhu and P. Samarati. Access control: principle and practice. *Communications Magazine, IEEE*, 32(9):40 – 48, September 1994.

[161] B. Schneier. *Applied Cryptography*, chapter 12, pages 566–572. Wiley, second edition, 1996.

[162] B. Schneier. *Managed Security Monitoring: Closing the Window of Exposure*, 2000. http://www.counterpane.com/window.html.

[163] C. Shirky. The case against micropayments. OpenP2P: http://www.openp2p.com/pub/a/p2p/2000/12/19/micropayments.html, December 2000.

[164] A. E. K. Sobel and J. Alves-Foss. A trace-based model of the Chinese Wall security policy. In *Proceedings of the 22nd National Information Systems Security Conference*, Arlington, Va., USA, October 1999.

[165] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *Proceedings of the 8th USENIX Security Symposium*, pages 123 – 140, Washington D.C., USA, August 1999. USENIX.

[166] J. A. Stankovic and K. Ramamritham. The Spring kernel: a new paradigm for real-time operating systems. *SIGOPS Oper. Syst. Rev.*, 23(3):54–71, 1989.

[167] T. L. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings of the International Conference of Parallel Processing (ICPP)*, volume 1, Urbana-Champain, Illinois, USA, August 11–14 1995.

[168] I. Stoica, R. Morris, D. Karger, M. Frans Kaashoeka, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM 2001*, San Diego, California, USA., August 2001. ACM.

[169] Sun Microsystems. *Enterprise JavaBeans(tm) Specification, Version 2.1*, June 2003. http://java.sun.com/products/ejb/docs.html.

[170] Sun Microsystems Inc. *The Java Website*. http://java.sun.com/.

[171] A. S Tanenbaum and A. S Woodhull. *Operating Systems Design and Implementation*. Prentice Hall, 3rd edition, 2006. ISBN: 0-13-142938-8.

[172] J. Touch, L. Eggert, and Y. Wang. Use of IPsec transport mode for dynamic routing. Request for Comments (RFC) 3884, Internet Engineering Task Force, September 2004.

[173] U. S. Department of Defense. Trusted computer system criteria. Technical Report CSC-STD-001-83, U. S. National Computer Security Center, August 1983. Known as “The Orange Book”.

[174] W. Venema. TCP WRAPPER: Network monitoring, access control and booby traps. In *Proceedings of the 3rd UNIX Security Symposium*, 1999.

[175] J. von Neumann. The principles of large-scale computing machines. *IEEE Annals of the History of Computing*, 10(4):243–256, October-December 1988. ISSN: 1058-6180.

[176] The World Wide Web Consortium (W3C). The PICS project. http://www.w3.org/PICS/.

[177] The World Wide Web Consortium (W3C). Web naming and addressing. http://www.w3.org/Addressing/.

[178] M. Wahl, T. Howes, and S. Kille. Lightweight directory access protocol (version 3). Request for Comment (RFC) 2251, Internet Engineering Task Force, December 1997.

[179] C. Weider, J. Reynolds, and S. Heker. Technical overview of directory services using the X.500 protocol. Request for Comment (RFC) 1309, Internet Engineering Task Force, March 1992.

[180] C. Weissman. Security controls in the ADEPT-50 time-sharing system. In *AFIPS Conference Proceedings*, volume 35, pages 119–133. FJCC, 1969.

[181] B. S. White, M. Walker, M. Humphrey, and A. S. Grimshaw. LegionFS: A secure and scalable file system supporting cross-domain high-performance applications. In *SC2001: High Performance Networking and Computing*, Denver, Colorado, November 10–16 2001.

[182] R. Wright, A. Getchell, T. Howes, S. Sataluri, P. Yee, and W. Yeong. Recommendations for an X.500 production directory service. Request for Comments (RFC) 1803, Internet Engineering Task Force, June 1995.

[183] L. Xiong and L. Liu. Peertrust: Supporting reputation-based trust in peer-to-peer communities. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2004. Special Issue on Peer-to-Peer Based Data Management.

[184] W. Yeong, T. Howes, and S. Kille. Lightweight directory access protocol. Request for Comment (RFC) 1777, Internet Engineering Task Force, March 1995.

[185] X. Zhang, S. Oh, and R. Sandhu. PDBM: A flexible delegation model in RBAC. In *Proceedings of the 7th ACM Symposium on Access Control Models and Technologies (SACMAT 2003)*, Como, Italy, June 2003.

[186] H. Zhou and S. N. Foley. A framework for establishing decentralized secure coalitions. In *Proceedings of IEEE Computer Security Foundations Workshop*, Venice, Italy, July 2006.

[187] P. Zimmermann. *The Official PGP Users Guide*. MIT Press, 1995.