Opsis: A Distributed Object Architecture Based on Bracket Capabilities

Mark Evered

School of Mathematical and Computer Sciences
University of New England, Armidale, 2351, Australia

Email: markev@mcs.une.edu.au

Abstract

The object-oriented access control in contemporary middleware is inadequate in view of the sensitivity of data stored on the internet and the growing threat from hackers and malicious software. In this paper we present the Opsis system, an architecture for distributed Java applications based on the strict use of 'bracket capabilities'. We describe the concept of 'capability servers' for supporting flexibility and transparency of remote invocation and for allowing the migration of objects. We demonstrate the power and simplicity of the system in an example E-commerce application including the definition of a form of secure electronic cheque[1].

Keywords: security, internet, middleware, E-commerce

1 Introduction

Despite a growing threat, insufficient attention has been paid to security constraints in middleware development. OMG's Corba (Blakley et al., 2000) and Microsoft's COM (Eddon, 1999) both include a form of access control list (ACL) but these are add-on features which remain limited and inflexible. Java's security package (Sun, 2000) focuses on the rights to be granted to foreign code rather than access control between the components of a distributed system. Much attention has been given to encryption techniques but, while encryption is very important, it protects only the communication and authentication in the system. It provides only the basis for a secure access control mechanism.

The components of a distributed system can be viewed as objects, with each object containing persistent data hidden by encapsulation and accessible via interface methods. (This may, of course, simply be a façade around a legacy software component such as a relational database). One advantage of an object-oriented approach is that the security can be based on the interface methods of an object. This may be called object-oriented access control. It provides a fine-grained *semantic protection* (Evered, 2000) in contrast to the course-grained read/write protection of traditional files and databases. This means that the access rights can be based on meaningful, high-level operations associated with the object in the real world (Fig. 1).

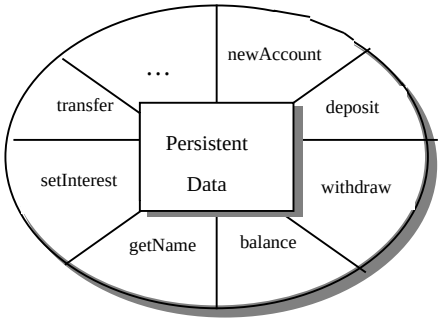

Fig 1: A persistent object of type 'Accounts' for a set of bank accounts

This idea goes back as far as Jones' and Liskov's suggestion of a static type-based constraint mechanism (Jones and Liskov, 1978) and it has been adopted in contemporary middleware mechanisms. The COM security mechanism, for example, does this by offering 'per-method access control lists' which record, for each method, a list of users allowed to invoke that method (Eddon, 1999). These kinds of mechanism limit the access to an object by returning an error message if certain methods are invoked. However, this is not the only kind of access restriction which is possible or useful in controlling access to a persistent object. In fact, even in terms of per-method access control, the standard mechanisms are not ideal since all the methods of the object are still known to all the users even if they cannot be called. Ideally, in a need-to-know security environment, someone who is not allowed to invoke an object should not know of the existence of that object and someone who is not allowed to invoke a particular method should not know of the existence of that method.

Some kinds of access control are not supported at all by standard middleware. For example, what access to an `Accounts` object should be given to the owner of an individual account? As well as restricting access to the methods `balance`, `getName` and `transfer`, we must also ensure that only the right account is being accessed. This means the parameters giving the account number of the account to be accessed must be restricted to a particular value.

In (Evered, 2001) the author has identified a number of different kinds of access control which may be necessary

for specifying the security constraints associated with different users of a system in accessing persistent objects. These include:

- specifying that some methods should return an access violation error
- specifying that an access violation error should be returned for parameter values other than a specified value
- restricting the type of the object as seen by a client to exactly the allowed methods and parameters
- enhancing the semantics of the object type with logging of accesses and access attempts
- specifying state-dependent rules such as ‘access only at specified times’ or ‘access allowed only once’

In (Evered, 2002) we introduced the concept of a 'bracket capability' as a simple security mechanism capable of supporting these kinds of security constraint. In this paper we present the Opsis system as a prototype middleware architecture for internet-based object systems, based on the strict use of bracket capabilities as object identifiers.

In the following section we briefly review the idea of bracket capabilities. Section 3 describes the basic aims and structure of the Opsis architecture. In sections 4 to 6 more detail is given on the role of capability servers, object creation and bracketing respectively. In section 7 we show how the system can be used in an example E-commerce environment, with capabilities representing a form of secure electronic cheque. Section 8 reports on related work and section 9 concludes the paper.

2 Bracket Capabilities

Our mechanism is based on object capabilities rather than ACLs because capabilities enhance and simplify security by unifying object naming with the protection mechanism (Wilkes and Needham, 1979). A number of possible alternatives have been suggested for implementing capabilities. These include special architectures (Rosenberg and Abramson, 1985), encryption (Mullender and Tanenbaum, 1986) and sparse (or password) capabilities (Anderson et al., 1986). We base our mechanism on sparse capabilities since these require no special architecture or costly encryption algorithms and also because they alleviate the revocation problem. A sparse capability generally consists of an object identifier (for locating the object) together with a large (un-guessable) random number (the password). The access rights associated with the capability (that is, with that particular random number) are not stored in the capability itself but with the object being accessed, so can easily be modified or revoked without access to the capability.

Our capabilities differ from traditional sparse capabilities in that they contain not an object identifier, but an identifier for a (capability) server which knows the location of the object. This indirection allows for object migration as well as flexibility in the communication mechanism. These are both particularly important for mobile applications. When a persistent object is created, a capability for the object is created and registered with a capability server.

The main distinguishing characteristic of bracket capabilities is seen in the process of *refinement*, that is, when the possessor of a capability wishes to grant a more restricted view of the object to other users in the system. This is done by a call to the `refine` method. Each persistent object, as well as implementing an interface such as `Accounts` also implements a number of administrative methods such as `deleteObject`, `deleteCapability` and `refine`. The `refine` method in a bracket capability system has the form:

```
refCap = x.refine(interface, class);
```

where `interface` denotes the type with which the persistent object ‘`x`’ is to be viewed when used via the capability `refCap` and `class` denotes the class of an object through which calls to the persistent object will pass when invoked via `refCap`. The result of the `refine` call is depicted in Fig. 2.

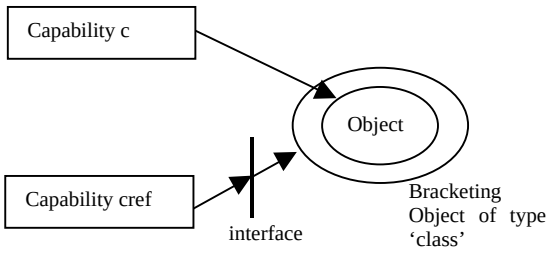

Fig 2: The result of the 'refine' operation

It can be seen that calls using the capability `refCap` are directed through a kind of proxy or *bracketing* object of class `class`. This bracketing object is stored together with the persistent object in the same way that access rights are stored with the object for traditional sparse capabilities.

A copy of `refCap` can be given to the users who are to have this kind of access. Traditional object-based capabilities or ACLs in which access is restricted to particular methods can easily be simulated with this mechanism while other bracket classes can be used to implement the full range of security constraints described in the introduction, including restrictions on parameters, logging of method calls and constraints based on time, number of accesses etc. A bracketing class may have more methods than those available in the view given by the interface type. These extra methods can be used by the creator of the capability for monitoring or altering the bracketing such as to inspect logging information or to revoke or alter access constraints.

Although the bracket capability mechanism allows

arbitrary bracketing classes, in a particular security environment, we may want to limit the set of classes that can be used as brackets. While still allowing users to create more restricted views, we may want to specify what kind of restriction they can impose. This is possible since the bracketing is part of the security mechanism. In fact, since the `refine` call is itself just a method call to the persistent object, we can use the mechanism itself to specify that, for some user, it can only be invoked with certain values for the `class` parameter.

3 Opsis – Aims and Architecture

Opsis is a prototype system for constructing distributed Java applications based on the mechanism of bracket capabilities. The system consists of middleware for connecting persistent objects distributed across the internet and a set of utilities for constructing the components of an application and specifying the security constraints.

The two main aims of the system are:

- fine-grained access control through the use of bracket capabilities as object identifiers
- flexibility and transparency in the mechanisms used for both remote invocation and persistence

In order to gain access to an object, the object is 'opened' using a capability. For example, assuming the interface:

```
interface Accounts {
  void newAccount(int newKey, String name);
  void deposit(int key, int amount);
  void withdraw(int key, int amount)
         throws insufficientFunds;
  int balance(int key);
  String getName(int key);
  void setInterest(float rate);
  void transfer(int fromKey, int toKey,
                int amount)
         throws insufficientFunds;
}
```

for the `Accounts` object described above, the object can be accessed with:

```
Accounts acc = (Accounts) cap.open();
int b=acc.balance(12345); // get the balance
                  // of account number 12345
```

where `cap` is a variable of type `Capability`.

Each capability is a 128-bit value and consists of a 36-bit capability-server identifier (CSID) and a 92-bit password. The CSID identifies a server which knows the location of the persistent object. The first 4 bits of the CSID specify the protocol to be used to contact the server. Currently, only one protocol is supported, with the remaining 32-bits of the CSID specifying the IP number of the server. The `open` operation on a capability leads to a look-up operation on the server.

The Opsis architecture supports multiple mechanisms for remote invocation and (orthogonally) multiple mechanisms for persistence. In all cases, the mechanisms used for the invocation and the persistence remain completely transparent to a client. The path of a method call from a client to a server object can be visualised as in Fig 3.

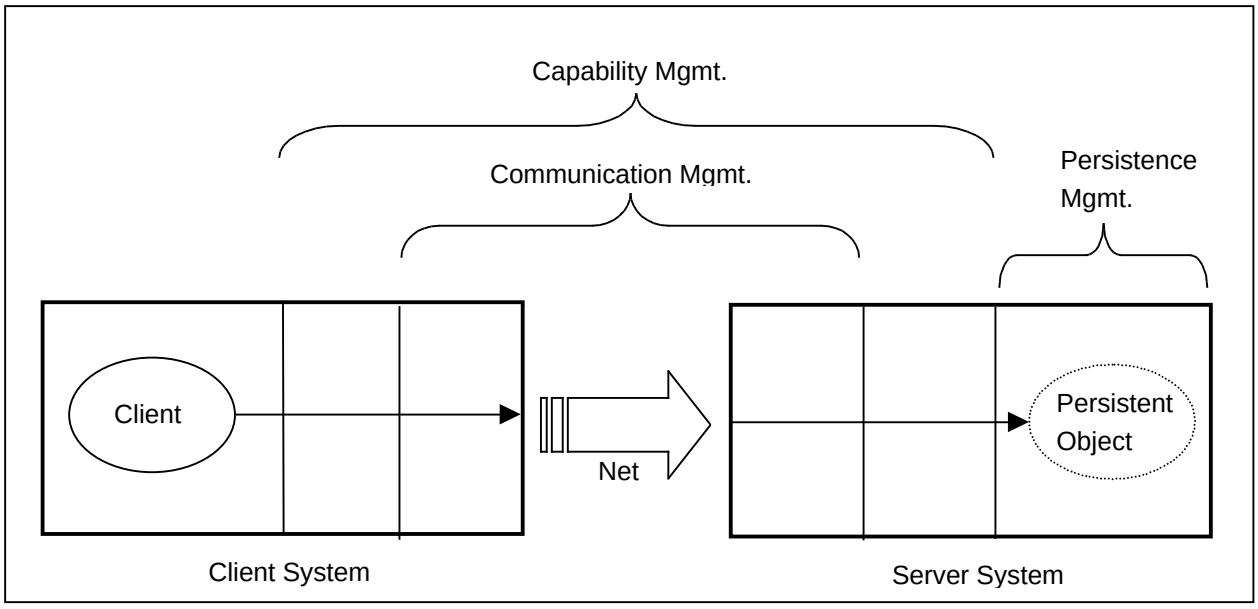

Fig. 3: A method call in Opsis

The communication management uses the appropriate mechanism for passing the method call and parameters to the remote system and returning a result. It supports multiple concurrent invocations.

The capability management on the client side is responsible for providing the right view of the persistent object to the client and for appending the capability used to open the object to each method invocation as an extra parameter. The capability management on the server side is responsible for checking that the access proceeds only as allowed for that capability. Nothing that can be done on the client system can increase the access rights to the remote object. The persistence management is responsible for making the persistent data available and shared, and presenting it in the form of an object of the appropriate type.

The mechanisms for remote invocation currently implemented are:

- Java's RMI
- a web-based CGI mechanism
- a mechanism using 'ssh', the secure shell protocol

Currently implemented mechanisms for persistence are:

- Java's built-in light-weight persistence (using inter-process communication to achieve sharing)
- wrapper objects around a ‘Postgres’ database

The (initial) mechanisms to be used for remote invocation and persistence are specified when an object is created. This is discussed further in section 5. The mechanism for remote invocation is provided to the middleware by the capability server when an object is opened. This is discussed in more detail in the next section.

4 Capability Servers

In order for a client to access a remote persistent object, certain information must be made available to it. This includes the type of the object (ie. its interface), its location, its name and the mechanism to be used in remote invocations of the object's methods. All this information is kept in capability servers rather than in the capabilities themselves. The capability contains the location of a server which is guaranteed to hold an up-to-date copy of the information. For robustness and efficiency, the information may be held in other servers as well but these are not guaranteed to be up-to-date. When an object is created, it is registered with a server and the location of that server is stored in the top 36-bits of the capability.

As well as simplifying the capabilities, the advantage of this indirection is that the information can be changed without affecting the capabilities themselves and therefore transparently for the clients. This allows an object to be moved easily to a new server and/or to be accessed via a different invocation mechanism. In a mobile application, for example, an object can be locked, installed at a new location re-registered with the server and then unlocked and calls will now be directed to the new location.

Capability servers are given only the lowest 32-bits of the capability’s password as an index rather than the whole 92-bits. This is because otherwise, the server would essentially own a copy of the capability and therefore have all the associated access rights.

A capability server is itself stored as a persistent object and access to this object can be controlled just as for any object in the system. The interface is defined as:

```
interface CapServer {
  void newObj(int cap,
              String typeName,
              String objName,
              String commMech,
              String location,
              String comment);
  void newCap(int cap,
              int oldcap,
              String view,
              String param,
              String comment);
  void movedObj(int cap,
                String objName,
                String commMech,
                String location);
  String lookup(int cap) throws CapNotFound;
  String getComment(int cap);
  void removeCap(int cap);
}
```

The `newObj` method is used to register a new object with the server while the `newCap` method registers a new capability for a known object. The `view` and `param` parameters are explained in section 6. The `comment` parameters are used to enter a description of the purpose of the capability. The method `movedObj` is used for re-registering an object which has been moved. The `lookup` method is used by the client-side middleware and returns a single string containing all the information required for an invocation.

5 Object Creation

Once an object has been created, a capability can be used to invoke the methods of the object. In any capability-based system, however, an important question is how the object can be created in the first place. In the system described in (Evered, 2000), the author defined a special purpose mechanism for object creation. This was unsatisfactory for two reasons: firstly, the code creating a persistent object took a different form from other operations and, secondly, there was no way of restricting the permission to create new objects.

This problem has been solved in Opsis by the introduction of 'creator objects'. A creator object is automatically created and registered with a capability server when Opsis is installed at a location. A capability for the creator object can then be given to a user to allow

the creation of objects at that location. Bracket capability access control can be used to restrict how many and what types of objects a particular user is allowed to create. The interface of a creator object is defined as:

```
interface Creator {
  void addType(String typeName,
               String defImpl,
               String defPersMech,
               String defCommMech);
  void deleteType(String typeName);
  Capability create(String typeName,
                    String comment);
  Capability create(String typeName,
                    String impl,
                    String persMech,
                    String commMech,
                    String comment);
}
```

An object can either be created with the default implementation, persistence mechanism and invocation mechanism for that object type at that location or with explicitly specified mechanisms. The names for the mechanisms are the names of Java classes used by the creator object for setting up the new object. The creator object registers each new object with a capability server that is specified at the time the creator object is installed.

The creation of an `Accounts` object can be achieved by:

```
(Creator) c = (Creator) creatorCap.open();
Capability objCap = c.create("Accounts",
                             "Branch xyz");
```

6 Administration and Refinement

As well as implementing a 'functional' interface such as `Accounts`, every persistent object in Opsis also implements the interface `Secure` which defines a set of administrative operations. This is defined as:

```
interface Secure {
  void close();
  int  lock();
  void unlock();
  void delete();
  long refineCap(String interf,
                 String param,
                 String comment);
  void deleteCap();
}
```

A call to the `close` method is not required but can be used to indicate that an application has finished using the object. The `lock` method prevents further calls to the object and returns the number of processes still active in the object. The `delete` method deletes the entire object while the `deleteCap` method removes all rights for the capability with which the object was opened (and also for all capabilities derived from that capability).

As described in (Evered, 2002), the parameters to the `refine` call are an interface and a class. In Opsis, this is realised via the `String` parameter `interf`. This assumes the existence of a Java interface type with the name given in that parameter and a Java class with that name followed by “Bracket”. So, for example, given the interface type:

```
interface Account {
  int balance();
  String getName();
  void transfer(int toKey, int amount)
         throws insufficientFunds;
}
```

and an appropriate class `AccountBracket` the following code creates a new capability which presents the `Accounts` object to the user as if it were his/her individual `Account` object.

```
(Secure) acc = (Secure) objCap.open();
Capability accountCap = acc.refine("Account",
     "12345", "Account 12345 at Branch xyz");
```

The server-side capability management maps the capability to an instance of the class `AccountBracket` and passes the call through this to supply the appropriate account number for the call to the underlying `Accounts` object.

Some bracketing classes may be hand-written but Opsis contains utilities which allow the automatic generation of the most common kinds of brackets. So, for example, given the interface types `Accounts` and `Account`, the class `AccountBracket` can be generated by using the utility `bracket` as:

```
bracket Account Accounts key=PARAM
                         fromKey=PARAM
```

This means that the `key` and `fromKey` parameters which are required for `Accounts` but missing in `Account` will be given the value passed to the bracketing object as a parameter in the `refine` operation (“12345” in the above example).

7 Example: Secure Electronic Cheques for E-Commerce

We now give an example of a simple system using Opsis for security in electronic funds management. At the centre of the system is an object of the type `Accounts` as described above. After creating this object, we have a capability `objCap` for unrestricted access.

The first level of security is the logging of all accesses and access attempts to the object. We can achieve this by creating a new capability as:

```
acc = (Secure) objCap.open();
logCap = acc.refine("LoggedAccounts",
                    "acclog",
                    "Branch xyz (logged)");
```

where `LoggedAccounts` is a class for a bracketing object which records the parameters, time and state of the relevant account in a file (in this case a file called `acclog`) and passes the call on to the `Accounts` object. Only copies of the `logCap` capability and not the original `objCap` capability will be further distributed in the system.

Next we can create a capability for an individual bank account holder. For account number 12345, this can be achieved as described in the previous section with:

```
acc = (Secure) logCap.open();
accountCap = acc.refine("Account", "12345",
              "Account 12345 at Branch xyz");
```

Finally, the account owner may wish to provide a restricted access to his/her account so that another account owner can transfer a certain amount, say $100, out of the account as a payment. The capability for such an access is in fact a *secure electronic cheque*. As well as fixing the amount, we must ensure that this capability can only be used once. We can achieve this as:

```
acc = (Secure) accountCap.open();
chequeCap = acc.refine("Cheque", 100,
               "Payment for your services");
```

where `Cheque` is defined as:

```
interface Cheque {
  void transfer(int toKey)
         throws insufficientFunds;
}
```

and `ChequeBracket` is a bracketing object which ensures that the right amount is being transferred and, additionally, calls the `deleteCap` method to ensure that the cheque cannot be used a second time. The electronic cheque can be deposited in an account, say account 23456, by the code:

```
Cheque c = (Cheque) chequeCap.open();
c.transfer(23456);
```

Clearly, the destination account could also be fixed if desired, or, by using a bracket such as `MonthlyDebitBracket` instead of `ChequeBracket`, the same mechanism could be used to create a capability for regular transfers rather than a once-off payment.

8 Related Work

As mentioned above, Corba (Mowbray and Zahavi, 1995) and COM+ (Eddon, 1999) both include the possibility of a per-method, role-based access control list for limiting the access of users to objects. In some cases, fixed forms of rule-based access, such as access at certain times of day, are supported. These correspond only to simple, special cases of access control. No direct equivalent of the logging and parameter restrictions as required for the above E-commerce example are supported. No direct equivalent of a restricted view type is supported for hiding the existence of unallowed methods and parameters from the users. In both of these middleware technologies, the use of ACLs instead of capabilities makes the security mechanism an add-on feature rather than fundamental and detracts from the security.

Object capabilities have been used in a number of research systems, most notably the Monads system (Rosenberg and Abramson, 1985) but these capabilities require architectural support (or at least a special operating system kernel) and so are not appropriate for heterogeneous networks. In a previous project, the author has developed a capability-based mechanism for heterogeneous distributed applications (Evered, 2000). Like the Monads system and the ACL approaches of Corba and COM, however, this supported only simple per-method access control.

The concept of ‘bracketing’ for applying access constraints has been suggested both as a programming language construct (Keedy et al., 2000) and as a form of ‘design pattern’ (Gamma et al., 1995). The suggested programming language approach is interesting in supporting the *reuse* of the bracketing code but it does not allow modification of the interface to the underlying object and, being integrated into the type system of the language, it is a static mechanism.

One use of the *proxy* design pattern is as a protection (or access) proxy. In this case, the interface is identical to the underlying object. The proxy decides whether the access can proceed and returns an error if it should not. Simple per-method access control can be realised by this kind of protection proxy. A proxy object which maintains a log of access attempts could be seen as a kind of *decorator* pattern (though this is most often seen as a graphical decoration) since it maintains the original functionality while enhancing it with a logging and reporting functionality. Bracketing objects which modify the interface offered to a client cannot be seen as strict proxies. They can be seen as special cases of the adapter pattern but whereas an adapter is usually used to provide the view the client would *like* to have of the underlying object, in these cases the adapter is providing the view the client is *allowed* to have.

The concept of providing a user with a restricted view of persistent data is reminiscent of database systems. Database views are attribute-oriented and not method-oriented, however, and do not support the flexible kinds of access control demonstrated in our example. This is true even for object-oriented databases (Mishra and Eich, 1994). Brose (1999) describes a ‘view-based’ mechanism

for Corba but this is again simply a kind of language-based per-method access control. It does not hide the unallowed methods and does not support views involving parameter restrictions.

9 Conclusion

The object-oriented access control in contemporary middleware is inadequate in view of the sensitivity of data stored on the internet and the growing threat from hackers and malicious software. These security mechanisms are not integrated at a fundamental level and are capable of enforcing only simple kinds of access control. The access control in a distributed object system should ideally enforce a strict need-to-know view of the persistent objects and should support more complex forms of security restriction including logging of accesses and restrictions on parameter values.

In this paper we have described the Opsis system, an architecture for Java applications composed of objects distributed across the internet. The main feature of the system is the strict use of bracket capabilities as object identifiers. Bracket capabilities are a new protection mechanism for distributed object systems. They associate a capability for an object with a type which defines a client's view of that object and with a nesting of bracketing objects through which the underlying object is to be accessed. The other important aspect of Opsis is the flexibility and transparency of the mechanisms used for remote invocation and for persistence.

The definition of a standard interface `Secure` for persistent objects in Opsis allows the use of bracket capabilities not only for the application-related methods of an object but also for administrative operations. This includes the management of which bracketing objects can be used in a particular security environment and the control of the creation as well as the use of persistent objects.

We have described the concept of capability servers. These are used to hold the information a client needs in order to access an object using a certain capability, including its location, the view type for that capability and the required form of remote invocation. Capability servers allow the capabilities themselves to be simple 128-bit values and support transparent changes such as migration of the objects as required by mobile applications.

Finally, we have demonstrated the power and simplicity of the system in controlling the access within an example E-commerce application including the definition of a form of secure electronic cheque.

References

ANDERSON, M., POSE, R.D., WALLACE, C.S. (1986): A Password-Capability System, *The Computer Journal*, 29,1, pp.1-8.

ATKINSON, M.P., JORDAN, M.J., DAYNES, L. SPENCE, S. (1996): Design Issues for Persistent Java: a Type-Safe Object-Oriented, Orthogonally Persistent System, *Proc. 7th Intl. Workshop Persistent Object Systems*, Cape May.

BLAKLEY, B., BLAKLEY, R., SOLEY, R.M. (2000): *CORBA Security: An Introduction to Safe Computing with Objects*, Addison-Wesley.

BROSE, G. (1999): A View-Based Access Control Model for CORBA, in: Jan Vitek, Christian Jensen (eds.), *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, LNCS 1603, Springer.

EDDON, G. (1999): The COM+ Security Model Gets You Out of the Security Programming Business, *Microsoft Systems Journal*, Nov.

EVERED, M. (2000): A Two-Level Architecture for Semantic Protection of Persistent Distributed Objects, *Proc. Intl. Conf. on Software Methods and Tools*, Wollongong.

EVERED, M. (2001): Type Operators for Role-based Object Security, WiP, *3rd IFIP/ACM Intl. Conf. on Distributed Systems Platforms - Middleware*, Heidelberg.

EVERED, M. (2002): Bracket Capabilities for Distributed Systems Security, *Proc. 25th Australasian Computer Science Conference*, Melbourne.

GAMMA, E. ET AL. (1995): *Design Patterns*, Addison-Wesley.

GOSLING, J., JOY, B. AND STEELE, G. (1996): *The Java Language Specification*, Reading, MA: Addison-Wesley.

HARRISON, M.A., RUZZO, W.L., ULLMAN, J.D. (1976): Protection in Operating Systems, *Communications of the ACM*, 19, 8.

JOSHI, J.B.D. ET AL. (2001): Security Models for Web-based Applications, *Communications of the ACM*, 44, 2.

JONES, A. AND LISKOV, B. (1978): A language extension for expressing constraints on data access. *Communications of the ACM*, 21(5):358-367, May.

KEEDY, J.L. AND VOSSEBERG, K. (1992): Persistent Protected Modules and Persistent Processes as a Base for a More Secure Operating System, *Proc. 25th Hawaii International Conference on System Sciences*, IEEE Computer Society Press, S. 747-756.

KEEDY, J.L., ET AL. (2000): Software Reuse in an Object Oriented Framework: Distinguishing Types from Implementations and Objects from Attributes, *Proc. Sixth International Conference on Software Reuse*, Vienna.

MISHRA, P. AND EICH, M.H. (1994): Taxonomy of views in OODBs, *Proc. ACM Computer Science Conference*.

MORRISON, R., BROWN, A.L., CARRICK, C. ET AL. (1989) The Napier Type System, *Proc. 3rd Intl. Workshop on Persistent Object Systems*, Newcastle.

MULLENDER, S.J., TANENBAUM, A.S. (1986) The Design of a Capability-Based Distributed Operating System, *Computer Journal*, 29,4, pp.289-299.

MOWBRAY, T.J. & ZAHAVI, R. (1995): *The Essential Corba - Systems Integration Using Distributed Objects*, Wiley, New York.

ROSENBERG, J., ABRAMSON, D.A. (1985): The MONADS Architecture: Motivation and Implementation, *Proc. First Pan Pacific Computer Conference*, p. 4/10-4/23.

SUN MICROSYSTEMS INC. (2000):. Java Security Architecture, *http://java.sun.com/products/jdk/1.2/docs /guide/security/spec/security-spec.doc.html*

WILKES, M.V., NEEDHAM, R.M. (1979): *The Cambridge CAP Computer and its Operating System*, North Holland.