Internal report, Computer Science, RAU, 1996

Self-protecting Objects in Multipolicy Federated Databases: A Prototype

Martin S Olivier
Rand Afrikaans University

Abstract

A multipolicy secure federated database is a distributed database that allows the various sites to implement different security policies. The policy of a site is then enforced by all sites for data owned by this site.

This paper describes a proof of concept prototype of such a multipolicy secure federated database. The prototype implements a multilevel federal security policy—that is a policy that applies to all members of the federation. The individual sites can then extend this policy for data owned by them with discretionary and/or multilevel site security policies.

The design of the prototype focusses on the interaction between the modules and services that implement it. The paper argues from these interactions that it is possible to implement the prototype such that it can be trusted.

Keyword codes: C.2.4, H.2.5, K.6.5
Keywords: Distributed Systems, Heterogeneous Databases, Security and Protection

1 Introduction

A federated database is a distributed database that allows a high degree of site autonomy. Often such a database is owned by more than one organisation, each with its own view about security. Such different views on security complicates interaction. One possible solution to allow cooperation is to 'encapsulate' the code that protects an object with the object itself. This means that, even though an object may be moved from one site to another site (for query optimisation or other reasons) the object will be protected by the code of the site that owns the object. In [7] a model has been proposed that may serve as an implementation approach for such self-protecting objects. The current paper describes a proof of concept prototype based on that model.

For ease of reference the implementation model in [7] will be referred to by the acronym SPO *(self-protecting object model)*; the prototype described in this paper will be called SPOP *(SPO prototype)*.

The prototype described in this paper shows that it is possible to build a federated database where each site maintains its own security policy. Although the prototype makes assumptions about the federal security policy (that applies to all sites) the design is not restricted to that security policy. No assumptions are made about the security policies of the individual sites. The design in this paper can therefore serve as a general design for any federation based on the SPO architecture. An extensive argument that the design is secure is included.

Section 2 of this paper gives the necessary background; in particular the operation of SPO is described. Section 3 describes how objects will be represented in SPOP. Next, section 4 describes the federal security policy used by SPOP. The various modules into which the SPOP services are divided are described in section 5. Section 6 argues that the design is secure. Section 7 contains the conclusion.

2 Background

SPO uses a 'core' of security functionality common to all sites of the federation. This core is trusted and shared by all members of the federation. However, every member of the federation is allowed to extend its security beyond the shared core for its objects. Each site's 'extended' protection then has to be enforced, even if the site's data is used on other sites of the federation. As an example, if an object is relocated to another site (say, for query optimisation purposes) then it will 'take its "extended" protection along' to the new site. The common core will be responsible to ensure that the other site honours the protection imposed by the owning site of the relocated object.

Each site in an SPO database consists of three components:

1. The trusted common core (TCC) provides security functionality available at every site of the federation. The TCC is equivalent to the TCC on all other sites of the federation (and is trusted by all members of the federation).

2. Additional security code may be added by the site to protect objects owned by the site. Such a trusted extension (TE) does not need to be trusted by any other member of the federation. A TE will be defined at the site that uses it; however, when an object is relocated from one site in the federation to another, the TE code that helps to protect it, may also be relocated to the new site. Therefore a site may contain TE code of its own as well as TE code from other sites.

3. Each site may additionally contain security code to support the TE methods (by providing them with information they require) and to provide general security functions. This component, known as the TLE *(Trusted Local Extension)* may provide support for the TEs of the concerned site; it may also provide functionality not provided by the TCC. The TLE also does not need to be trusted by other members of the federation (but see the discussion later).

The three components and their relationships are depicted in figure 1. LDB refers to the local database at a site.

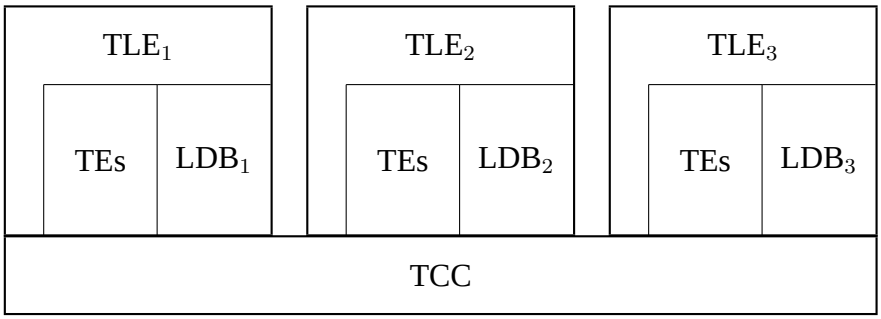

Figure 1: The components of SPO

The prototype described in this paper focusses on the TCC. The TCC on each site of the prototype is identical (and not merely equivalent). The purpose of the prototype is to demonstrate that it is indeed possible to build a system that uses SPO. The prototype is further intended as a vehicle to (eventually) experiment with the possible local security extensions that individual sites may add.

The purpose of this paper is to present the design of the prototype and argue that the design is indeed secure.

Put in a nutshell, the security assumptions made by SPOP are the following: The TCC provides multilevel security. Individual sites may add additional checks using the multilevel secure core; individual sites may also add discretionary checks. It is assumed that each individual site performs authentication and authorisation functions using its TLE. These assumptions will be discussed in more detail in a following section.

SPOP is focussed on the security aspects of a database and, for reasons of simplicity, does not address traditional database problems (such as efficient management of large volumes of data and concurrency control) at all. The additional implications of such problems will only be studied in the SPOP context after the direct consequences of the SPO approach have been studied.

Current research regarding independent security decisions by different sites tends to either determine and remove possible clashes during integration (see for example [3]) or retain as much of the existing database systems, even at the cost of functionality of the federated database (see for example [4]). The SPO (and therefore SPOP) approach is intended for cases where it is indeed possible to decide on the database system for the sites of the federation, but where it is important to have the ability to decide locally, at each site, on the security policy. SPO is therefore more applicable when designing a new federated database, rather than composing one from existing stand-alone databases.

Most federated databases add a federal layer to integrate the various nodes of the database securely [4, 12]. This layer is (presumably) managed centrally. SPO maintains decentralised management and individual site security policies. Another prototype that allows multipolicy interoperation is given by [1]. That prototype differs from SPO because it allows interoperation of transactions for which agreements have been entered into the system, it is based on the relational model and makes explicit provision for legacy databases. In contrast SPO allows interoperation for any transactions for which authorised users are known (or recognised) at the owning sites, it is based on the object-oriented model and is primarily designed for non-legacy databases.

See Özsu and Valduriez [11] for a discussion of distributed databases and [11, pp81,89] and [14] for a discussion of federated databases in particular. See [15] for a discussion of the issues that must be considered when designing a secure federated database.

3 Data representation in SPOP

The research in this paper assumes that the underlying databases are object-oriented. See [5] for a description of object-oriented databases and [6, 10, 13] for a discussion of security in such databases.

Objects in an object-oriented database are described using some high-level object-oriented data language. This data language can be converted to an internal representation using a compiler. The data language used by humans to describe objects will be called the *external data language*, while the internal representation of objects will be called the *internal data language*.

The external data language may support all the usual object-oriented features, such as classes and inheritance. However, SPOP assumes that the internal representation of an object will only consist of a collection of instance variables and a collection of methods. The fact that many objects have the same methods (defined in their common classes and superclasses) is ignored by SPOP. The problems and potential benefits of code sharing made possible by inheritance, will be investigated in a later study.

In some languages classes are considered as objects themselves (see for example Smalltalk [2]). If the external data language falls in this category, the classes are ‘factory’ objects, capable of producing other objects. Such ‘factory’ objects will eventually be represented in the internal data language just like any other object. Some other languages view classes merely as templates for creating objects. In such languages the class need not be represented in the internal data language at all.

The external data language, as well as the compiler that translates it to the internal data language, fall outside the scope of this paper. The only assumption made is that the external data language can indeed be translated to the internal data language.

Although not required by SPO, SPOP uses the same internal data language to represent objects on all sites, to transfer objects between sites and to add new objects to the system.

3.1 Representation of objects

The internal data language representation of an object uses three tables: The primary table is one containing information about all methods and variables, with pointers to the representation of the methods and variables. The methods and variables will be held in separate tables, pointed to from the object table. The table representing an object is given in figure 2.

ResNo	Name	Details		Security			
				Sens	Extension		
		Pointer	Length		OSID	OID	MID

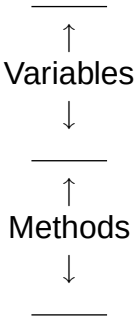

Figure 2: Object table

In this figure **ResNo** is the *resource number*: all instance variables and all methods are numbered uniquely. Whenever a method wants to read or write a variable, it will refer to this variable by its resource number. Similarly, whenever a method wants to send a message, it can do so by using the resource number of the method that will handle the message. However, resource numbers are only unique per object—to distinguish between similarly numbered resources of different objects, objects are given unique identifiers by its owning site; these identifiers are called object identifiers (OIDs)—that is, no two objects owned by the same site will have the same OID. Further, sites are identified uniquely within the federation. Therefore the owner site identifier (OSID), OID and method or variable identifier uniquely identifies the method or variable.

Next, the object table contains the internal **Details** of the method or variable: a pointer to the memory where the representation is kept as well as the length (in words) of the facet's representation.

Lastly, the **Security** information of the facet is recorded in the object table. Since SPOP uses multilevel security, the sensitivity level of the facet is recorded in the **Sens** column, while the OSID, OID and MID (method identifier) of the TE method (that additionally protects the facet) is recorded in the **Extension** columns of the object table.

As stated earlier, the object table is always accompanied by other data structures: Firstly, a *code table* contains the executable code of the methods identified in the object

table, while a *value table* contains the values of the variables identified by the object table.

3.2 Representation of methods

As described in the previous section, a method is identified in the object table and pointed to by a pointer. The actual code of the method is kept in the code table.

The code of a method is represented in postfix notation: A stack is used for intermediate results. In order to send a message to an object, all the parameters for the message are pushed onto the stack. Then an identification of the object is pushed, followed by an identification of the message. After this a 'send message' instruction is executed. The result of the message (if any) is left on top of the stack after it has been handled by the designated object. In addition to the general 'send message' operation, provision is made for a number of primitive operations: as an example the 'add' operation will pop two numbers from the stack, add them and push the result back onto the stack.

The postfix notation has been selected since it avoids the complexities inherent in most computer architectures, while it is still fast enough for most database applications.

Lack of space prevents a detailed discussion of the precise notation used. Operations supported include the basic arithmetic operations, the generic 'send message', reading and writing of instance variables, as well as directory lookup operations. It is worth pointing out that the approach does allow late binding.

3.3 Representation of variables

Variables are kept in the value table. Primitive values, such as integers and real numbers are represented directly. References to other objects are kept as $\langle$ OSID, OID $\rangle$ pairs.

4 The security policy for SPOP

SPOP provides the federal security functionality that allows all the sites to implement their own security restrictions.

The SPOP TCC uses multilevel security. Users from any site will be assigned a clearance level by that site. Similarly, any site has to indicate the sensitivity of any object owned by that site. Whenever a message is sent by a subject at a site, the message is labelled with clearance and sensitivity labels. The clearance label is determined from the clearance of the subject that sent the message. The sensitivity label indicates the sensitivity level of the most sensitive information that the message has had access to—this label usually starts at the system-low security label and is then increased whenever a more sensitive variable is read or written.

However, sites may wish to implement additional security functionality. In particular, to implement discretionary security, it is necessary to know the identification of the subject that originated the message. Further, some sites may record information about the circumstances under which the message was originated, such as the time when, and terminal identification where the message originated, and then use this information for access control purposes. SPOP therefore includes the following authorisation information with every message sent: The clearance of the subject that sent the message, the identification of the subject that sent the message and a unique message identification number that will allow later determination of information such as time sent. This authorisation information is determined at the point where the message enters the system through the user interface (that is, from a human user or application program); messages that are sent by methods activated as a result of a message will be given the authorisation information of that message message at the concerned point.

For its multilevel security, SPOP assumes that all facets (instance variables and methods) are labelled. However, it does not preclude an individual site from labelling objects, as well as facets, or only labelling objects: In the first case, an object label merely serves as a minimum label for all facets of the object. In the case of single level objects (where only objects are labelled) the object label will be used for all facets of the object.

It is clear from the above that a SPOP security policy will consist of a federal security policy as well as a number of site security policies. The federal security policy will be described in the following section, while the possibilities for site security policies will be addressed in the next section.

The federal security policy

Here we list the design choices made for the federal policy of SPOP using the parameters identified in [10].

Labelling semantics

X1.1 Underlying model: SPOP uses explicit numeric labels to classify information (sensitivity) and users (clearance). A user (subject) may read a variable or execute a method if its clearance dominates the sensitivity of the variable or method. In addition to the use of security levels, sites may add any additional access checks they wish for objects owned by them. These checks may also use the assigned security levels, but are not restricted to use only them. (See later for examples).

X1.2 Protection interpretation: The federation does not prescribe the use of access protection, existence protection or any other possibilities. (Existence protection is used when the fact that an entity exists is as sensitive as the contents (or value) of that entity. Access protection is used when use of an entity is more restricted than knowing that the entity exists. See [10] for complete definitions.) However,

SPOP will ensure that the choice made by each site will be enforced for that site's objects wherever they may be located in the federation when accessed.

Structural labelling

X2.1 Protectable entities: The variables and methods can be labelled; sites that wish to label entire objects can do so by labelling its facets appropriately. Sites that wish to store classes as objects can do so, in which case the class methods and variables can also be labelled.

X2.2 Label instantiation: Depends on site; new objects must be labelled when inserted at the TCC.

X2.3 Relationship restrictions: Relationship restrictions are not checked by SPOP; sites can support any restrictions they want (as long as the mandatory restrictions are included—see [10]).

Dynamic labelling

X3.1 Authorisation flow: Authorisation checked by the TCC is based on clearance of primary accessor, that is on the identity of the user who sent the concerned message. Sites can add checks based on the identification of the primary accessor and initiation details of message—see discussion later.

X3.2 Sensitivity flow: The message sensitivity is the least upper bound of the sensitivity of all variables accessed up to that point during message execution. Sites can also increase this sensitivity above this level (according to the site's security policy) when an object owned by the site is accessed.

X3.3 Information flow restrictions: SPOP aborts execution of the message when the message sensitivity exceeds the sensitivity of the variable to be written.

The site security policy

From the federal security policy it is clear that each site's security policy will have to specify (at least) the following:

1. How information is classified and how users are cleared—see [7] for a discussion of the implications if one site does not approve of the classification policy of another site.

2. Any additional checks to be performed when information owned by this site is accessed. The particular check may depend on the exact variable or method that is accessed. Checks thus performed may include content and context based checks.

3. The protection interpretation (existence protection, access protection, or other).

4. Which entities are to be protected (objects, classes, facets of objects, facets of classes, etcetera).

5. How labels are decided on—SPOP needs to know the labels of facets whenever a new object is entered into the database. However, how and when the site determines the labels before this point is up to the site.

5 The modules of the TCC

The TCC consists of nine modules. Most of these modules are intended for internal TCC use only. The only two exceptions are the user interface handler (UIH) and the external request handler (ERH).

The UIH can accept requests from the user interface to perform a computation, relocate an object from one site to another or add a new object to the system. Note that UIH is not intended for direct use by the human user; many layers of software can exist between the human user and the UIH. Such software can translate requests from the form used by the human, perform query optimisation, etcetera.

The external request handler (ERH) is intended for communication between TCCs. Each TCC includes an ERH as well as an external request sender (ERS). Whenever the TCC needs to communicate with another TCC, it formulates a request and sends it via its ERS. The request will then be received by the ERH of the target TCC, handled by the target TCC and a result returned to the requesting TCC's ERS.

In order to handle the received request, both the UIH and the ERH rely on the three 'primary' modules: the message handler, the relocator and the security information broker. The message handler receives messages intended for database objects, finds the object, performs access control checks and then causes the appropriate method to be executed. The relocator packs the object appropriately and then transfers it (via the ERS) to the appropriate site. The information broker either supplies the requested security information directly or after obtaining it from a TLE at its site, or from a TCC at another site.

The three remaining modules are 'service' modules, intended for use by the three primary modules mentioned in the previous paragraph: the memory access module, the method interpretation module and the object packing module. The memory access module (MAM) handles all accesses to variables. For example, during execution of a method, whenever a variable is to be read or written, the MAM will do it on behalf of the method. The method interpretation module (MIM) interprets the operations that make up a method. The MIM will be used by the message handler to execute methods. Note that the entire MIM need not be trusted; this will be discussed later. The object packing module (OPM) will 'pack' objects in the transfer format so that they can be sent to another site. OPM will also unpack (install) objects that have been received from others site at the local site.

The nine modules are depicted schematically in figure 3. The services offered by each of these modules will now be discussed in more detail. Discussion of the three primary modules is left for last, since they are based on the other modules. More detailed information about the operation of the various services, including arguments on the safety of the services will be given in a later section.

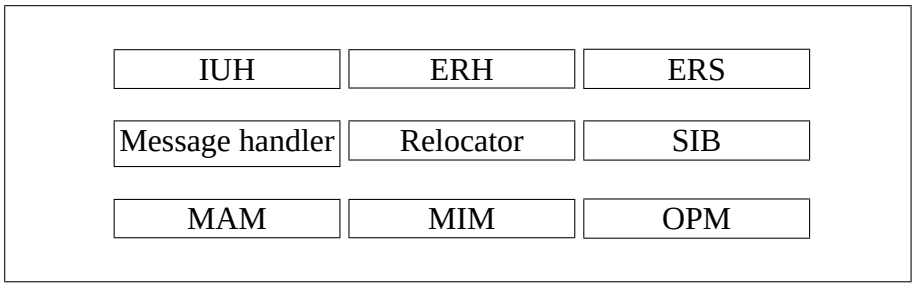

Figure 3: The nine modules of the TCC

5.1 The memory access module (MAM)

The memory access module handles all read and write operations to variables. `AppRead` and `AppWrite` are used by the message handler to read and write variables on behalf of user methods. These two services will perform the necessary access checks before reading or writing. The message sensitivity will also be updated when necessary.

`TccRead` is used by the message handler to read variables on behalf of TE methods. TE methods are not allowed to write to variables: This implies that they cannot 'leak' any sensitive information that they have access to.

5.2 The method interpretation module (MIM)

The method interpretation module has access to an array of operations that constitutes a method, as well as a stack. A pointer (instruction pointer, IP) points to the next operation to be executed. The operation can affect the values on top of the stack, as well as the value of IP. If the operation is a variable read or write, or a message send, this is referred to the message handler. Note from this that the MIM can only read methods, read and write its stack and communicate with other trusted modules of the TCC. Since the stack is cleared between independent method invocations the stack cannot be used to 'leak' information to other method invocations.

TE methods are similar to application methods in that they need to be interpreted. In contrast to application methods, TE methods may access sensitive values without increasing the sensitivity of the message. For example, user X may want to access the secret salary field of an employee. However, although user X may be cleared at a

secret level, the TE associated with the salary field may allow it only if the top secret job field is not 'spy'. The TE method therefore needs to access the top secret job field even though user X may not be authorised to access this field. In addition, the fact that the job field has been accessed, should not cause the current sensitivity of the message to be increased to top secret.

In addition to data handling that has special requirements when executing TE methods, trust in TE security depends on correct (verified) interpretation of TE methods.

For the reasons mentioned in the previous paragraphs, the MIM module contains two services: `AppInterp` interprets methods from application methods and `TccInterp` interprets TE methods.

5.3 The object packing module (OPM)

OPM can 'install' an object at the local site and also 'pack' an object so that is can be transported to another site. An object to be installed should be in the transfer format. Similarly, an existing object that is 'packed', is coverted to the transfer format.

The two services offered by OPM are `ObjPack` that 'packs' a specified object in the transfer format (that is, as an object table, value table and code table), while `ObjUnpack` 'installs' the object supplied (in the transfer format) at the local site.

Having considered the three service modules, we will now turn our attention to the three interface modules.

5.4 External request handling and sending

The external request handler (ERH) and external request sender (ERS) are mirror images of one another: For every request that can be sent by ERS a handler exists in the ERH of the target site.

The services offered by the ERS-ERH combination are

- `ErsAppMes-ErhAppMes` transfers an application message from one site to another;
- `ErsTccMes-ErhTccMes` is used to transfer messages to execute TE methods from one site to another;
- `ErsGetSI-ErhGetSI` obtains specified security information from the security information broker (SIB) at the target site.
- `ErsGetObj-ErhGetObj` transfers the requested object from the target site to the requesting site.
- `ErsSendObj-ErhSendObj` transfers the included object from the requesting site to the target site.

5.5 The user interface handler (UIH)

The user interface handler is that component that receives requests from the user interface. As indicated earlier, the user interface (on top of UIH) can consist of many layers of software, including the query language processor and optimiser.

The UIH can accept the following requests from the user interface:

- `UsrMes` accepts a message from the user interface, directed at any object in the federation.
- `UsrRel` relocates an object: the object as well as the site where the object should be relocated to, are specified.
- `UsrNewObj` gives a new object to be installed into the system. The object is described in the transfer format: an object table, a value table and a code table.

Note that `UsrNewObj` is used to 'install' all new objects, including TE methods. It is up to the site to employ its own 'extended' protection mechanisms to ensure that only authorised TE methods are installed and that other objects are installed with the proper protection. However, we will argue later that this apparent openess does not influence the security of objects owned by other sites.

The prototype, as described thus far, is depicted in figure 4. The three modules not discussed yet are the primary modules embodying most of the complexity of the TCC. These will be discussed next.

5.6 The security information broker (SIB)

The SIB is responsible for performing the federal access control checks; the SIB also answers security related questions about subjects or messages, either by using its own mechanisms, or by requesting the information from another site or from the TLE at its own site.

The SIB service that performs access control checks is `FedCheck`. `FedCheck` implements the policy specified in section 4: it aborts execution if the clearance of a user does not dominate that of an instance variable to be read or a method to be executed, or if the current message sensitivity exceeds the sensitivity of a variable to be written. `FedCheck` also initiates the TE method that implements the local security policy for the entity to be accessed. This TE method will abort execution if access to the concerned entity is not permitted by the local security policy of the owning site.

All requests to the SIB for information indicate, firstly, whether the information should be obtained from a particular site, or whether it is (global) knowledge, available from all TCCs. For example, if the SIB is requested to supply the GLOBAL:CURRENTTIME, the SIB will obtain the time from the local clock and supply that value to the requesting process. However, a process can also request the SITEONE:CURRENTTIME (from, say, the SIB at SiteTwo) in which case the SIB will send the request (via `ErsGetSI`)

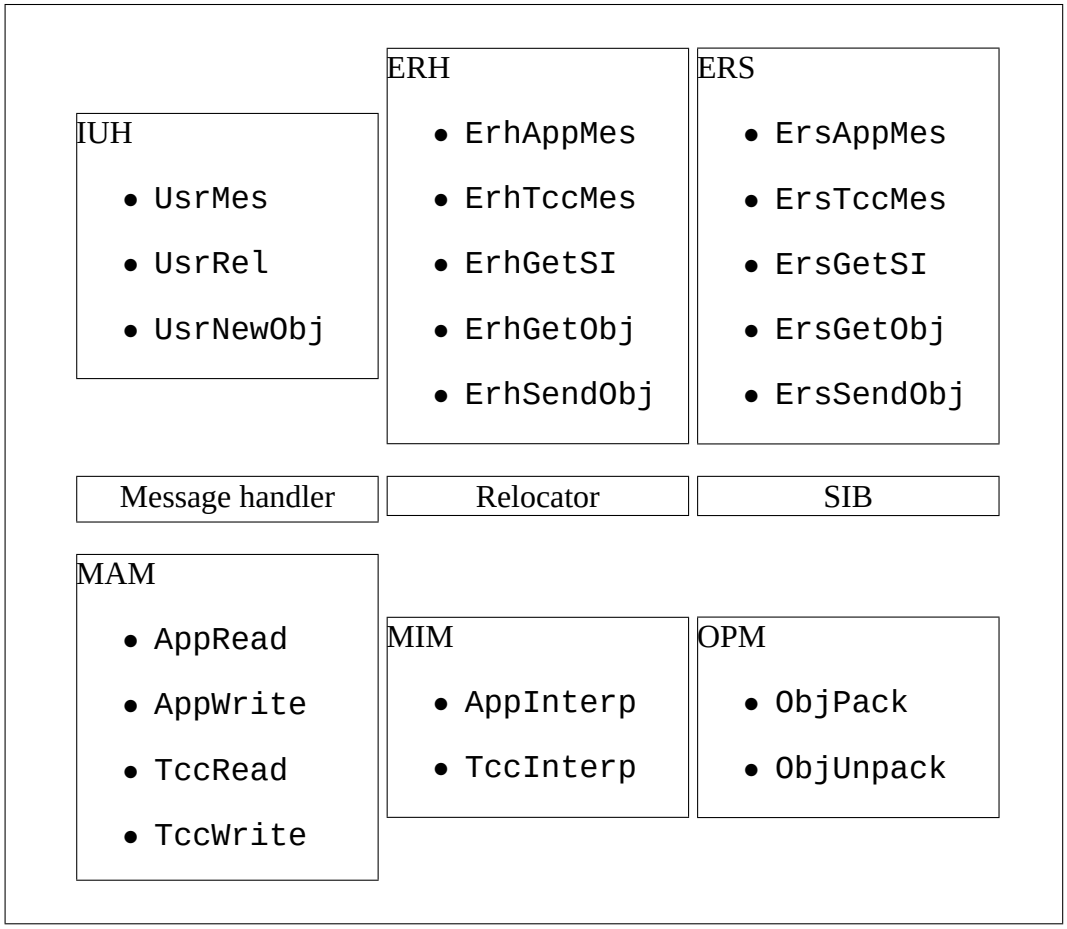

Figure 4: The service and communication modules of the TCC

to the SIB at SiteOne. At SiteOne the SIB will obtain the time from the TLE at its site, before returning it to the SIB at SiteTwo.

Obviously 'global' security information should be agreed to between all members of the federation. In contrast, 'local' security information can be supplied by any site at its discretion. For example, if a particular site only wants to grant access to an object if the user who sent the message logged in at terminal 123 at that site, then that site can define THATSITE:TERMNO as the terminal where the message was initiated. If a message is now sent to that object then, as part of the TE access checking, the terminal number where the message originated from, will be requested from that site. If the message did originate from THATSITE, THATSITE had to log the terminal number where the message was initiated and the SIB at that site can simply return it. If the message did not originate from that site, that site will not know the terminal number and return a 'not known' which will cause the access to be denied. See [7, 8] for a discussion of the complexities and possibilities surrounding local and global security

information.

For SPOP we assume that the only global security information is the time and the current message sensitivity. Further, we assume that the identification of the user who initiated a message, as well as a (federally unique) identification of this original message is part of the security information that accompanies the message. Therefore, individual sites can record information about local users, messages that originate from their site and information about the site itself and use such values in their extended access checking. (In addition, any similar information that may be obtained from other sites by mutual agreement, may also be used—see [7, 8].) The clearance level and current sensitivity level of the message may also be used.

Note that the request from the TCC to the TLE does imply that information flows from a TCC to a TLE not necessarily trusted by it. However, as argued in [7], this does not constitute a security risk.

The SIB service that supplies security related information is `SibGetInfo`; it will supply a global or local security value, given the clearance and sensitivity information of the concerned message.

5.7 The message handler (MH)

The message handler can receive requests to handle application messages or messages relating to TE access checks. The message handler deals with these two cases using two services, *viz.*

- `AppMH` handles messages sent as part of the execution of an application process. Before the message is handled, all required access checks are performed. In addition, during execution of the message the message sensitivity information is updated as required by the security policy.

  `AppMH` uses `AppInterp` to interpret application methods.

- `TccMH` handles messages that perform extended access checking. It returns a boolean value that indicates whether access should be granted or not.

  `TccMH` uses `TccInterp` to interpret TE methods. `TccMH` obtains security information by using `SibGetInfo`. If the TE method is not available locally, `TccMH` does no processing locally, but delegates the work to the remote `TccMH` via `ErsTccMes`.

5.8 The relocator

In principle, it is possible for the relocator to modify objects before relocation to enhance security [7]. It may for example be possible for a site to deny relocation of any top secret information to other sites; an object that contains top secret information will then be stripped of such information, the top secret information will not be relocated,

but rather kept at the owning site; any references made at the final site will then be referred by the modified object to the original site for (local) access control.

However, SPOP implements a rather simple version of the relocator: no modification of the object occurs before relocation. Object modification is considered in [9].

SPOP's relocator consists of a single service: `Relocate`. It takes two parameters: the identification of the object to be relocated (⟨OSID, OID⟩), and the site identification of the site where the object is to be relocated to. `Relocate` will use the directory service (see section 7) to determine the current location of the object to be relocated. If the object resides at the local site, `ObjPack` will be used to pack it; otherwise `ErsGetObj` will be used to obtain the object from its remote site. If the object is to be relocated to the current site, `ObjUnpack` will be used to install it; otherwise `ErsSendObj` will be used to send it to its (remote) destination site.

6 Safety guarantees

SPOP can be considered secure if it implements the access control policies specified by the federation and by the site that owns the particular information. Note that the SPOP security policies address confidentiality and integrity by allowing sites to limit access to variables and methods of objects. Availability is not currently addressed explicitly and will receive attention in future. This section argues that the SPOP design does implement the concerned security policies.

All the services of SPOP have access to the security information associated with the message as implicit parameters. In the current prototype it includes the message sensitivity (that indicates the sensitivity of the 'accumulated' information at any point), the identity of the user who originally sent the message and an identification number of the message.

The SPOP design requires that all services are isolated—they can only communicate via the explicit parameters, the implicit security parameters and the code tables, value tables and object tables at each site.

Any message sent by a user results in a number of services that are activated to handle the message. All such services can be divided in three categories:

1. Some services are allowed to access information the user is not cleared to access. For such services in SPOP it can be shown that they cannot (directly or indirectly) communicate such 'privileged' information to the user. They also cannot cause changes to the state of the database.

2. Some services are only allowed to access information accessible to the user in accordance with the relevant security policies. For such services in SPOP it can be shown that the required access control checks are performed before *any* access is allowed, that is before any instance variable is read or written, or any method is activated.

3. Some services are only allowed to manipulate objects as encapsulated units. For such services in SPOP it can be shown that they cannot communicate the contents of objects with users directly, they cannot directly modify the contents of objects and they cannot indirectly enable users to access objects in unauthorised ways.

It is easy to see that, if all services behave according to the requirements set for *one* of the three categories, they restrict access to the protected entities in accordance with the security policies.

For ease of reference the category 1 services will be referred to as *security services*, the category 2 services as *user services* and the category 3 services as *object management services*.

Our argument that SPOP is secure will proceed as follows: All services will be categorised into one of the three categories listed above. The fact that the requirements for category 1 methods do apply is argued in section 6.1, while the fact that the requirements for category 2 methods do apply, is argued in section 6.2. In section 6.3 it will be argued that the object management services do behave as specified by the requirements for category 3.

Table 1 contains a list of all services used in SPOP, including the category of the service and a list of all other services that uses the particular service. This information will be useful in the following sections.

Tables 2 and 3 describe the only information flows that occur inside SPOP's TCC (and also into and from the TCC). An arrow indicates that the service listed on the left of the arrow uses the services to the right of the arrow. The only information that may flow between the calling service and called services is listed as parameters alongside the name of the called service. If a parameter is labelled **in** it means that the associated service will only refer to it, but not change it; if it is labelled **out** the associated service will not refer to it, but will return a value for use by the caller; parameters labelled **inout** will be referred to by the called service and a value will be returned.

The notation used in the description of the information flow includes a vertical line to indicate that information either flows to one service or the other. The notation also uses braces to indicate repetition: it is equivalent to repeating the contents of the braces any number of times.

As an example, the `AppRead` service given in table 2 reads an instance variable on behalf of an application method. It will determine, from the application stack, which variable is to be read. The `FedCheck` service will then perform the federal security checks (which also includes activation of the site security checks). After this the variable is read from ValTab (the appropriate value table) and pushed onto the application stack.

The symbol Σ is used to indicate services requested at the user interface (that is, from the user or from software, external to the TCC, acting on behalf of the user). A double arrow ($\Rightarrow$) is used to indicate that one service activates a service on another TCC.

Module	Service	Category	Used by
MAM	AppRead	2	AppMH
	AppWrite	2	AppMH
	TccRead	1	TccInterp
MIM	AppInterp	2	AppMH
	TccInterp	1	TccMH
OPM	ObjPack	3	Relocate, ErsSendObj and ErhGetObj
	ObjUnpack	3	Relocate, ErhSendObj, ErsGetObj, UsrNewObj
ERS-ERH	ErsAppMes	2	AppMH
	ErsTccMes	1	TccMH
	ErsGetSI	1	TccMH
	ErsGetObj	3	Relocate
	ErsSendObj	3	Relocate
UIH	UsrMes	2	External use
	UsrRel	3	External use
	UsrNewObj	3	External use
SIB	FedCheck	1	AppRead, AppWrite and AppMH
	SibGetInfo	1	TccMH, ErhGetSI
MH	AppMH	2	UsrMes, ErhAppMes and AppMH
	TccMH	1	ErhTccMes
REL	Relocate	3	UsrRel

Table 1: Summary of services

The services that are underlined (such as AppRead) are those that can only handle information that the user is authorised to access (user services). Those that are not underlined are the services that handle information without prior access checks—security services in table 2 and object management services in table 3.

6.1 Security services

Security services in SPOP cannot (directly or indirectly) communicate information for which access checks have not been performed to the user. Note that security services cannot be activated directly by the user, but only by user services and other security services. Since no security service is allowed to send any information to any service, except another security service, it can be demonstrated that a security service cannot communicate information with a user. In particular, it is easy to verify that security

services cannot write any information to any variable, except a local variable or a parameter (see table 2). Since local variables are only accessible to the service itself, it is only necessary to verify that no information 'leaks' via parameters to non-security services. The fact that no information flows via any explicit parameters to non-security methods is easy to verify: `FedCheck` is the only security service used by non-security services (see tables 1 and 2) and its parameter only allows information to flow from the user service to the security service (an not vice versa).

Security information does flow between security services and user services (as implicit parameters). It is easy to verify that no user service accesses the security information.

Note that `SibGetInfo` can communicate with the TLE. See section 5.6 and [7, 8] for more information and an argument that this communication does not pose a security risk.

6.2 User services

For user services in SPOP the required access control checks are performed before any access is allowed, that is before any instance variable is read or written, or any method is activated. Only `AppRead` can read a variable on behalf of a user service, while only `AppWrite` can write a variable on behalf of a user service. (`TccRead` is the only other service with access to the value tables. `TccRead`, however, is a security service and it has been argued above that no security service can communicate such information with any user service.) Similarly, `AppMH` (using `AppInterp`) is the only service that can interpret methods on behalf of a user service. (`TccInterp` is the only other service with access to the code tables, but `TccInterp` is a security service which means that it cannot modify instance variables or return a result to any user service. Therefore, although `TccInterp` can potentially interpret a user method, it cannot cause any effect usable by a user.) It is easy to verify that any user service that reads an instance variable (with `AppRead`), writes an instance variable (with `AppWrite`) or initiates a method (with `AppMH`) will only do so after the access checks according to the security policies have been performed: In all three cases the services call the `FedCheck` service, which performs the multilevel security check and then initiates the site check, represented by the associated TE method, before allowing access.

The security checks associated with the three services given above will ensure that any information accessed by it is authorised and that the security information will be correctly updated by these three checks. In addition it is expected that each site correctly initiates the security information when the user request is entered into the system [Σ(**inout:**Val) $\rightarrow$ `UsrMes`(**inout:**Val)]. (Since each site is free to decide on such information autonomously, this may not be an unreasonable expectation. However, in some federations such information may be maintained—and certified—by some federally trusted service, such as Kerberos.) It is simple to verify that user methods can only obtain information from any other user services (see tables 1 and 2). It therefore follows by induction that all information accessed by any user service is authorised

FedCheck(**in:**EID, **in:**RWE) → read(ObjTab[EID], Sens, TccMethNo) mls-check TccMH(**in:**TccMethNo)

AppRead(**inout:**AppStack) → read(AppStack, VarNo) FedCheck(**in:**VarNo, 'read') read(ValTab[VarNo], Val) write(AppStack, Val)

AppWrite(**inout:**AppStack) → read(AppStack, VarNo, Val) FedCheck(**in:**VarNo, 'write') write(ValTab[VarNo], Val)

TccRead(**inout:**TccStack) → read(TccStack, VarNo) read(ValTab[VarNo], Val) write(TccStack, Val)

AppInterp(**in:**Method, **inout:**AppStack) → { read(Method, Operation) | read(AppStack, Val) | write(AppStack, Val) }

TccInterp(**in:**MethNo) → new(TccStack) { read(CodeTab[MethNo]) | TccRead(**inout:**TccStack) | read(TccStack, Val) | write(TccStack, Val) | SibGetInfo(**inout:**Val) } TE-check dispose(TccStack)

ErhAppMes(**in:**MethNo, **inout:**Parms) → new(AppStack) write(AppStack, Parms) AppMH(**in:**MethNo, **inout:**AppStack) read(AppStack, Parms) dispose(AppStack)

ErhTccMes(**in:**MethNo) → TccMH(**in:**MethNo)

ErhGetSI(**inout:**Val) → SibGetInfo(**inout:**Val) | ErsGetSI(**inout:**Val)

ErsAppMes(**in:**MethNo, **inout:**Parms) ⇒ ErhAppMes(**in:**MethNo, **inout:**Parms)

ErsTccMes(**in:**MethNo) ⇒ ErhTccMes(**in:**MethNo)

ErsGetSI(**inout:**Val) ⇒ ErhGetSI(**inout:**Val)

UsrMes(**in:**MethNo, **inout:**Parms) → new(AppStack) write(AppStack, Parms) AppMH(**in:**MethNo, **inout:**AppStack) read(AppStack, Parms) dispose(AppStack)

SibGetInfo(**inout:**Val) → { TLE.GetAttr(**inout:**Val) | TLE.CheckAttr(**inout:**Val) | ErsGetSI(**inout:**Val) }

AppMH(**in:**MethNo, **inout:**AppStack) → FedCheck(**in:**MethNo, 'execute') read(CodeTab[MethNo]) { AppInterp(**in:**Method, **inout:**AppStack) | AppRead(**inout:**AppStack) | AppWrite(**inout:**AppStack) | [read(AppStack, CalledMeth) AppMH(**in:**CalledMeth, **inout:**AppStack] | [read(AppStack, CalledMeth, Parms) ErsAppMes(**in:**CalledMeth, **inout:**Parms) write(AppStack, Parms)] }

TccMH(**in:**MethNo) → TccInterp(**in:**MethNo)

Σ(**in:**MethNo, **inout:**Parms) → UsrMes(**in:**MethNo, **inout:**Parms)

Table 2: Information flow to handle user messages

`ObjPack`(**out:**Obj) → read(ObjTab) read(CodeTab) read(ValTab)
`ObjUnpack`(**in:**Obj) → write(ObjTab) write(CodeTab) write(ValTab)
`ErhGetObj`(**out:**Obj) → `ObjPack`(**out:**Obj)
`ErhSendObj`(**in:**Obj) → `ObjUnpack`(**in:**Obj)
`ErsGetObj`(**in:**Obj) ⇒ `ErhGetObj`(**in:**Obj)
`ErsSendObj`(**out:**Obj) ⇒ `ErhSendObj`(**out:**Obj)
`UsrRel`(**in:**Val) → `Relocate`(**in:**Val)
`UsrNewObj`(**in:**Obj) → `ObjUnpack`(**in:**Obj)
`Relocate`(**in:**Val) → [`ObjPack`(**out:**Obj) | `ErsGetObj`(**out:**Obj)] [`ObjUnpack`(**in:**Obj) | `ErsSendObj`(**in:**Obj)]
Σ(**in:**Val) → `UsrRel`(**in:**Val)
Σ(**in:**Obj) → `UsrNewObj`(**in:**Obj)

Table 3: Information flow for object management

and the security information correctly reflects the information accessed.

6.3 Information flow for object management

Object management services are only allowed to manipulate objects as encapsulated units. For such services in SPOP it has to be shown that they cannot communicate the contents of objects with users directly, they cannot directly modify the contents of objects and they cannot indirectly enable users to access objects in unauthorised ways.

`ObjPack` and `ObjUnpack` are complimentary services: any object 'packed' into the transfer format by `ObjPack` will be 'unpacked' to an identical representation by `ObjUnpack` albeit usually in an object table, value table and code table at a different site. In particular, the same sensitivities and TE methods will still be associated with the concerned instance variables and methods.

Object management services cannot communicate the contents of variables directly with users since no parameters exist for such communication. Neither does any mechanism exist to initiate any method of any object. To show that information be indirectly made accessible to a user, it is necessary to show that the correct access checks will be performed. In the case of relocation it can be shown that exactly the same access checks (as at the initial site) will be performed before access at the new location is permitted. Assume that a method M_1 initiates a method M_2, where M_2 is a facet of an object that has been relocated. If M_1 and M_2 resided on the same site prior to the relocation the sequence of services to handle M_2 would have been

$\underline{\texttt{AppMH}}_1 \rightarrow \underline{\texttt{AppMH}}_2$

where $\underline{\texttt{AppMH}}_1$ is the instance of $\underline{\texttt{AppMH}}$ that would have handled M_1 and $\underline{\texttt{AppMH}}_2$ the

instance that would have handled M_2. After the relocation M_1 and M_2 presumably do not reside on the same site anymore and the sequence of services will be

$\underline{\texttt{AppMH}}_1 \rightarrow \underline{\texttt{ErsAppMes}} \Rightarrow \underline{\texttt{ErhAppMes}} \rightarrow \underline{\texttt{AppMH}}_3$

It has been argued earlier that the security information associated with M_2 will not be changed by relocation. Since none of the services in the new sequence can modify the message security information the security information used by `FedCheck` when called by $\underline{\texttt{AppMH}}_3$ will be exactly the same as that which would have been used by $\underline{\texttt{AppMH}}_2$. The same access decision will therefore be made at the new site of M_2. Similar arguments can be used for the case where M_1 and M_2 resided on different sites before relocation, as well as for instance variables.

Insertion of new objects (with `UsrNewObj`) by the TCC can also be trusted because the new object is inserted precisely as specified by the site where it is inserted. We do not give a detailed argument.

6.4 Trusted implementation

The argument above that the design of SPOP is secure obviously does not imply that a SPOP implementation is necessarily secure—it will only be secure if all the specifications of the design are taken into account in the implementation. In particular, the arguments applied above for each of the services can again be used to verify that the code conforms to the requirements. This includes aspects such as verifying that the code that implements a security service does not write to any non-local variables (and that local variables do not retain values between invocations), verifying that user services do perform the required access checks at the indicated points and do not utilise security information, etcetera.

Note that most services will not need much more code than that specified in tables 2 and 3; in fact most services can be coded in a dozen or fewer lines of code. Verification that the implementation conforms to the design is therefore simple. Only `AppInterp` is relatively long, but the extent to which it needs to be verified is very small: Since it only manipulates data on the application stack, it can only access data for which access checks have already been performed. It is therefore only necessary to verify that it does not 'remember' any information from one call to the next and that it does not communicate data to other services—except on the application stack.

7 Conclusion

We have presented the design of a prototype using self-protecting objects and argued that it can be trusted. The paper has shown that it is indeed possible to construct a system that uses the SPO model. It has further been shown that the inherent complexity of SPO is not high: The current SPOP design uses 28 services organised in nine modules. Most of the services are simple enough to be coded in a small number of lines.

It is still not clear how to handle a number of remaining problems. Firstly, SPOP uses multilevel security, which means that, as long as sites are satisfied with one another's assignment of clearances, the authorisation of a subject is known throughout the federation. It is not as simple when discretionary security is used. Is it reasonable to expect all sites to individually grant permissions to all authorised federation users in such a case? See [8] for a discussion of these issues.

Secondly, the support of legacy databases needs attention. In [9] a first attempt has been made to support heterogeneous databases. A similar approach can possibly be used to incorporate legacy databases in a secure federation.

Lastly, support services need further attention. In particular, the use of the SPOP database itself to store directory information looks promising.

References

[1] BT Blaustein, CD McCollum, A Rosenthal, KP Smith and L Notargiacomo, Autonomy and Confidentiality: Secure Federated Data Management, in: A Motro and M Tennenholtz, eds, *Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems*, Naharia, Israel (1995) 59–68.

[2] A Goldberg and D Robson, *Smalltalk 80: The Language and its Implementation* (Addison-Wesley, 1983)

[3] L Gong and X Qian, The complexity and Composability of Secure Interoperation, *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, Oakland, California (1994) 190–200.

[4] D Jonscher and KR Dittrich, An Approach For Building Secure Database Federations, *20th VLDB Conference*, Santiago, Chile (1994).

[5] W Kim, ed, *Modern Database Systems: The Object Model, Interoperability and Beyond* (Addison-Wesley, 1995).

[6] TF Lunt, Authorization in Object-Oriented Databases, in: W Kim, ed, *Modern Database Systems: The Object Model, Interoperability and Beyond* (Addison-Wesley, 1995) 130–145.

[7] MS Olivier, "Self-protecting Objects in a Secure Federated Database", Ninth IFIP WG 11.3 Conference on Database Security, Rensselaerville, New York (August 1995).

[8] MS Olivier, "Supporting Site Security Policies for Members of Federated Databases", *Fourth European Conference on Information Systems* (Lisbon, 1996).

[9] MS Olivier, “Secure Mobile Nodes in Federated Databases”, *Submitted*, 1996.

[10] MS Olivier and SH von Solms, “A Taxonomy for Secure Object-oriented Databases”, *ACM Transactions on Database Systems*, **19**, 1 (1994) 3–46.

[11] MT Özsu and P Valduriez, *Principles of Distributed Database Systems*, (Prentice-Hall, 1991).

[12] G Pernul, “Canonical Security Modelling for Federated Databases”, in: DK Hsiao, EJ Neuhold and R Sacks-Davis, eds, *Interoperable Database Systems*, Elsevier, (1993) 207–222.

[13] F Rabitti, E Bertino, W Kim and D Woelk, A Model of Authorization for Next-Generation Database Systems, *ACM Transactions on Database Systems*, **16**, 1 (1991) 88–131.

[14] AP Sheth and JA Larson, Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases, *ACM Computing Surveys*, **22**, 3 (1990) 183–236.

[15] B Thuraisingham, Security issues for federated database systems, *Computers & Security*, **13** (1994) 509–525.