RJ2041(28513)7/18/77
Computer Science

DATA BASE SYSTEM AUTHORIZATION

D. D. Chamberlin
J. N. Gray
P. P. Griffiths
I. L. Traiger
B. W. Wade

IBM Research Laboratory
San Jose, California 95193

ABSTRACT: Authorization of users and programs to access data management objects such as transactions and data is discussed. First, authorization common in existing Data Base Data Communication systems is sketched. The trend towards decentralization of the administration of such systems is then discussed. Finally, several architectural issues are considered, and a design to cope with decentralization is presented. This paper discusses an architecture which is novel in that it allows "ordinary" users to perform authorization functions usually reserved for the data base administrator. The issues of value dependent authorization, authorization contexts, and granting and revocation are considered. The paper discusses issues and refers to other papers for algorithmic solutions.

IBM Research Division
Yorktown Heights, New York · San Jose, California · Zurich, Switzerland

I. INTRODUCTION

This paper focuses on the rather specialized topic of authorization of access to a Data Base-Data Communication system (DB-DC system.) Many DB-DC systems currently need little authorization beyond that provided by the operating system (e.g. in-house or one-person data bases). However, there is a growing class of large and sophisticated data management systems which require tight controls over the use and dissemination of data.

The next section discusses how large existing (commercially available) systems appear to do authorization. The remaining sections suggest improvements to these mechanisms.

II. A TYPICAL SYSTEM

User Roles

Large DB-DC systems typically have several roles for users. The broad roles are:

- System Administrator: defines and installs the system. Makes policy decisions about the operation of the system.
- System Operator: handles the operation of the system, manages system startup-shutdown, responds to user requests, and manages physical plant.
- System Programmer: installs and maintains the DB-DC code, and the underlying operating system.
- Application Programmer: defines and implements new application programs to be used by end users, by the system administrator, and by the system operator.
- End User: uses the system to enter and retrieve data.

These roles are (typically) further refined into sub-categories (e.g. end users include the roles: teller, loan officer, branch manager, auditor,...). Over time, a particular user may perform several of these roles, but usually a user is authorized to only one role.

The concept of role serves the purpose of grouping users of the system together, thereby decomposing authorization decisions into the two questions:

- what should the authorization of a role be?
- who should be authorized to use the role?

Authentication

Individuals sign on to the system in a particular role. The individual's identity is validated by a combination of

- Personal identification (key, magnetic stripe, password,...)
- Physical location of terminal (teller must be at own bank,...)
- Physical security of terminal (it is in a bank,...)
- Time of day (bank teller terminals only work at certain hours,...)

This mechanism is usually specified as a decision table so that it is easy to understand. In the above instance the decision table would be:

PERSON X PASSWORD X TERMINAL X TIME -> ROLES

Transactions

Once a person establishes a role, he is authorized to perform certain transactions on the system. There is great variety among the transactions available to different roles. Someone on the shipping dock will have a different set of transactions than a member of the purchasing department. So there is a further table which authorizes

ROLE -> TRANSACTION.

An installed transaction's definition carries a complete list of the objects it accesses (except for objects passed as parameters to the transaction such as input and output terminal or queue.) The transaction is strictly limited to this domain when executed.

When the transaction is invoked the data management system constructs a domain consisting of only these objects and operations. This domain is usually represented as a set of control blocks (one per object) in protected storage. Since these control blocks are in protected storage, they perform the functions of capabilities <3>. All operations by the transaction name one or more of these objects (control blocks). This limits what objects can be touched by the transaction. The control blocks further limit what operations may be performed on the

object (e.g. a file may be read-only.)

Authorization aspects of roles

We now discuss authorization as viewed by each generic role.

End users are usually limited to pushing buttons which cause forms to appear on the display screen. After filling in a form, another button causes the form to be validated, and if it passes the test, to be acted upon by the system.

The application programmer defines (implements) transactions. Depending on the degree of care exercised by the programmer, he may be able to prevent the users from doing terrible things to the data base. For example, the transaction might refuse to handle withdrawals of more than five hundred dollars without the branch manager's approval. In general, the application programmer seeks only to guard against end user abuses and mistakes.

The application programmer might be able to protect the privacy of his data from the system operator and from the data base administrator by encrypting it. Communication over insecure lines is often unprotected. Some systems do encryption/decryption in order to protect the security of communication data. For example, some cash dispensing terminals encrypt customer passwords and transaction information when communicating with the central host. However, the usefulness of local encryption of data residing within a host is doubtful at present because data appear in the clear while in main storage (accessible to almost anyone reading a dump), and because operational personnel usually have a back door to the authorization system. Lastly, there are technical problems associated with constructing indices on encrypted data and keeping a system log that contains encrypted log records. In summary, the application programmer must trust the system administrator, system operator, and system programmer.

The system administrator defines system objects and authorizes access to them. The principle system objects are users, terminals, transactions, views, and physical files.

When installing a new transaction, the system administrator is careful to validate the program and to narrowly describe the subset of the data and terminals available to the transaction. Installing a transaction consists of entering its programs and descriptors into system catalogs and authorizing one or more roles to use the transaction.

In order to proscribe the domain of a transaction, the system administrator:

- Limits the transaction to access a particular set of files.
- Within each file, makes only certain record instances visible.
- Within a record instance, makes only certain fields visible.
- Makes only certain visible fields updatable.

Continuing the example above, a bank teller transaction might be allowed to see only those records from the central ledger which pertain to the local branch and be allowed to update only the balance field of the teller cash drawer records.

Aside from deciding which transactions and files will be stored and what access paths will be maintained on files (indices, hash chains, sibling pointers, etc.), the system administrator also installs exits (data base procedures) which enforce the integrity of the system. Examples of exits are:

- Exit to encrypt-decrypt objects on secondary storage.
- Exit to validate the reasonableness of the contents of records being inserted into the data base.
- Exit to check the authorization of the caller to manipulate the objects named in the DB-DC operation.

These exits allow an installation to tailor the system to perform authorization appropriate to its application.

The system operator is limited to a very special set of commands which allow him to manage the physical resources of the system, to restart the network and the system, and to reconfigure the network and system. In point of fact, the system operator has the "keys to the kingdom" and can easily penetrate the system.

Similarly, the system programmer has a very limited set of commands. However, to ease debugging and maintenance, one or more of these commands opens almost any door in the system.

In general, users are not unhappy with the rather primitive access control mechanisms outlined above. In general, user level authorization is quite application dependent (e.g. only the military seems to understand or care about the star property

<4>.) Hence, there seems to be general agreement that this authorization should be imbedded in exits or in application code rather than being included in a general purpose DB-DC system.

III. PROBLEMS AND TRENDS

Why we expect things to change

At present, most "real" systems are doing "operational" processing. They are very static applications which have automated the "back-office" of some large enterprise. Usually, human tasks were directly replaced or augmented with computers. These applications are often prescribed by law or accounting practice and are reasonably well understood.

Computers are moving into the "front office" and into smaller operational units. At first these systems will be small and isolated (i.e., stand alone mini-computer). But eventually these systems will be integrated with the "back office" system and with other front office systems. This implies that networks of loosely coupled systems will appear.

When this happens, one should expect the system to be much less static and expect control of the system to be much less centralized.

The Case Against A Central System Administrator

The definition and control of objects (transactions, users, queues, data bases, views, catalogs,...) has been a highly centralized function residing with a single individual or group of individuals (system administrator). Several trends encourage the development of a less centralized administrator function.

- The foremost trend is that systems are becoming much more dynamic. A large system typically has several groups of users. Each group wants to share a central pool of data and perhaps share data with some other groups. But also it wants to be able to easily create and maintain private data and transactions. The requirement that all new definitions be funneled through a central system administrator is quite restrictive as well as inconvenient.

- The existence of a central system administrator also has the psychological drawback that the "owner" of the data does not control it. Rather, the system administrator controls it.

- Independent data management systems are being integrated in order to selectively share information among co-operating organizations. Even if the network is homogeneous (similar machines and data management systems), each node of the network is an autonomous unit with its own management and procedures. This is because networks cross organizational lines and yet responsibility for the data at a node ultimately rests with the organization which maintains that node.

Sketch of a decentralized administrator function

Our goal is a simple mechanism to dynamically create and share objects among users of a data management system. This simplicity is important for a community of individuals who control their own data, as well as for a more centrally controlled system where authorization is handled by a (human) data base administrator.

We have been able to draw on the experience and techniques used in operating systems used for authorization to files. However, we have had to refine these facilities because more semantics are associated with the objects.

We have been trying to design a decentralized authorization mechanism which provides the following functions: The system administrator function is distributed among all application programmers and even to some end users. The central system administrator allocates physical resources (space and time) and grants some transactions to particular users thereby delegating his authority to others. These objects include views of system catalogs and transactions which install new users and new terminals and new space.

Much as in a traditional operating system, the user has a catalog of named objects he can manipulate and use. Objects come in three general flavors:

- Data objects: physical files and logical files (views).
- Communication objects: logical ports and message queues.
- Transaction objects: encapsulated (parameterized) programs which perform operations on data objects and communication objects.

Each object type has a set of operators defined on it and these operators are individually authorized.

A user with no transactions in his catalog can do nothing. The catalog of a minimally privileged user consists of a limited set of transactions which may be invoked. The catalog of an application programmer might contain transactions (commands) which allow him to define new objects and grant them to others. Each time a user defines a new object, an authorized entry for it is placed in his catalog.

The authors of the system implement transactions which allow the invoker to define system objects. These transactions include:

- DEFINE_USER: enrolles a new user in the system.
- DEFINE_TERMINAL: makes a new terminal known to the system.
- DEFINE_FILE: defines a new physical file.
- DEFINE_INDEX: defines an index on some file.
- DEFINE_LINK: defines a N-M mapping from records to records.
- DEFINE_VIEW: defines a new view in terms of existing files.
- DEFINE_TRANSACTION: defines a new transaction in terms of existing data objects.

Other transactions are available to MODIFY, DROP, GRANT and REVOKE objects.

Each of these transactions may be invoked from the terminal or from a program so long as the invoker is authorized to run the transaction. Other commands are available to MODIFY definitions and to DROP definitions and their associated objects. Currently we propose that only the creator of an object can modify it and only the creator or the person who enrolled him in the system can drop an object.

By selectively granting these transactions to users, the system can delegate the function typically thought of as "system administrator" to autonomous individuals.

Once an object is defined, the creator may grant other users access to the object (subject to constraints explained below). A unique set of authorities is associated with each object type. Individual users are granted subsets of these authorities. Each authority has two possible modifiers:

- GRANT: the ability to grant another user this authority to this object (this is a property of a granted privilege to the object rather than being a property of the object itself.)

- REVOKE: the ability to selectively revoke this authority. (Only the grantor may revoke access so REVOKE implies GRANT.)

For example, RUN authority for a transaction may be granted in the following modes:

- RUN & GRANT & REVOKE

- RUN & GRANT & ¬REVOKE

- RUN & ¬GRANT & ¬REVOKE

It is not clear that one need distinguish GRANT and REVOKE in which case it might be called CONTROL. Chamberlin <2> and Griffiths <5> discuss these authorities in greater detail.

Authorization to data.

Authorization to data objects has traditionally consisted of making certain files, records, fields invisible (a subset of the data base). Much finer control can be obtained if one can, in addition,

- Do value dependent authorization (i.e. only fetch or replace record instances whose field values satisfy certain criteria).

- Define views (virtual files) which are not physically stored but are synthesized from existing stored files.

Files and view objects have the additional authorities:

- READ: the ability to read records.

- INSERT: the ability to insert records.

- DELETE: the ability to delete records.

And for each field of the file or view:

- UPDATE: the ability to update values in this field.

The justification for providing update authorization on individual fields rather than on the entire view is that some fields within a record are more sensitive than others. For example one might be allowed to read and update the QUANTITY_ON_HAND but only to read the UNIT_PRICE field. The view as a whole carries the authorizations for READ, INSERT, and DELETE. As explained in the preceding section each of these authorities potentially has the modifiers GRANT and REVOKE.

The definer of a file is fully authorized to the file. The definer of a view gets the "intersection" of the authorizations he has to its components. For example, if a user has only read authorization to a file then any view he defines based on that file will be (at most) read only.

Each user catalog is really a view of the system catalog file. Each user gets a view of his subset of the catalog and some transactions which display and modify his view of the catalog (DEFINE and GRANT insert entries in the catalog; DROP and REVOKE remove entries from the catalog). The user's view is qualified in that he cannot directly modify some fields in the catalog (e.g. authorization fields). He may be given GRANT authority on individual authorities of his view so that he can grant other users selective access to his view. If he wants to grant access to a subset of his catalog, he can define a new view which subsets his catalog and then grant the subset view to others, or he may define a transaction which accesses his catalog and then grant that transaction to others.

Only the system administrator has a view of the entire catalog.

The paper by Chamberlin <2> discusses the virtues and problems of views in detail. Stonebraker <6> presents another approach to views and proposes an interesting implementation.

Transaction authorization

One reason for defining transactions is to encapsulate objects so that others may use them without violating the integrity of the constituent objects.

Transactions have the additional authority:

- RUN: the ability to run a transaction.

Just as for views, a transaction RUN authority has the modifiers GRANT and REVOKE. If the transaction definition consists entirely of objects grantable by the transaction definer, then the transaction will be grantable. Otherwise, the definer gets the transaction with RUN&¬GRANT&¬REVOKE authority.

If a transaction is held RUN&GRANT, the definer can grant RUN&¬GRANT authority to others, who can then run the transaction. He can also grant others the ability to grant run authority by granting RUN&GRANT authority.

For example, a banking system provides transactions which credit and debit accounts (according to certain rules) rather than granting direct access to the accounts file. This effectively encapsulates the procedures of the bank and insures that all users of the data follow these procedures. An application programmer would write the transactions and grant run authority to the tellers of the bank and grant RUN&GRANT authority to the branch managers so that they could authorize new tellers at their branches.

Authorization times.

The authorization of a transaction can be done at any of three times:

- Definition: The text and environment of the transaction is described by the application programmer.

- Installation: The transaction is made known to the system.

- Invocation: The transaction is "used" by the end user.

For reasons of efficiency, authorization should be done as early as possible in this process. If possible, no authorization tests are performed at invocation time (except for validation that earlier authorization decisions have not been revoked).

When defining a transaction, the application programmer has some notion of what objects the transaction will touch and what operations will be performed on these objects (e.g. get message from queue "A", put record in file "B"). Further, he has some notion of what is allowed on the data (e.g. one should not debit an account to a negative value). The application programmer includes these tests in his program and at invocation the transaction aborts or takes remedial action if the tests are

violated.

When the transaction is installed the ability of the author to access the objects the transaction references is checked. Also the operations themselves are authorized (e.g. read authority is required on the account number and balance fields and update authority is required on the balance field). This checking is done by a program which examines the transaction text, discovering what calls the transaction makes. If everything is OK the processor enters the transaction in the system catalog along with a descriptor of the transaction domain.

If authorization fails, there are two possible alternatives: one can abort the operation, or one can defer the operation, giving a warning message. We propose to defer when authorization fails at definition or installation time. An attempt to actually operate on an unauthorized object fails. This philosophy allows programmers to define and install views and transactions which make unauthorized calls. These transactions may even be run so long as the unauthorized calls are not actually executed. As will be seen, the logic for run-time authorization must be present anyway so this decision adds little to the system complexity. The approach has the virtue that it detects many authorization errors rather than only the first.

When the transaction is invoked, the invoker's authorization to invoke the transaction is checked. When the transaction runs, both the system and the application program do value dependent authorization. For example, if a view is qualified by a selection criterion then each record which is fetched or stored via the view must satisfy that criterion. As another example, the application program may refuse to insert user-provided data which does not satisfy application-dependent criteria.

Given this motivation, it is clear that the authorization of the transaction may be different from the authorization of the invoker of the transaction.

Authorization environments

When an application programmer installs a transaction which is to be granted to another user there is some question as to which authorization environment should be used to authorize the transaction. Candidate authorization environments are:

(a) Authorize the transaction in the environment of the definer (application programmer).

(b) Authorize the transaction in the environment of the user.

(c) Sometimes (a), sometimes (b) and sometimes (a) and (b).

It might seem obvious then that the transaction should be authorized in the context of the definer. However, if the transaction is parameterized then access to parameters must be authorized in the environment of the invoker of the transaction. Similarly, if the program is a "shell" which takes in user commands and executes them, then certain of its actions should be authorized in the context of the user.

Perhaps the most extreme example of this is a program call User Friendly Interface (UFI) in System R <1>. The UFI is a program which accepts data base requests in symbolic form, translates them into system calls, and executes them against the invoker's data base. It is a combination data-base-editor and report generation language. The authors of UFI have no idea what files it will be used with or what operations may be performed on these files. All its authorization comes from the user of UFI. Clearly UFI calls to the system must be evaluated entirely in the context of the invoker.

As another example, consider authorization to objects which are created by the transaction at run time. In some cases, (e.g. an internal scratch file) the invoker should not be able to see the object while, in other cases (the report file) the invoker should be allowed to see the object. In general it seems best to attribute these transient objects to the definer who can then GRANT them to the invoker as part of the transaction logic if he so chooses.

The general rules seem to be:

- Perform authorization tests as soon as possible.
- Authorization of an operation known at installation should be done in the context of the object definer at installation time.
- Authorization to operations not known at installation (e.g. parameters) must be done at transaction invocation.

- The transaction runs in a new authorization context which is a synthesis of objects granted it by the object definer and objects granted by the object invoker.

Revocation and redefinition

As explained above, one may grant another user type "x" authority to an object if the grantor has grant authority to the type "x" authority of the object. Any subset of grantable authorities may be granted together. These authorities may then be revoked individually. (e.g., grant read authority to a view and selectively grant write authority to individual fields of the view).

The problem of revoking access to objects is very difficult. When a object is destroyed, it is deleted from the catalog of all users to whom it was granted. This also invalidates all objects which derive from that object and authorizations on them, recursively. When someone with revoke authority modifies the authorization of an object, that modification is propagated to all objects derived from it. One may selectively revoke access to the object. For example:

```
REVOKE HIRE_EMPLOYEE FROM JONES;
```

revokes Jones' access to the HIRE_EMPLOYEE transaction.

One may imagine objects organized into a dependency hierarchy. If one object is defined in terms of another, then changes in the parent will affect the child and all its descendants.

Proper implementation of this notion is very subtle. The problem is further discussed and a solution is presented by Griffiths <5>.

III. References

<1> Astrahan, Blasgen, Chamberlin, Eswaran, Gray, Griffiths, King, Lorie, McJones, Mehl, Putzolu, Traiger, Wade, Watson. System R: Relational Approach to Database Management, ACM TODS, Vol. 1, No. 2, June 1976. pp. 97-137.

<2> Chamberlin, Gray, Traiger. Views, Authorization and Locking in a Relational Data Base System. ACM National Computer Conference Proceedings, 1975. pp. 425-430.

<3> Dennis and Van Horn. Programming Semantics for Multiprogrammed Computations. CACM Vol. 9, No. 7, July 1977. pp. 145-155.

<4> Bell, LaPadula. Secure Computer Systems. ESD-TR-73-278. (AD 770768, 771543, and 780528). MITRE, Bedford Mass., Nov.

(b) Authorize the transaction in the environment of the user.

(c) Sometimes (a), sometimes (b) and sometimes (a) and (b).

It might seem obvious then that the transaction should be authorized in the context of the definer. However, if the transaction is parameterized then access to parameters must be authorized in the environment of the invoker of the transaction. Similarly, if the program is a "shell" which takes in user commands and executes them, then certain of its actions should be authorized in the context of the user.

Perhaps the most extreme example of this is a program call User Friendly Interface (UFI) in System R <1>. The UFI is a program which accepts data base requests in symbolic form, translates them into system calls, and executes them against the invoker's data base. It is a combination data-base-editor and report generation language. The authors of UFI have no idea what files it will be used with or what operations may be performed on these files. All its authorization comes from the user of UFI. Clearly UFI calls to the system must be evaluated entirely in the context of the invoker.

As another example, consider authorization to objects which are created by the transaction at run time. In some cases, (e.g. an internal scratch file) the invoker should not be able to see the object while, in other cases (the report file) the invoker should be allowed to see the object. In general it seems best to attribute these transient objects to the definer who can then GRANT them to the invoker as part of the transaction logic if he so chooses.

The general rules seem to be:

- Perform authorization tests as soon as possible.
- Authorization of an operation known at installation should be done in the context of the object definer at installation time.
- Authorization to operations not known at installation (e.g. parameters) must be done at transaction invocation.

1973.
<5> Griffiths, Wade. An Authorization Mechanism for a Relational Database System. Proceedings ACM SIGMOD Conference, June 1976.
<6> Stonebraker. Implementation of Integrity Constraints and Views by Query Modification. ACM SIGMOD Conference. May 1975. pp. 554-556.