Design Documentation for the SINTRA Global Scheduler

Myong H. Kang

Center for High Assurance Computer Systems
Naval Research Laboratory
Washington D.C. 20375

Rodney Peyton

Kaman Sciences Corporation
Locus Directorate
Suite 100
2560 Huntington Ave
Alexandria, VA 22303

ABSTRACT

In this report, we present the detailed description of the SINTRA[1] global scheduler. The detailed description includes: (1) the replica control algorithm, (2) design descriptions, and (3) rational behind our decision to choose specific methodology, an implementation language, and software engineering principles.

---

1. Secure INformation Through Replicated Architecture

1. Introduction

Database security researchers have proposed several approaches to providing users with different security clearances the capability to perform database operations securely on information classified at different security levels. These approaches provide a multilevel view of information to which the user has legitimate access and then perform database operations to respond to the user's query. Most research has concentrated on schemes for materializing this multilevel view from single-level fragments stored in files. Decomposing multilevel relations into single-level relations in a way that assures that the composition of the fragments is the same as the user's view of the multilevel relation in every case has presented many challenges for preserving database functionality as well as providing the required security. The SeaView [Lun90] approach leads to the generation of classified information whose classification is not a result of the sensitivity of the information but is instead an artifact of the decomposition approach. Materializing multilevel relations from single level base relations requires fairly complex mechanisms, and results in some performance degradation.

These concerns led to the initiation of a project to investigate replication as a promising alternative to achieve a MLS database system (DBS). The Secure INformation Through the Replicated Architecture (SINTRA) project uses physical separation as a protection measure. A Trusted Front End (TFE) mediates access to separate untrusted backend DBSs (UBD) for each security class. Each backend DBS contains information at a given class and replicated information from each lower backend database. A user has access to all information that he can legitimately view.

The replicated approach has several advantages. Each user can access all information he is authorized to see in a timely manner. Good performance and full database capabilities are project goals. Since the SINTRA system does not materialize the user's view from single-level relations, this system should lead to high performance retrieval. The mandatory access control is enforced through physical separation of data. The mechanisms required to allow a user at a given session level access to a backend database are straightforward and easy to understand. Hence, if one uses an evaluated product as a trusted front end, very little trusted code needs to be developed to assure that this separation is maintained. Other approaches are susceptible to some kinds of inference attacks arising from tricking the MLS DBS into materializing unauthorized views. The SINTRA DBS is not vulnerable to these attacks.

The SINTRA database system, which is currently being prototyped at the Naval Research Laboratory, uses Honeywell XTS-200 system as a trusted frontend and untrusted ORACLE databases which are running on SUN4/300 as backend databases. The backend and frontend computers are connected through Ethernet.

In the SINTRA project, we make the following assumptions:

(1) All UBDs use the same database query language (e.g., SQL).

(2) The TFE changes the database states of the UBD through a database query language.

(3) Each UBD performs some type of scheduling which produces a serializable history.

Figure 1 illustrates the SINTRA architecture.

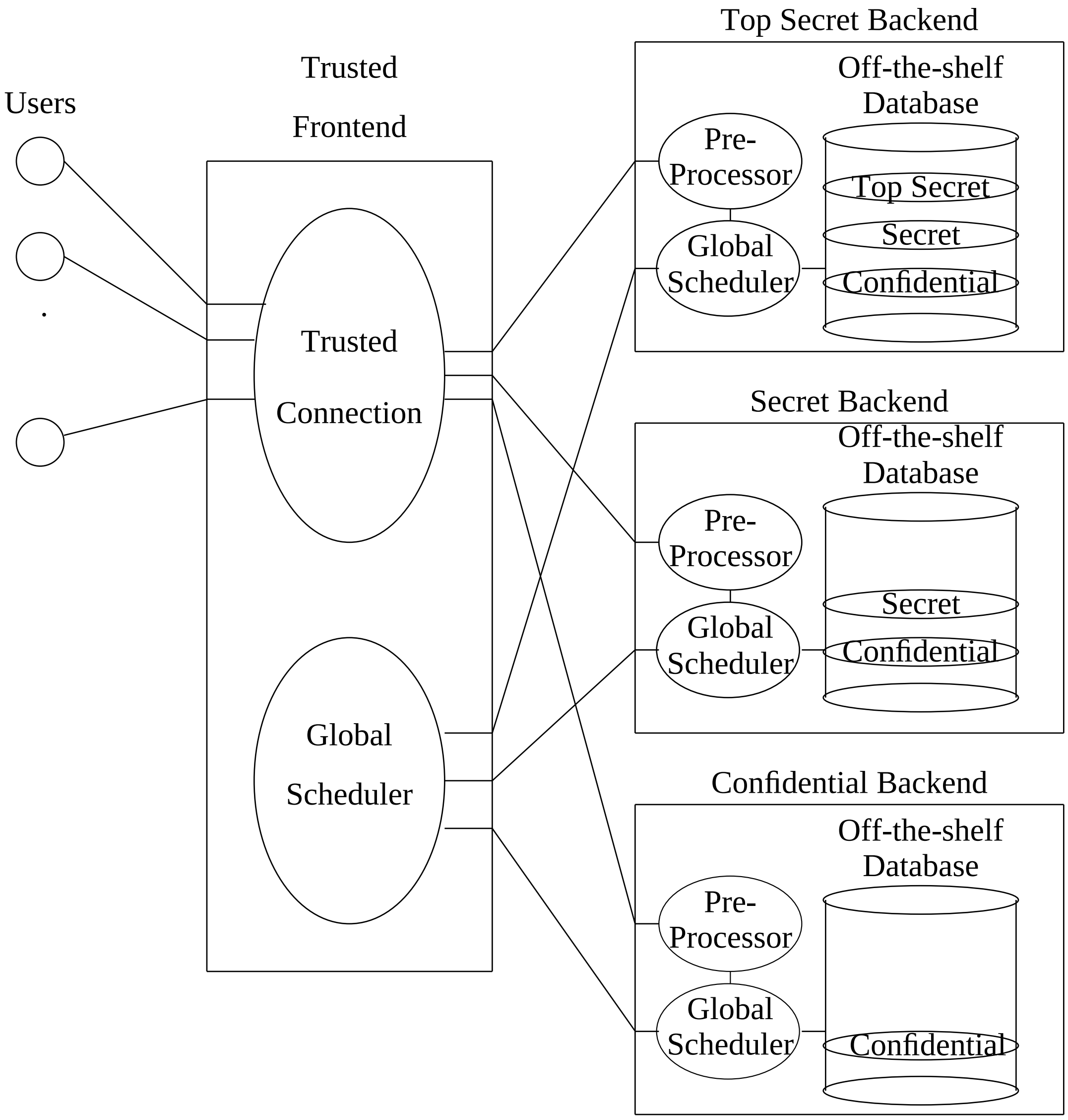


Figure 1: The SINTRA Architecture.

There are two components between the trusted frontend and an off-the-shelf database: (1) global scheduler and (2) query preprocessor. These two components assure the consistency and integrity of replicated data among different backend databases.

Global Scheduler

Since each UBD in a replicated architecture contains data from lower levels, update queries have to be propagated to higher security level databases. If this propagation of update queries, which will be called *update projections* in this paper, is not carefully controlled, inconsistent database states among backend databases can be created. Consider that two confidential level update transactions T_i and T_j are scheduled with serialization order $<T_i, T_j>$ at the confidential level backend database system. Since these two transactions are update transactions, they have to be propagated to the secret level. If these two transactions are scheduled with serialization order $<T_j, T_i>$ at the secret level, an inconsistent database state between confidential and secret level backend databases may be created. The criterion for consistency used here is one-copy serializability. Therefore, the serialization order introduced by the local scheduler at the user's session level must be maintained at the higher level UBDs.

SINTRA's global scheduler guarantees that the serialization order introduced by the local scheduler at the user's session level is maintained at the higher level UBDs. In summary, the global scheduler performs the following tasks:

(a) Receive queries from the preprocessor and the global scheduler of lower security levels and send them to the appropriate backend database.

(b) Guarantee that the serialization order introduced by the local scheduler at the user's session level is maintained at the higher level UBDs.

(c) When a transaction is committed, send an update projection to higher security level backends.

A generalized theory of a global scheduler for the SINTRA database system has been presented in [KFC92].

Query Preprocessor

The SINTRA query preprocessor plays an important role in maintaining data consistency among different backend databases, preserving data integrity, and bridging the semantic gap between conventional and multilevel-secure databases. Some of our security policy is embedded in our query preprocessor which will limit user's query to access certain portion of a relation. The query preprocessor is based on query modification. When a user query is submitted to the query preprocessor, the user's security level and role is passed also. Depending on these data, user queries are modified so that our security and integrity policies are not violated. A detailed description of the SINTRA query preprocessor appears in [Kan92].

2. The Model

In this section, models are presented for security, replicated architecture, and transaction processing. The transaction model, which is presented in this section, can alleviate the difficulties described in section 1.2.

2.1. Security Model

The security model used here is based on that of Bell and LaPadula [BeL76]. The database system consists of a finite set **D** of *objects* (data item) and a set **T** of *subjects* (transactions). There is a lattice **S** of security classes with ordering relation <. A class S_i *dominates* a class S_j if $S_i \geq S_j$. There is a *labeling function* **L** which maps objects and subjects to a security class:

$$\mathbf{L}: \mathbf{D} \cup \mathbf{T} \rightarrow \mathbf{S}$$

Security class **u** *covers* **v** in a lattice if **u** > **v** and there is no security class **w** for which **u** > **w** > **v**.

We consider two mandatory access control requirements:

(Simple Security Property)

If transaction T_i reads data item x then $L(T_i) \geq L(x)$.

(Restricted *-Property)

If transaction T_j writes data item x then $L(T_j) = L(x)$.

The simple security property allows a transaction to read data items if the security level of a transaction dominates the security level of data items. The restricted *-property allows a transaction to write if the security level of a transaction is the same as that of data items (i.e., no write-ups or write-downs are permitted). Write-ups (i.e., T_i writes to data item x and $L(T_i) < L(x)$) are undesirable in database systems for integrity reasons.

2.2. Replicated Architecture Model

The system has a TFE, which mediates the access of subjects to objects. The TFE contains the trusted computing base (TCB), but not all of the TFE need to be trusted. The system also contains a set of single level untrusted backend databases **C**, one for each element of the security lattice. Each backend database $\mathbf{C_u}$ contains copies of all data items in all databases whose security level is dominated by security level **u**. Alternatively, if **L(x)** = **u** such that $\mathbf{x} \in \mathbf{C_u}$, then there is a copy of **x** in each database whose security level dominates **u**.

2.3. Transaction Model

We adopt a layered model of transactions, where a transaction is a sequence of queries, and each query can be considered as a sequence of reads and writes. For example, `replace` and `delete` queries can be viewed as a read operation followed by a write operation, `insert` can be viewed as a write operation, and `retrieve` can be viewed as a read operation. A layered view of two transactions T_1 and T_2 is shown in figure 2.

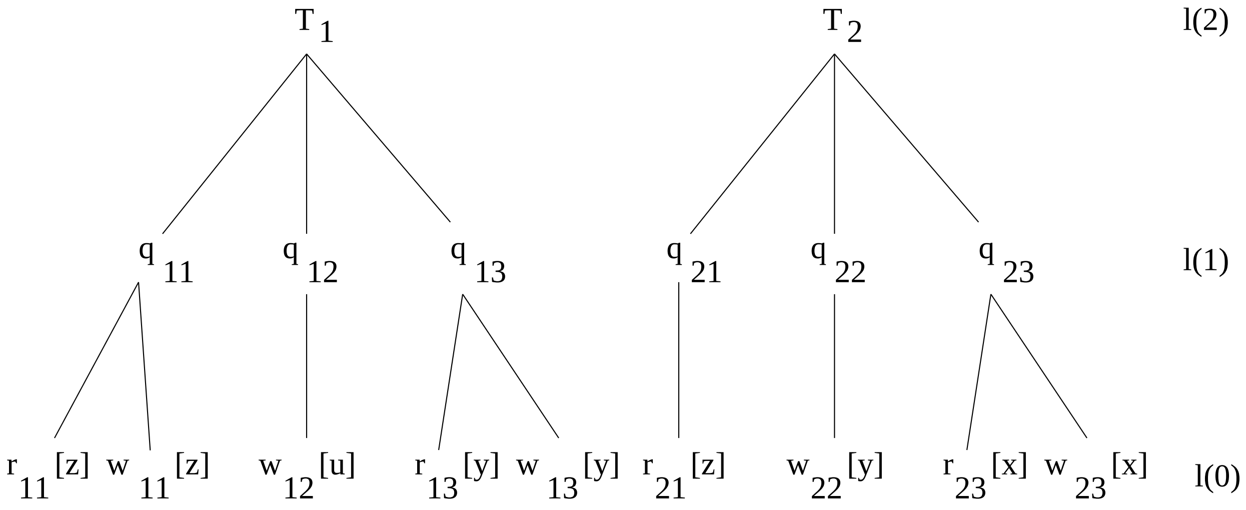


Figure 2: Layered model of two transactions

Definition 1.

A **transaction** T_i is a sequence of queries, i.e., $T_i = <q_{i1}, q_{i2}, ..., q_{in}>$. Each query, q_{ij}, is an atomic operation and is one of `retrieve`, `insert`, `replace`, or `delete`.

To model the propagation of updates produced by a given transaction to higher security level databases, *update projection* is defined.

Definition 2.

An **update projection** U_i, which corresponds to a transaction T_i, is a sequence of update queries, e.g., $U_i = <q_{i2}, q_{i5}, ..., q_{in}>$ obtained from transaction T_i by simply removing all `retrieve` queries.

To describe concurrency control mechanisms, we adopt the following definition of *conflict*.

Definition 3.

Two operations at the same layer **conflict** if and only if there is a possible state in which they do not commute. Alternatively, two operations conflict if they operate on common data and not both are `retrieve` operations.

In the following section, we present a concurrency control algorithm using the transaction model above. In our concurrency control algorithm, the global scheduler works at the query level (i.e., *l(1)* in figure 2).

3. A Concurrency Control Mechanism

In this section, a concurrency control algorithm is presented which makes no assumptions about UBD scheduling. In this algorithm, two types of schedulers can be identified, global and local schedulers. The global scheduler enforces data consistency among different security levels. On the other hand, the local scheduler enforces serializability among transactions, including update projections, which are submitted to the backend database system. The local scheduler deals with layer *l(0)* in figure 2, and the global scheduler deals with layer *l(1)* and upper layers. The global scheduler detects conflicts at level *l(1)*. Therefore, no knowledge of the specific items to be accessed or even the granularity of the lower level concurrency controller is required.

3.1. Algorithm C

To describe the concurrency control protocol, we need to define several mechanisms:

- A queue $\mathbf{Q_u}$ is associated with each backend database $\mathbf{C_u}$, where **u** is a security level. The purpose of $\mathbf{Q_u}$ is to maintain a list of update projections which have been executed and committed at $\mathbf{C_u}$. The queue is ordered by the serialization order of the execution of these transactions at $\mathbf{C_u}$.
- In addition, there is an untrusted mechanism $\mathbf{R_u}$ which maintains $\mathbf{Q_u}$ and can read the contents of $\mathbf{Q_v}$ for all **v** which are dominated by **u** in the security lattice.
- Another queue $\mathbf{A_u}$ is associated with each backend database $\mathbf{C_u}$. The purpose of $\mathbf{A_u}$ is to maintain a list of update projections which come from $\mathbf{Q_v}$, where **v** is covered by **u**, and are waiting to be sent to $\mathbf{C_u}$. The order of update projections in $\mathbf{A_u}$ is determined by the concurrency control algorithm which will be described later.

In our algorithm, $\mathbf{Q_u}$, $\mathbf{A_u}$, and $\mathbf{R_u}$ are considered parts of a global scheduler. Since mechanism $\mathbf{R_u}$ has to read the contents of $\mathbf{Q_v}$ for all **v** which are dominated by **u**, the $\mathbf{R_u}$ and the $\mathbf{Q_u}$ may be located in the TFE. However, $\mathbf{A_u}$ may be located in the backend system (see figure 1). Also in our algorithm, we say a backend database $\mathbf{C_u}$ *covers* $\mathbf{C_v}$ if **u** covers **v** in the security lattice. The protocol processes transactions as follows:

Algorithm C:

At each backend database $\mathbf{C_u}$:

[1] Primary transactions (that are submitted directly by the user) and update projections are received from the global scheduler and submitted to the local backend scheduler.

[2] As local transactions (primary transactions and update projections) are committed, a report of their serialization is sent to the global scheduler. These reports are sent in an order consistent with the serialization order determined by the local scheduler.

At the global scheduler:

[1] For each primary transaction $\mathbf{T_i}$ submitted to the TFE, $\mathbf{T_i}$ is dispatched to $\mathbf{C_u}$ for processing where $\mathbf{L(T_i) = u}$.

[2] Whenever a serialization report for $\mathbf{T_i}$ or $\mathbf{U_j}$ is received from $\mathbf{C_u}$, it is added to the end of $\mathbf{Q_u}$.

[3] The $\mathbf{R_u}$ scans the queue $\mathbf{Q_v}$ for those $\mathbf{v}$ for which $\mathbf{C_u}$ covers $\mathbf{C_v}$. The $\mathbf{R_u}$ will retrieve an update projection $\mathbf{U_i}$ from $\mathbf{Q_v}$ and add it to the end of $\mathbf{A_u}$ when the following condition is satisfied for all $\mathbf{v} \in \mathbf{S}$:

- If $\mathbf{C_u}$ covers $\mathbf{C_v}$, and $\mathbf{U_j}$ can eventually appear in $\mathbf{Q_v}$, then it does appear in the beginning of $\mathbf{Q_v}$.

[4] For update projections in the queue $\mathbf{A_u}$, update projections are sent to $\mathbf{C_u}$ one after another. Specifically, if $\mathbf{U_i}$ is before $\mathbf{U_j}$ in the queue $\mathbf{A_u}$, then send $\mathbf{U_i}$ and wait until $\mathbf{U_i}$ is committed at $\mathbf{C_u}$, and then send $\mathbf{U_j}$.

[5] An update projection is removed from $\mathbf{A_u}$ once it is committed.

[6] If an update projection $\mathbf{U_i}$ is aborted then resubmit $\mathbf{U_i}$ to $\mathbf{C_u}$.

In algorithm C, we assume that local schedulers report the serialization order of transactions to the global scheduler. However, most database systems do not provide their serialization order. Also it may not be easy to modify database systems to report the serialization order. In such cases, the *take-a-ticket* [Geo91] operation can be used to determine serialization order. The *take-a-ticket* operation consists of reading the value of the ticket just prior to commit time, and incrementing it through regular data manipulation operations. The value of a ticket determines the serialization order. All operations on tickets are subject to the local concurrency control.

Note that algorithm C does not slow down user (primary) transactions. The global scheduler of algorithm C concerns the serialization order of the update projections in $\mathbf{A_u}$ at each security level. Concurrency control among primary transactions and update projections is the responsibility of the local scheduler in the UBD.

Also note that $\mathbf{Q_u}$ and $\mathbf{R_u}$ are not needed if the security classes form a completely ordered set, since $\mathbf{A_u}$ satisfies all the requirement of the algorithm. The proof of correctness of the algorithm C is presented in [KFC 92].

4. Object-Oriented Development

The development methodology of the SINTRA global scheduler closely resembles the object-oriented development method [Boo86]. Object-oriented development method enable us to apply software engineering principles such as data hiding, modularization, abstraction, etc. Many objects such as queries, transactions, processes, etc, have been identified, and the relationships among these objects have been established. Many layers are also introduced to hide lower-level details. C++ has been chosen as our main implementation language because the object-oriented programming language provides the capabilities of information hiding and the abstraction of interfaces. Some of our code which resides in the front-end is written in C, because XTS-200 provides neither a C++ interpreter nor the C compiler that can compile C code generated by a C++ interpreter.

We first introduce the conceptual process-level architecture (see figure 3) and the role of each process.

The responsibilities of the processes are as follows:

1. Database Server:

   Look for a user and create a database server child if a user logs in successfully.

1´. Database Server Child:

   Ensure data flows among user, preprocessor, and user transaction scheduler conform to connections in diagram.

2. Preprocessor:

   Modify queries to enforce security/integrity properties (see [Kan92]).

3. User Transaction Scheduler:

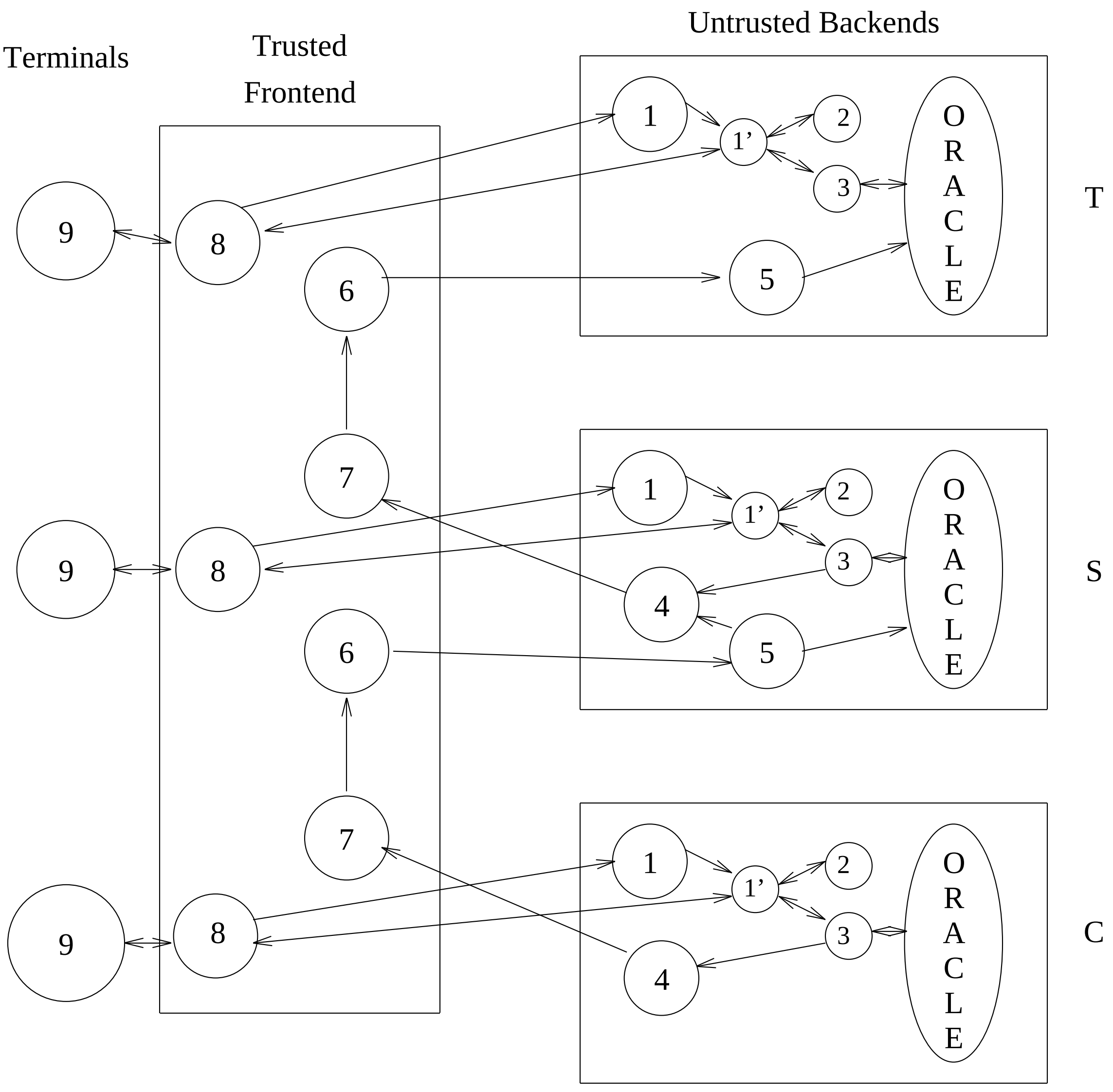
   Submit user transactions to the ORACLE database. Send the response from ORACLE to the user and send update queries to the propagation scheduler.

4. Propagation Scheduler:

   Receive transactions and send them to the corresponding front-end update projection receiver according to the serialization order.

5. Update Projection Scheduler:

   Receive update projections from the lower level backend, submit them to the ORACLE database, and send them to the propagation scheduler.



1: database server	1’: database server child
2: preprocessor	3: user transaction scheduler
4. propagation scheduler	5. update projection scheduler
6: projection sender	7: projection receiver
8: front-end connector	9: terminals

Figure 3: The SINTRA Process-level Architecture.

6. Projection Sender:

Read-down to get the update projections which are in the lower level projection receiver and send the projections to the update projection scheduler.

7. Projection Receiver:

Receive update projections from the propagation scheduler and store them for retrieval by the projection sender.

8. Front-end Connector:

Establish a virtual connection between the user and the backend depending on the user's session level.

The above process-level architecture has been implemented using objects, and inter-process communication can be implemented by message passing between objects. Complex inter-process communication details has been hidden by inter-process communication (ipc) object methods. Figure 4 shows process objects, communication paths, and messages which are passed.

Note that update projection scheduler has been implemented using two separate processes (figure 4). The reason for using two processes is that the transactions which are propagated from the lower level can be received by update projection scheduler (child) even if update projection scheduler (parent) is busy with processing update projections.

The Communication method between two processes in database systems should not only be recoverable in case of system failure but also be able to collect garbage in a reasonable way. There are several communication methods between two processes at different security levels:

Read-down:

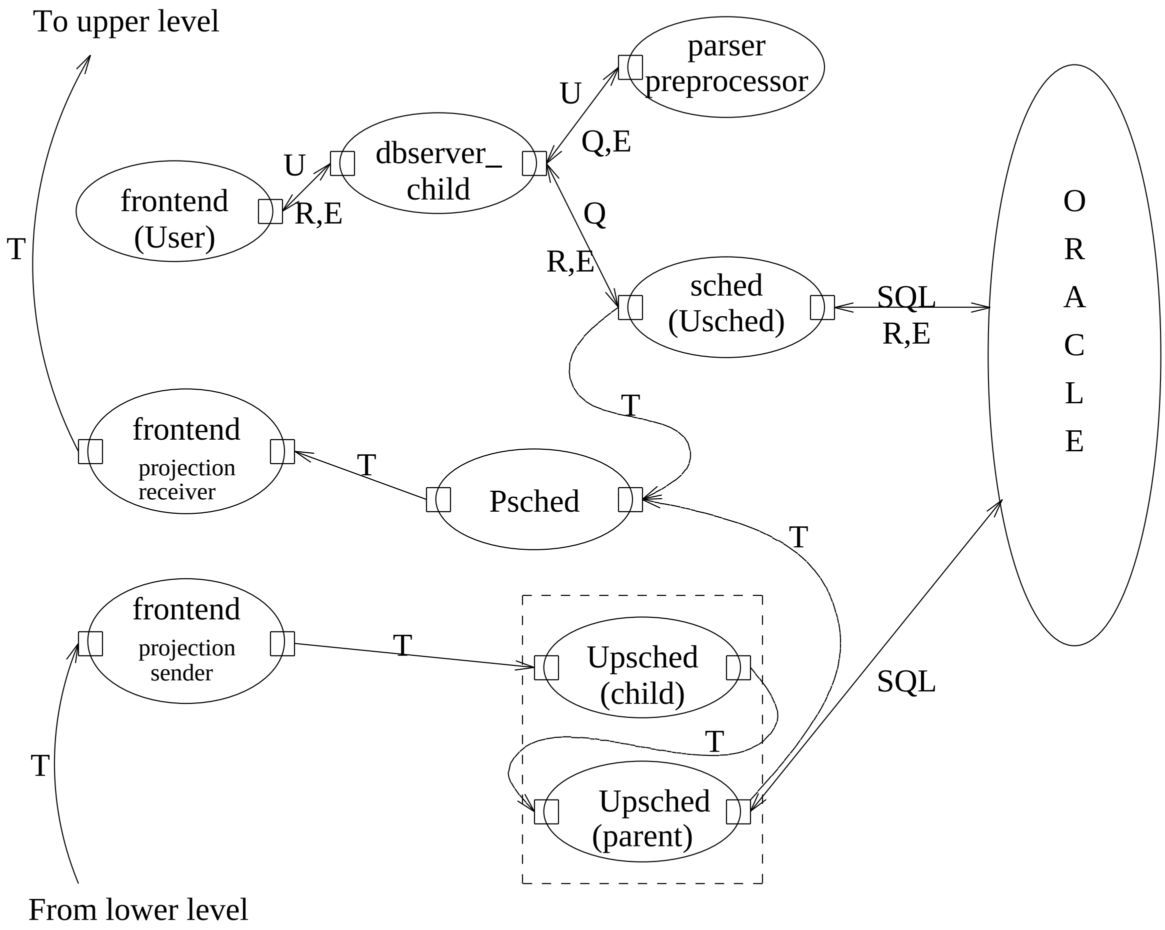
A higher-level process reads-down the message in the lower level process. This method is recoverable but has a difficulty to collect garbage. Since the lower level process does not know when the higher level process has read the message, the lower level process has a difficulty to determine when to remove the message from its message queue.

Blind Write-up:

A lower-level process sends messages to a higher level process. The lower level process then assumes that the higher level process receives the message and removes the message from its message queue. This method either has the same problem as the *read-down* or is not recoverable because the lower level process never be sure if the higher level process actually receives the message or not.

Write-up with Acknowledgement:

This method sends an acknowledgement when the higher level process receives a message from the lower level process. This method is recoverable and no difficulty to collect the garbage. However, since the higher level process sends an acknowledgement, this can be used as a covert timing channel by Trojan horses. Even though there are few ways to reduce the bandwidth of this timing channel, the



U: user command R: reply
T: transaction Q: query
E: error

Figure 4: The SINTRA Process Objects and Communication Paths.

trade-off between the bandwidth of this channel vs. the performance penalty is yet to be investigated.

Currently, the communication between the projection receiver and the projection sender is accomplished through a read-down method (the projection sender reads-down transactions in the lower level projection receiver).

Figures 5.1 and 5.2 show the composition of SINTRA process objects along with the inheritance relationship between major classes that compose the system.

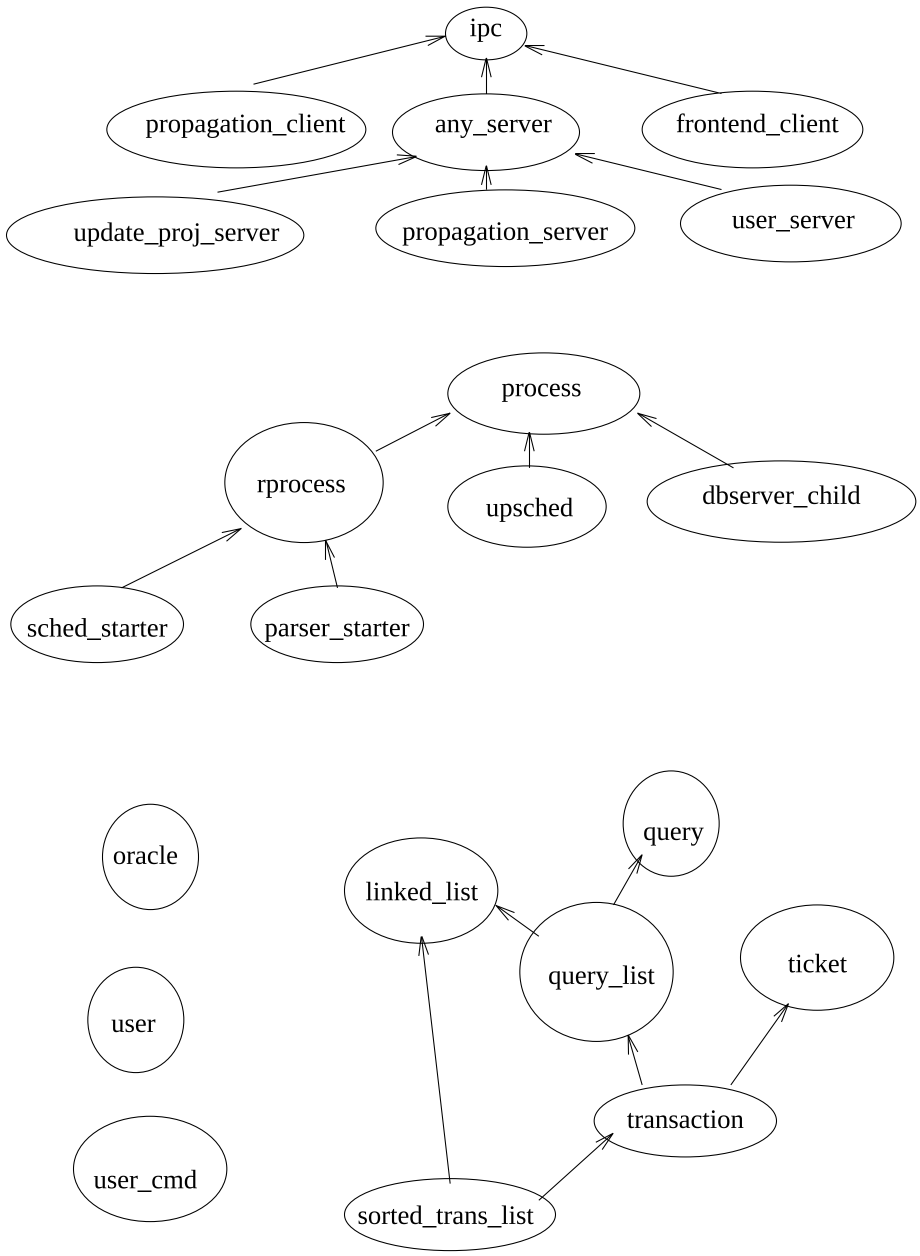


Figure 5.1: Class Hierarchy

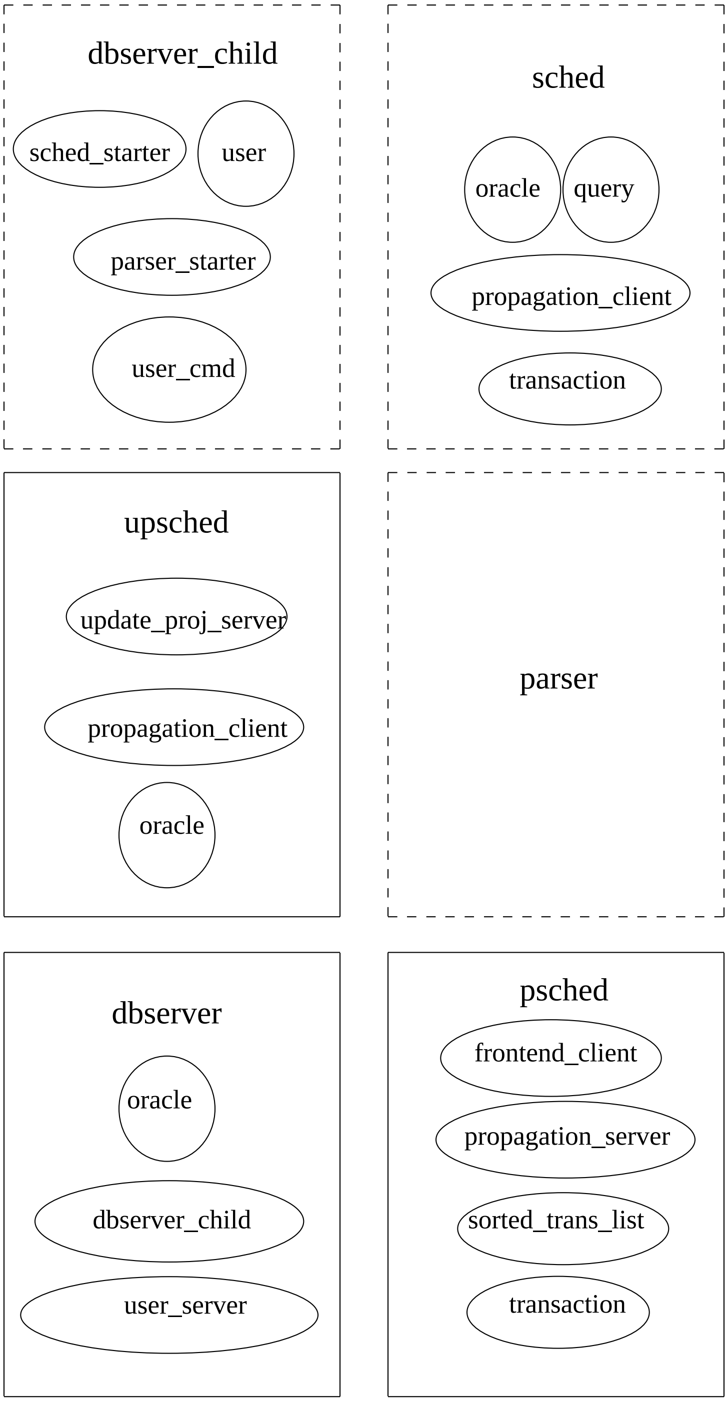


Figure 5.2: Composition of Process Objects

Dbserver (database server), psched (propagation scheduler) and upsched (update projection scheduler) are static process objects that are created when SINTRA is started. One copy of the process objects dbserver_child (database server child), parser (query preprocessor) and sched (user transaction scheduler) is created per user login and destroyed when the user logs out. The responsibilities of the process objects have been described in section 4. A description of the classes pictured in figure 5.1 that compose the process objects follows:

Ipc:
: This class provides methods which can be used to communicate between process objects. It is implemented on top of Unix sockets.

Propagation_client:
: In addition to the functionality inherited from ipc, it can be used to connect to the propagation scheduler.

Any_server:
: This class advertises a communication end-point for other processes to connect to.

Frontend_client:
: In addition to the functionality inherited from ipc, it can be used to connect to the projection receiver which is executing on the frontend.

Update_proj_server:
: In addition to the functionality inherited from any_server, it can be used to register the user update projection scheduler and provides a send and receive port to process objects wanting to send and receive update projections.

Propagation_server:
: In addition to the functionality inherited from any_server, it can be used to register the propagation scheduler within the system and provides a send and receive port that can be used to send and receive update projections.

User_server:
: In addition to the functionality inherited from any_server, it can be used to register the dbserver within the system and provides a send port that can be used to send text to the user (via the frontend connector) and a receive port that can be used to receive commands from the user.

Process:
: It can be use to create a copy of the current executing process.

Rprocess:
: In addition to the functionality inherited from process, it can be used to redirect the standard input, output and error streams of the process that it is invoked by.

Sched_starter:
: In addition to the functionality inherited from rprocess, it can be used to start the

user transaction scheduler in a manner that causes the user transactions scheduler’s i/o streams to be connected to the dbserver_child.

Parser_starter:

In addition to the functionality inherited from rprocess, it can be used to start the preprocessor in a manner that causes the preprocessor’s i/o streams to be connected to the dbserver_child.

Oracle:

It can be used to access the ORACLE DBMS.

User:

It contains information about the currently logged on user.

5. Conclusion

We presented the detailed description of the SINTRA global scheduler. Even though large portions of the SINTRA global scheduler do not have to be trusted, we have tried to follow good software engineering principles in designing and structuring the code.

We have prepared this report for system designers and programmers who want to understand the structure of the SINTRA global scheduler. We also hope this report is also helpful to the people who will maintain the SINTRA global scheduler code.

References

[BeL76]

Bell, D. E., and LaPadula, L. J. Secure computer systems: Unified exposition and multics interpretation. The Mitre Corp, (1976).

[Boo86]

Booch, G. Object-oriented development. IEEE Transactions on Software Engineering, 12, 2 (1986).

[Geo91]

Georgakopoulos, D., et el. On serializability of multidatabase transactions through forced local conflicts. Proceedings of Conference on Data Engineering (1991).

[Kan92]

Kang, M. H., Costich, O., Froscher, J. The replicated architecture data model: Structure and operation. Internal document (1992).

[KFC92]

Kang, M. H., Froscher, J. N., and Costich, O. A practical transaction model and untrusted transaction manager for multilevel-secure database systems. The Eighth IFIP Workshop on Database Security (1992).

[Lun90]

Lunt, T., et el. The SeaView security model. IEEE Transaction on Software Engineering, 16, 6 (1990).

Appendix: Description of Top-level Routines

The followings are the description of top-level routines of process objects written in C++ looking pseudo-code:

```
/* preprocessor class */

 while(1) {
   user_cmd.receive(dbserver_child.out());
   process();
   query.send(preprocessor.out());
 }


/* dbserv class */

 initialize_ticket();

 USER_SERVER user;
 while(1) {
   while(try_connect());
   // user is connected, create necessary child processes
   DBSERV_CHILD db_per_user(user); // create dbserver child instance
 }
```

```
/* dbserv_child class */

  PREPROCESSOR_STARTER preprocessor();// create preprocessor instance
  SCHED_STARTER sched();          // create user transaction scheduler
  // establish connection from/to the user, preprocessor, and user scheduler
  establish_connection();
  // receive messages
  do {
    if(ready(user.out())) {
      user_cmd.receive(user.out());
      user_cmd.send(preprocessor.in());
    }

    if(ready(preprocessor.out())) {
      query.receive(preprocessor.out());
      query.send(sched.in());
    }
    else if(ready(preprocessor.err())) {
      error.receive(preprocessor.err());
      error.send(front_end.in());
    }

    if(ready(sched.out())) {
      reply.receive(sched.out());
      reply.send(user.in());
    }
  } while (connection_established);
```

```
/* user scheduler class */

 // establish connection to propagation scheduler and ORACLE
 connect_to_psched();
 connect_to_oracle();
 while(1) {
   query.receive(dbserver_child.out());
   if(query.is_commit() || query.is_exit()) {
     trans.get_ticket_and_commit();
     if(level != TOP_SECRET)
       trans.send(psched.in());
   }
   else {
     query.send(oracle.in());        // execute the query
     reply.receive(oracle.out());
     reply.send(sched.out()); // send reply
     if(query.is_abort())
       trans.clear();      // clear the queries in transaction
     else if(query.is_update())
       trans.append(query);
  }
  if(query.is_exit())
     break;
 }
 // if user wants to exit then disconnect from propagation scheduler and ORACLE
 disconnect();
```

```
/* update projection scheduler */

// create a queue to be shared between the receiver and sender below
create_trans_queue();

create_child_process();       // child process will receive projections
                              // from the frontend

if (parent) {
  // establish connection to the propagation scheduler and ORACLE
  connect_to_psched();
  connect_to_oracle();

  while(1) {
    trans.get_trans(queue);        // get one update projection from queue
    trans.send(oracle.in());       // send transaction to ORACLE
    if(level != TOP_SECRET)
        trans.send(psched.in());
  }
} else {                          // child
  // establish connection to the frontend
  connect_to_front();

  while(1) {
    trans.get_trans(frontend.out());       // get one update projection
                                           // from the frontend

    trans.send(queue);                     // enqueue the transaction
  }
}
```

```
/* propagation scheduler */

connect_to_front();            // connect to the front-end
let_client_connect();          // allow clients' connections
INT expected = FIRST_TICKET;   // initialize expected ticket number
while(1) {
  trans.get_trans();        // get one update projection
  if(trans.get_ticket() == expected) {
    trans.send(frontend.in());
    expected++;
  }
  else                      // if transaction's ticket is not in order
    sorted_list.insert(trans);   // then put it in a queue

  while(sorted_list.first() && sorted_list.first().get_ticket() == expected)  {
    trans = sorted_list.get_first();
    trans.send(frontend.in());
    expected++;
  }
}
```