SUPPORTING
ACTIVE DATABASE SEMANTICS
IN SYBASE

By

DAVID VANCE

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

1996

Copyright 1996

by

David Vance

TABLE OF CONTENTS

page

ACKNOWLEDGMENTS v

ABSTRACT vi

CHAPTERS

1 INTRODUCTION 1

2 POLLING 4

3 POLLING WITH TRIGGERS AND EVENT TABLES 10

4 ASYNCHRONOUS TRIGGER TO OPEN SERVER RPC 23

5 REPLICATION SERVER TO OPEN SERVER 34

6 SYNCHRONOUS TRIGGER TO OPEN SERVER RPC 38

7 OPEN SERVER GATEWAY 46

8 RULE EVALUATION/EXECUTION IN THE GATEWAY ARCHITECTURE 52

9 GATEWAY DESIGN AND IMPLEMENTATION 56

10 CONCLUSION 61

REFERENCES 62

BIOGRAPHICAL SKETCH 63

ACKNOWLEDGEMENTS

Firstly, thanks are due to my advisor, Dr. Sharma Chakravarthy, for his continual support, advise, and guidance in this research. I am thankful also to Dr. Eric Hanson for serving on my committee and especially for teaching an excellent class on Database Implementation, which has greatly helped me in the preparation of my thesis as well as the advancement of my understanding of databases. Dr. Doug Dankel deserves special thanks, not only for serving on my committee, but also for approving my admission to the graduate program against his better judgment.

I would like to take this opportunity to thank my Mother and Step-Father for their help, support, and love throughout the year, as well as for the only nourishing meals I had between September and May.

Finally, and supremely, I would sincerely thank the God whose I am and whom I serve for His love, for His special acts of providence, and for His sustaining hand upon me in every area of my life. I could have accomplished absolutely nothing apart from His will or without His mercy and grace which were purchased for me by the Lord Jesus Christ. Thank You for the many lessons in humility and for the kindness You have shown to Your very undeserving servant. May honor be given to Whom honor is due. If there is anything of worth, or insight, or help in this thesis,

SOLI DEO GLORIA (To God Alone be the Glory).

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

SUPPORTING
ACTIVE DATABASE SEMANTICS
IN SYBASE

By

DAVID VANCE

August 1996

Chairman: Dr. Sharma Chakravarthy
Major Department: Computer and Information Science and Engineering

It is possible to add an active database component to Sybase, a commercial DBMS, to support the full range of active semantics. There are two primary advantages to this approach. First, existing applications and “productivity tools,” such as Microsoft Access, can continue to access the database without modification. This means that active database capability may be inserted into an existing system without changing the client application programs. Second, all of Sybase’s underlying functionality is retained.

Active database semantics can be supported on an existing Sybase SQL Server by building components based on Sybase’s connectivity products. Nearly the full range of active database functionality can be supported on top of an existing database without requiring changes to applications, although some architectures require very minor additions to the database. Both immediate and detached semantics can be implemented. Several different architectures for

supporting immediate and detached semantics are described in detail, and their features and limitations are identified.

In addition to describing several alternative methods for capturing database events, this thesis develops the process of integrating production rule evaluation and execution into an existing system, and describes the design and implementation of an actual active database component.

CHAPTER 1
INTRODUCTION

Traditional database systems are said to be passive, meaning that they only perform the transactions and queries which are explicitly submitted by a user or an application program. Active databases, however, can monitor the state of the system for particular events and trigger appropriate and timely responses when those events arise. The definitions of the desired behavior are specified in production rules, also known as event-condition-action (ECA) rules, which are stored inside the database [DAY94].

While a number of research prototypes of active database systems have been built [HAN92] [CHA89] [DAY88] [STO90] [STO91] [WID91] [ANW93], production rule capability in commercial systems is very limited [ING92] [ORA92] [SYB96]. The research prototypes listed are representative of a much larger number of active database systems which have been designed using an integrated approach. In other words, the production rule components of active database systems have typically been integrated directly into the kernel of the DBMS's.

Would it be possible to separate the active part from the database part? If so, it might be possible to turn any traditional database management system into a true active database. What would the limitations be of such an architecture? This thesis considers the possibility of adding an active component into the Sybase SQL Server DBMS.

Advantages Of Decoupling The Active Component From The DBMS

Although there are some advantages to integrating the active component into the DBMS engine, there are many reasons why the active component should be decoupled from the DBMS:

- Existing applications and software would not have to be changed to call a new DBMS's API.
- Existing databases would not have to be converted or up/down-graded.
- Layered architectures are generally preferred because of their modularity and simplicity of design.
- Separation produces a more scaleable architecture.
- Once a separate active component has been developed, it may be ported to access other DBMS's.
- None of an existing DBMS's functionality would be lost.
- A more distributed architecture can be designed if the active component is separated.
- There is nothing about active database semantics which demands that the rule execution must be integrated into the DBMS.

The Reason For Choosing Sybase

Sybase has many features which facilitate the implementation of an active component on top of an existing database. Sybase has an open architecture and offers many connectivity tools which other DBMS vendors do not. The following features are particularly useful in implementing the active database architectures:

- triggers,
- server-side cursors,
- Sybase RPC's,
- OpenServers, and
- Replication Servers.

These features and their usefulness are described at appropriate places in the paper. It should be noted here that not every architecture requires all of these features; and many of the architectures described in this paper can be implemented on top of any DBMS.

Another advantage of Sybase is that their OpenClient database API, also known as DB-Library, is widely accepted in the industry, and many productivity tools and applications support that protocol. In addition, it is also the standard API supported by Microsoft and some TP-Monitors.

Seamless Integration

One important goal in designing an active component is to ensure that any existing Sybase or Microsoft SQL Server application can continue to function without modification. Therefore, all of the architectures considered in this thesis will not require any changes to the database clients. The clients are not aware that any changes have taken place.

CHAPTER 2
POLLING

Overview Of Detached Architectures

Polling only provides what is known as detached semantics. In detached semantics, the production rule engine, which is called an ECA Server in this thesis, can not affect the outcome of a transaction. Events are generated after a transaction has committed, but the rules may perform further inquiries on the database.

Description

Perhaps the simplest way of determining when an event has occurred is to check or poll for changes on a regular basis. A polling application is a client of the database where the data of interest resides. At a specified time interval, the application examines the current state of the system to determine if any events need to be triggered. The other client application programs do not need to be modified to implement the polling architecture.

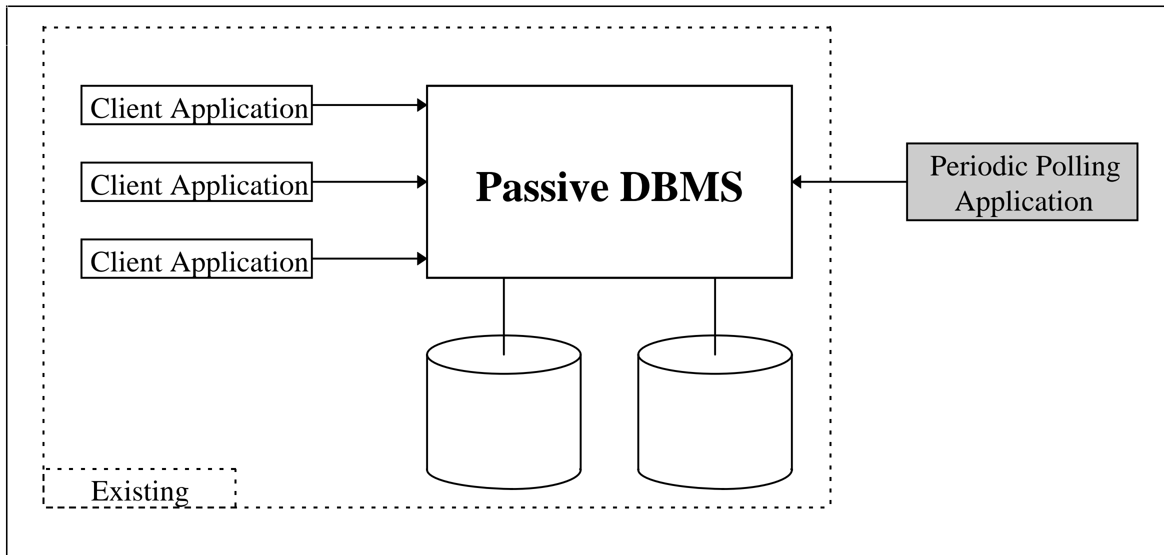


Figure 2-1: The Polling Architecture

For example, Figure 2-2 shows a sample polling program which will check the Thermometer table every ten seconds to determine if any of the thermometers' temperatures exceed 100 degrees:

```
while (1 = 1)
begin
  select  temperatureId
  from    Thermometer
  where   temperature > 100

  waitfor delay "00:00:10"
end
```

Figure 2-2: A Simple Polling Application

Polling is an easy solution if events can only be described as predicates on the current state of the system. The problem becomes more complicated if events can also be defined as changes to

the state of the system. For example, we might be interested to know if a thermometer's temperature has risen 10% since the last time we checked. We also might want to know if a new thermometer is added to the system.

This problem can be solved by comparing the current state of the system with the previous state. A copy of the data can be kept locally on the client or on the database. Figure 2-3 shows a polling application which checks for the two events mentioned above.

```
while (1 = 1)
begin
  select  t.thermometerId,
          "Temperature increased by at least 10%." event
  from    Thermometer t, LastThermometer l
  where   t.temperature > l.temperature * 1.1 and
          t.thermometerId = l.thermometerId

  select  thermometerId,
          "New Thermometer Added." event
  from    Thermometer t
  where   thermometerId not in
            (select  thermometerId
             from    LastThermometer)

  delete from LastThermometer

  insert into LastThermometer
    select * from Thermometer

  waitfor delay "00:00:10"
end
```

Figure 2-3: Another Simple Polling Application

Features

Polling offers the following attractive features:

1. Polling can be added without changing the existing system.

The application code does not have to be aware of the polling, and the polling piece may be changed or removed without affecting the other parts of the system.

2. The architecture is simple and widely-used.

Polling is often used to report changes to the current state of a system. The architecture is very simple and easy to implement. The tradeoffs are well understood.

Limitations

There are many limitations to polling which limit its practical usefulness:

1. Some events may be missed.

There is no guarantee that all of the events in a system will be captured by a poll. If the polling interval is too long, or the size of the database is large, two or more changes to the data can occur between polls.

2. The order of events can not be determined.

If the order of events is important to the ECA system, this architecture is not suitable. It is not possible to even determine the order of the transactions.

3. The architecture is not scaleable.

While a poll may perform satisfactorily with a small amount of data, the performance may be unacceptable with a large amount of data. Every time the application polls, Sybase will move

the pages accessed to the front of the LRU/MRU page chain. If the data size is large, this has the effect of flushing out the data which should really be kept in the page cache. Also, if the size of the data being polled is larger than the page cache, performance will quickly degrade.

If events can be defined as changes to the state of the system, the poller must maintain a copy of the previous state. If the data quantity is small, this is not a problem. However, for a large quantity of data, it is impractical to either copy the data to other tables within the database or to retrieve all of the data every time.

4. The response time window may be missed.

If the polling interval is too long, or if locking delays a poll from completing within a certain period of time, the ECA server will not be able to respond to changes within the required time frame. If a long transaction is running on the system, it will force the polling application to wait indefinitely. This may be unacceptable.

5. Client applications may be delayed waiting for the poll to complete.

Ironically, the very modification events which the poll is interested in capturing may be delayed by the poll. The updates will be forced to wait until the poll releases its read locks on the table. This problem obviously becomes more severe as the interval between polls decreases.

Conclusion

Polling may be an appropriate solution when the problem does not require every event to be captured and the size of the data is small. Despite the limitations of this architecture, it is

probably used in practice more frequently than any other method to check for events and record changes to the state of a system.

CHAPTER 3
POLLING WITH TRIGGERS AND EVENT TABLES

Description

The previous architecture suffered from scaleability problems. The entire data set had to be read every time to determine whether the current state requires events to be generated. If we already checked the state of the system on the previous poll, it would be preferable to only examine the data that changed since the last time.

There are a few ways that changes can be detected by the poller. One way is to compare the entire set of data every time with the previous set. This is obviously very expensive for a large amount of data, but it does not require any modification to the existing system. Another possibility is to add a column to the tables which the client applications would update whenever a modification was made. This reduces the amount of data which has to be retrieved, but it requires modification to existing applications. Additionally, the poller must still keep another copy of the data for comparison.

An alternative solution in Sybase is to record changes to the data using triggers. “Triggers” in Sybase are stored procedures, written in Transact-SQL, which can be set to run automatically when an insert, update, or a delete on a table takes place. They are executed by the SQL Server on the SQL Server.

```
create trigger RecordTenPercentIncreaseEvent
on Thermometer
for update
as
    select i.temperature newTemperature,
           d.temperature oldTemperature
    from   inserted i, deleted d
    where  i.thermometerId = d.thermometerId and
           i.temperature >= d.temperature * 1.10
```

Figure 3-1: A Sample Trigger

While triggers are very flexible, there are some limitations to their usefulness. Triggers must be written in Transact-SQL. T-SQL has many limitations which make it impractical for performing ECA rule evaluation:

- There are no complex data types.
- Only atomic values (and not tables) may be passed as parameters to stored procedures.
- There is no direct access C, other programs, or the underlying operating system.
- The speed of execution is very slow relative to other languages.
- Stored procedures and triggers may only be nested to a depth of 16.
- The SQL Server is usually busy serving data anyway.

Although T-SQL is not suitable for performing ECA rule evaluation, it is possible to use triggers to record changes to the database state. Changes can be stored in a separate table and can then be retrieved by the polling mechanism. First, we must create tables to contain the data necessary to describe the event occurrences:

```
create table Thermometer
  (
  thermometerId       int         NOT NULL,
  temperature         float       NOT NULL
  )

create unique clustered index thermometerId
  on Thermometer (thermometerId)
```

Figure 3-2: An Existing Table

```
create table NewThermometerEvent
  (
  thermometerId       int         NOT NULL,
  temperature         float       NOT NULL,

  eventTime           datetime    NOT NULL
  )

create table TenPercentIncreaseEvent
  (
  thermometerId       int         NOT NULL,
  temperature         float       NOT NULL,

  eventTime           datetime    NOT NULL
  )
```

Figure 3-3: Additional Event Tables

A clustered index should be created on each event table to prevent a bottleneck. If the index is not created, all new rows will be added to the end of the last page. Note that the index can not be unique, since there is nothing unique about the rows. As a result, the index should be created with the ALLOW_DUP_ROW option.

```
create clustered index thermometerId
  on NewThermometerEvent(thermometerId)
  with ALLOW_DUP_ROW

create clustered index thermometerId
  on TenPercentIncrease(thermometerId)
  with ALLOW_DUP_ROW
```

Figure 3-4: Event Table Indexes

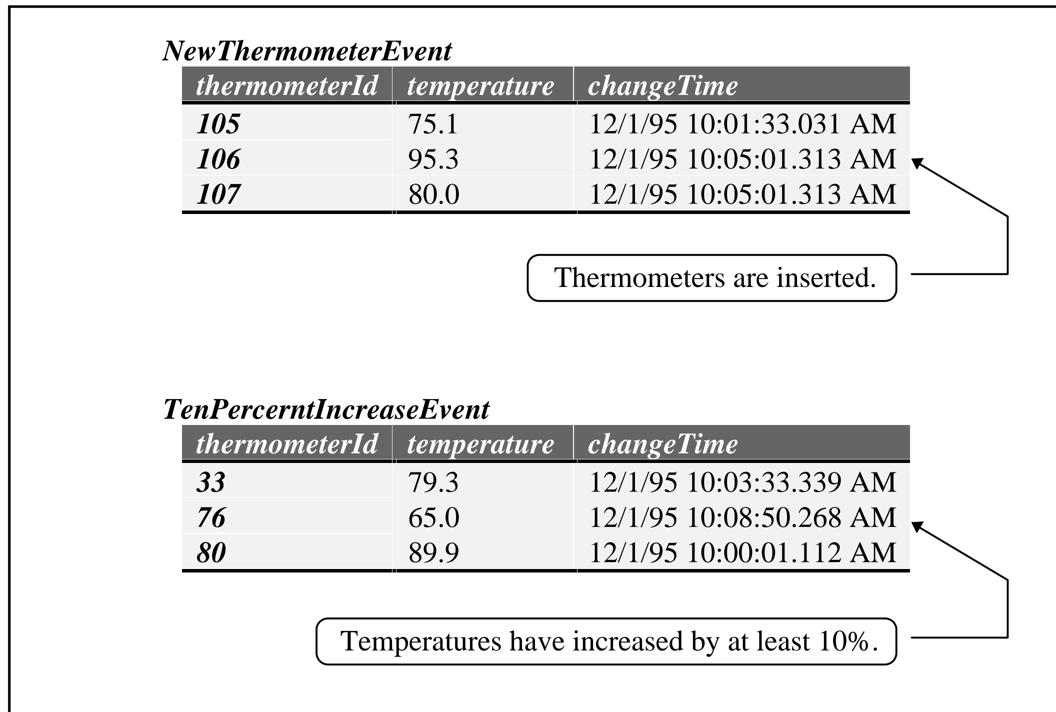
If this implementation is not acceptable, an identity column can be added to the table to artificially impose row uniqueness.

Now a trigger must be added to insert changes into the event table when a modification occurs.

```
create trigger RecordNewThermometerEvent
on Thermometer
for insert
as

  insert into NewThermometerEvent
    select *, getdate ()
    from   inserted


create trigger RecordTenPercentIncreaseEvent
on Thermometer
for update
as

  insert into TenPercentIncreaseEvent
    select i.*, getdate ()
    from   inserted i, deleted d
    where  i.thermometerId = d.thermometerId and
           i.temperature >= d.temperature * 1.10
```

Figure 3-5: Create Triggers

The trigger will add rows to the event tables whenever the original table is modified.



NewThermometerEvent

thermometerId	temperature	changeTime
105	75.1	12/1/95 10:01:33.031 AM
106	95.3	12/1/95 10:05:01.313 AM
107	80.0	12/1/95 10:05:01.313 AM

TenPercerntIncreaseEvent

thermometerId	temperature	changeTime
33	79.3	12/1/95 10:03:33.339 AM
76	65.0	12/1/95 10:08:50.268 AM
80	89.9	12/1/95 10:00:01.112 AM

Figure 3-6: Event Tables After Events Have Occurred

Now the polling mechanism can retrieve the events without copying or querying the original data.

```
while (1 = 1)
begin
  begin transaction

    select  *,
            "New Thermometer Added." event
    from    NewThermometerEvent
    order by eventTime
    holdlock

    delete from NewThermometerEvent

  commit transaction


  begin transaction

    select  *,
            "Temperature increased by at least 10%." event
    from    TenPercentIncreaseEvent
    order by eventTime
    holdlock

    delete from TenPercentIncreaseEvent

  commit transaction

  waitfor delay "00:00:10"
end
```

Figure 3-7: Example Polling Application

Notice that the “holdlock” keyword must be used to ensure that new data will not be entered between the time when the events are read and the time when they are deleted. Since holdlock is prone to cause deadlocks, it is possible to rewrite the transaction to first acquire an exclusive lock on the table by doing a false update. By first performing a global update by executing “update NewThermometerEvent set temperature = temperature where 0=1”, the entire table will be locked and the chance of deadlocks will be reduced.

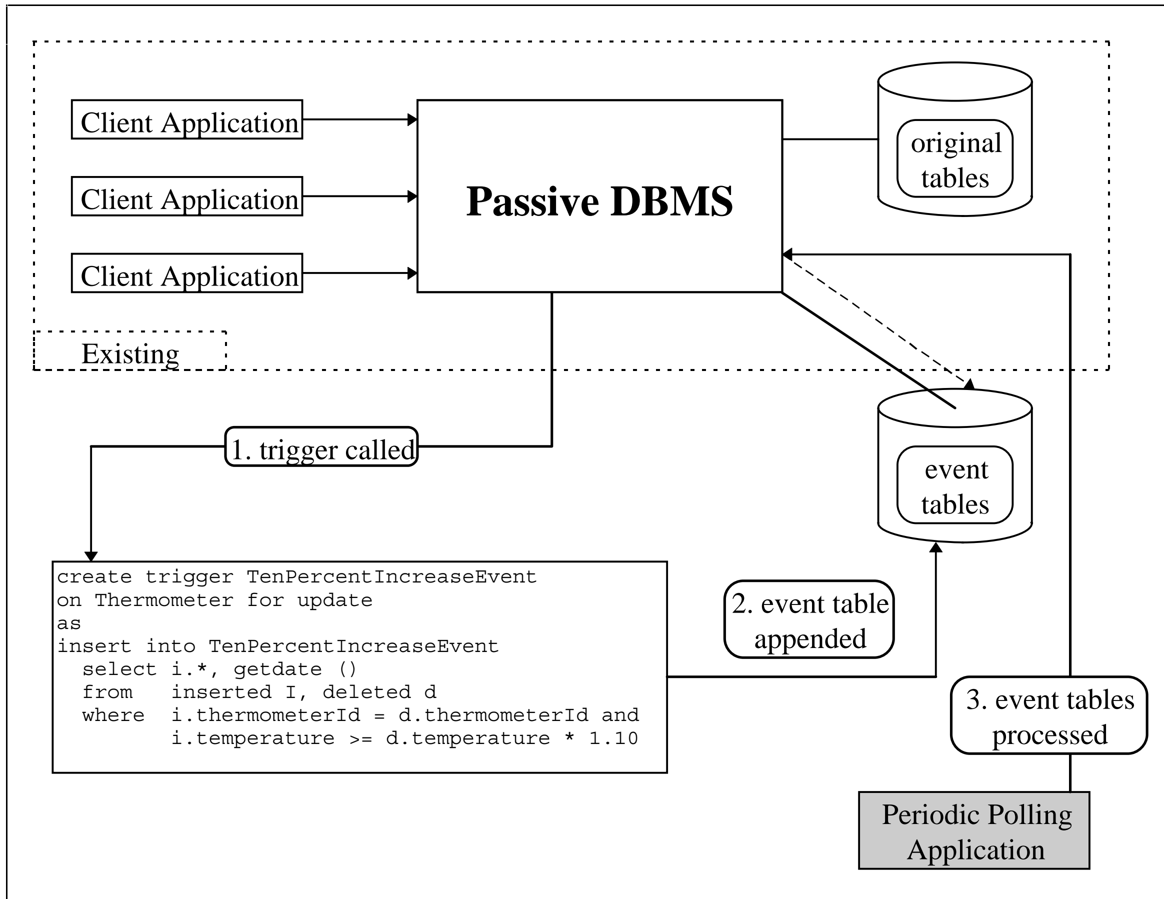


Figure 3-8: The “Polling With Triggers And Event Tables” Architecture

Generating events with Sybase’s immediate triggers may record intermediate modifications as events which are not present at the completion of the transaction. For example, consider the following client application which records temperatures for different kinds of thermometers:

```
#!/bin/sh

##   The getNextNewTemperature checks thermometers for new
##   temperatures.

/usr/local/bin/getNextNewTemperature |
read thermometerId temperature

isql -Uuser -Ppassword << !

use Monitor
go

begin transaction

update  Thermometer
set     temperature = $temperature
where   thermometerId = $thermometerId


/*
**  Compensate for Kelvin thermometers
*/

if exists (select   *
           from     KelvinThermometers
           where    thermometerId = $thermometerId)
begin
  update  Thermometer
  set     temperature = temperature - 273.16
  where   thermometerId  = $thermometerId
end

commit transaction
go
!
```

Figure 3-9: The Intermediate State Problem

In this case, two updates can occur before the system has reached the final state. Using this mechanism, the Kelvin thermometers will always generate the +10% temperature event. While it is possible for the triggers to compensate for this, there may be other inconsistent states yet waiting to occur.

It is a poor architectural solution to have the triggers compensate for intermediate results and temporary inconsistencies which are generated by applications. The problem is inherent in the nature of the immediate trigger semantics. Ideally, Sybase would provide deferred triggers which would fire before the end of a transaction. Then the triggers would not have to be concerned with determining whether changes to system states are real or transient. (Incidentally, I have personally made a feature request to Sybase to allow deferred integrity checks. They did not seem eager to incorporate this feature.)

The best solution to this problem would be to have the polling application check for offsetting results within a transaction. Unfortunately, there is no way in T-SQL to determine the current transaction ID#. Therefore, it is not possible to look at an event table and tell which events are in the same transaction. Otherwise, deferred events could be recorded in a separate event table.

```
create trigger RecordTenPercentIncreaseEvent
on Thermometer
for update
as

  insert into TenPercentIncreaseEvent
    select i.*, getdate ()
    from   inserted i, deleted d
    where  i.thermometerId = d.thermometerId and
           i.temperature >= d.temperature * 1.10

  /*
  **  Offset previous results in this transaction
  */

  update   DeferredTenPercentIncrEvent
  set      temperature = i.temperature,
           eventTime   = getdate ()
  from     inserted i, DeferredTenPercentIncrEvent e
  where    i.thermometerId = e.thermometerId and
           i.temperature  >= 1.10 * ????????? and
           e.transactionId = ???????

 /*
 **  Create a new deferred event if the temperature
 **  has not been modified during this transaction.
 */

 insert into DeferredTenPercentIncrEvent
    select i.*, getdate ()
    from   inserted i, deleted d
    where  i.thermometerId = d.thermometerId and
           i.temperature >= d.temperature * 1.10 and
            i.thermometerId not in
                (select  thermometerId
                 from    DeferredTenPercentIncrEvent
                where   transactionId = ????????)
```

Figure 3-10: The Reason That Net Effects Of Transactions Cannot Be Determined

Features

1. The method for generating events is sound and complete.

The insertions made by trigger will be rolled back if the original transaction is rolled back. Therefore, this method will not miss any modifications will not produce any incorrect event records. Since the dataserver itself will be making the insertions, there is no possibility that a client would fail to record a change.

2. Existing applications remain unchanged, only minor database changes are necessary.

This method requires only the addition of triggers to the database and does not require changes to the existing applications.

3. Enhanced performance.

Performance is substantially better than the previous architecture, because only the changes to the system are examined.

4. Robustness.

If the poller fails and a period of time passes before it can be restarted, no events are lost.

Limitations

1. The events may actually be intermediate changes which do not reflect a consistent database state.

Generating events with Sybase's immediate triggers may record intermediate modifications as events which are not present at the completion of the transaction. It is not possible to determine

the net result of offsetting modifications in a trigger. If deferred semantics are required, this architecture will not be suitable.

2. There is no way to determine the order of events.

Although the modification time was recorded, the time which the transaction actually completed is unknown. If the ordering of events is important to the ECA system, this architecture is not suitable. Deferred triggers would make the times more accurate, but there is still no guarantee that the time in the event tables reflects the actual order in which the transactions committed.

3. The architecture is not scaleable.

Since there would theoretically be only a few event entries per poll, every client and the poller must contend to access a very few pages. It should be noted, however, that the scaleability is much better than the previous solution.

4. The response time window may be missed.

This problem is intrinsic to polling applications. If the polling interval is too long, or if locking delays a poll, the ECA server will not be able to respond to changes within the required time frame. If a long transaction is running on the system, it will force the polling application to wait indefinitely. This may be unacceptable.

5. Client applications may be delayed waiting for the poll to complete.

The client applications may actually be delayed longer in this architecture than in the last one because they must wait for the poller to perform a delete and not just a select. There are also increased locking requirements in this architecture.

6. The possibility of deadlocks is introduced.

Since the poller and the clients are all modifying the same small table, it is possible for deadlocks to occur. Consider the case where the poller has acquired the first page of a two-page event table, but must wait for a client application to finish its transaction and release the second page. The client needs to make one more modification before the end of the transaction, but it needs to acquire the first page of the event table to do so. A deadlock will occur.

It is possible to avoid this problem by forcing both the trigger and the poller to acquire exclusive table locks before they continue. This will degrade the system's throughput, however.

Conclusion

The main benefit of this architecture is that events are never lost and that performance is increased. However, the intermediate event problem can be difficult or impossible to solve, and the architecture still suffers from the problems of polling.

CHAPTER 4
ASYNCHRONOUS TRIGGER TO OPEN SERVER RPC

Description

The Sybase system architecture has allowed one way to escape from T-SQL: the Sybase RPC. Sybase RPC's are not compatible with any operating system's RPC's, and they should not be confused.

In T-SQL, it is possible to execute a procedure on a remote server by predicating the call with the server's name.

```
/* Execute sp_who on the MathDataServer */

exec MathDataServer...sp_who


/*
**  Execute sharma's grade update procedure in the
**  cop5725 database on the server SYBASE
*/

declare @aGrade char (1)
select @aGrade = "A"

exec SYBASE.cop5725.sharma.updateGrade
           @student = "Vance", @grade = @aGrade
```

Figure 4-1: Examples Of Executing RPC's On Remote Servers

Not only is it possible to execute procedures on remote SQL Servers, it is possible to execute procedures on remote OpenServers.

OpenServer/C is a non-preemptive multithread C library available from Sybase upon which the SQL Server is based. In fact, it is the basis for most Sybase server and middleware products including OmniSQL Server, all Sybase gateways, and Replication Server. The library allows for a C program to become a server to multiple Sybase client programs. It allows the programmer to authenticate logins, receive language (e.g. SQL) requests, receive procedure calls, and return results in Tabular Data Stream (TDS) format, which all Sybase clients can receive.

It is possible to have a trigger call an OpenServer whenever a modification has occurred. Consider the following architecture:

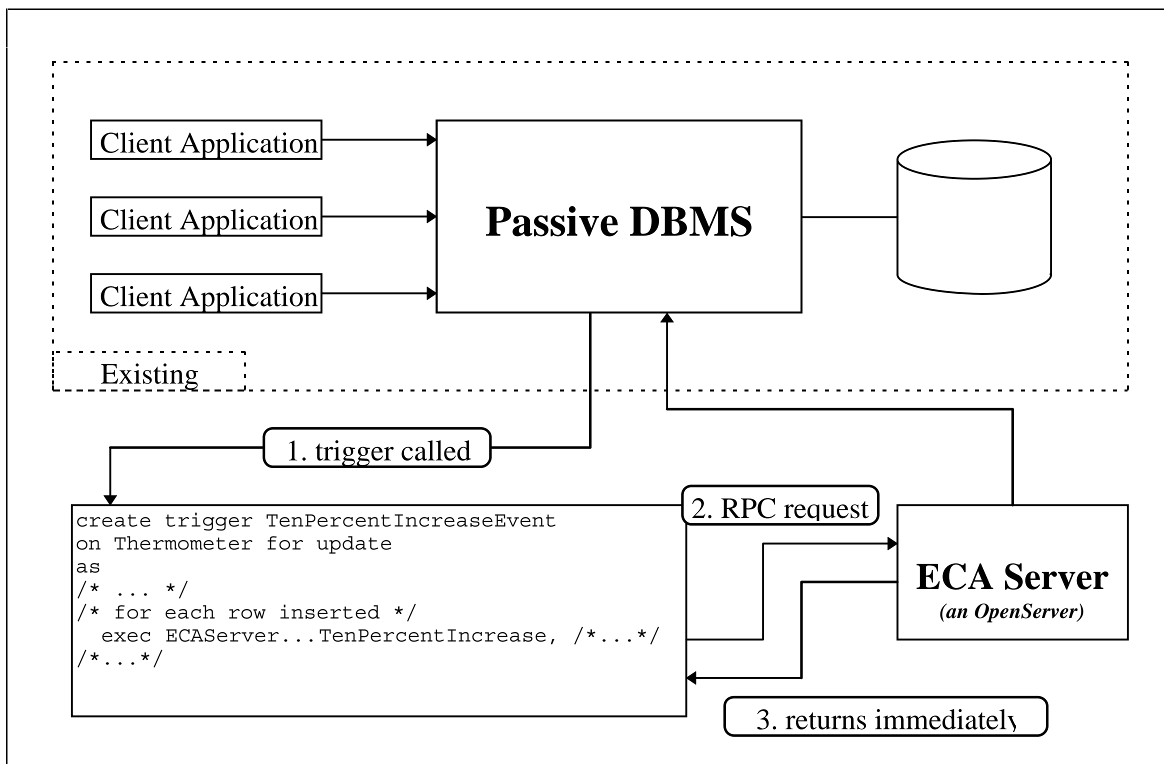


Figure 4-2: The “Asynchronous Trigger/OpenServer RPC” Architecture

The trigger can alert the OpenServer that a change in the database state has occurred and pass up to 255 (simple) parameters describing the change. Upon receiving the RPC notification, the OpenServer can immediately return a confirmation to the SQL Server and allow it to continue processing.

Since RPC's, like stored procedures, can only pass simple, atomic data values, the trigger must use a cursor to loop through the inserted and deleted tables.

```
create trigger RecordTenPercentIncreaseEvent
on Thermometer
for update
as

  declare @thermometerId int,
          @temperature   float

  declare cursor eventCursor for
    select i.thermometerId, i.temperature
    from   inserted i, deleted d
    where  i.thermometerId = d.thermometerId and
           i.temperature >= d.temperature * 1.10


  fetch eventCursor into @thermometerId, @temperature

  while (@@sqlstatus = 0)
  begin
    exec ECAServer...TenPercentIncrease
           @thermometerId = @thermom eterId,
           @temperature   = @temperature

    fetch eventCursor into @thermometerId, @temperature
  end

  close cursor eventCursor
  deallocate cursor eventCursor
```

Figure 4-3: Trigger Which Calls An RPC

If a transaction aborts, the server does not "roll back" the RPC calls. Once the RPC call has been sent, it can not "retrieved" when the transaction aborts. Even with deferred triggers, there would not be a way to simulate two-phase commit to an OpenServer. It would have to be integrated into the deepest level of the Sybase kernel.

Since this is the case, the ECA Server must "call back" the database to see if the event really took place. Of course, the ECA Server will have to wait until the calling transaction either commits or aborts to acquire read permission on the modified pages. Here is the revised architecture:

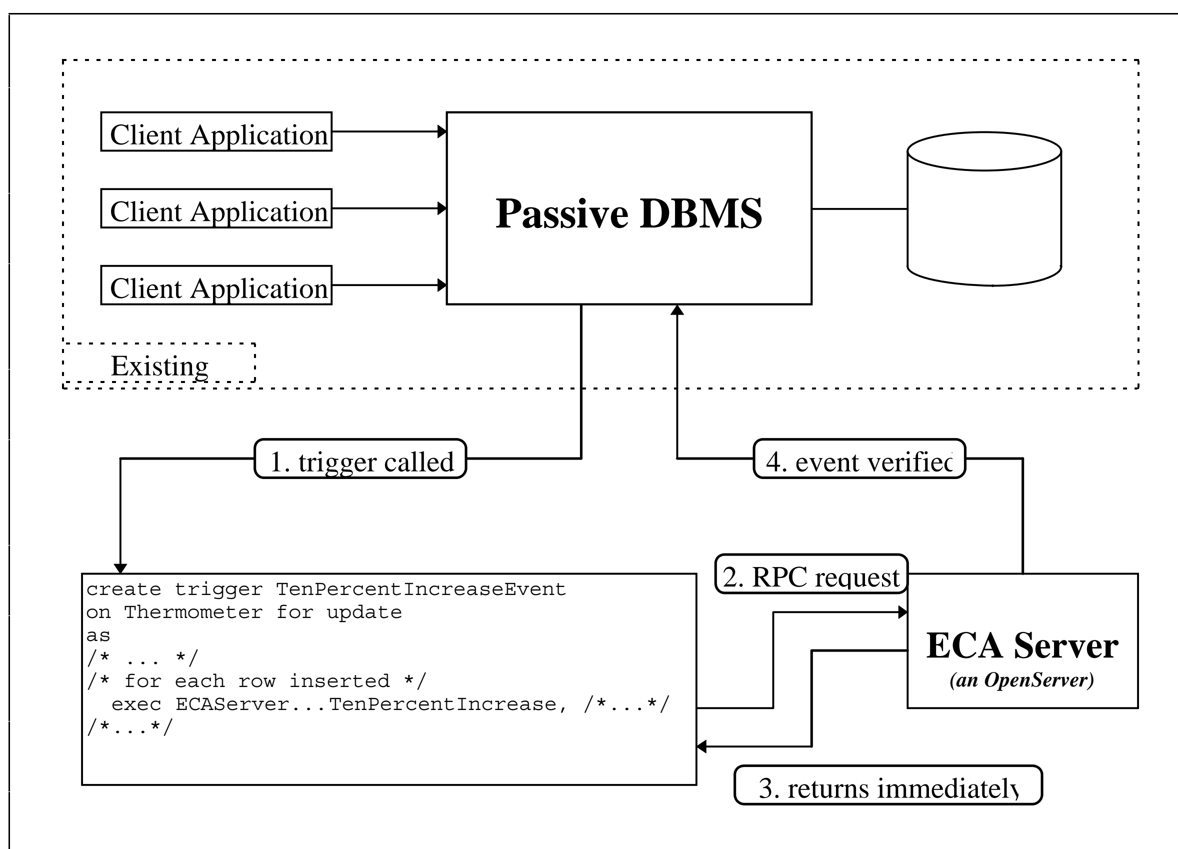


Figure 4-4: The Revised "Asynchronous Trigger/Open Server RPC" Architecture

By calling the server back, we can solve the problem where intermediate events are generated and deferred semantics are required. Now, the RPC becomes a notification that an event has possibly occurred. When the ECA Server logs back in, it must wait for the transaction to release its locks by committing or aborting. After the transaction has finished, only the final consistent state remains. Thus, any intermediate results that generated false events can be verified.

This method for verifying the validity of the event is not an guaranteed to be correct. By the time the ECA Server tries to access the data, another modification may have taken place, obscuring the previous results.

This can be solved by incorporating the event tables of the previous architecture. Instead of verifying the results by looking at the original tables, changes can be checked against the event tables. However, it is still impossible to tell which changes resulted from which transaction. We can only verify that some transaction produced the change. Figure 4-5 shows the revised architecture.

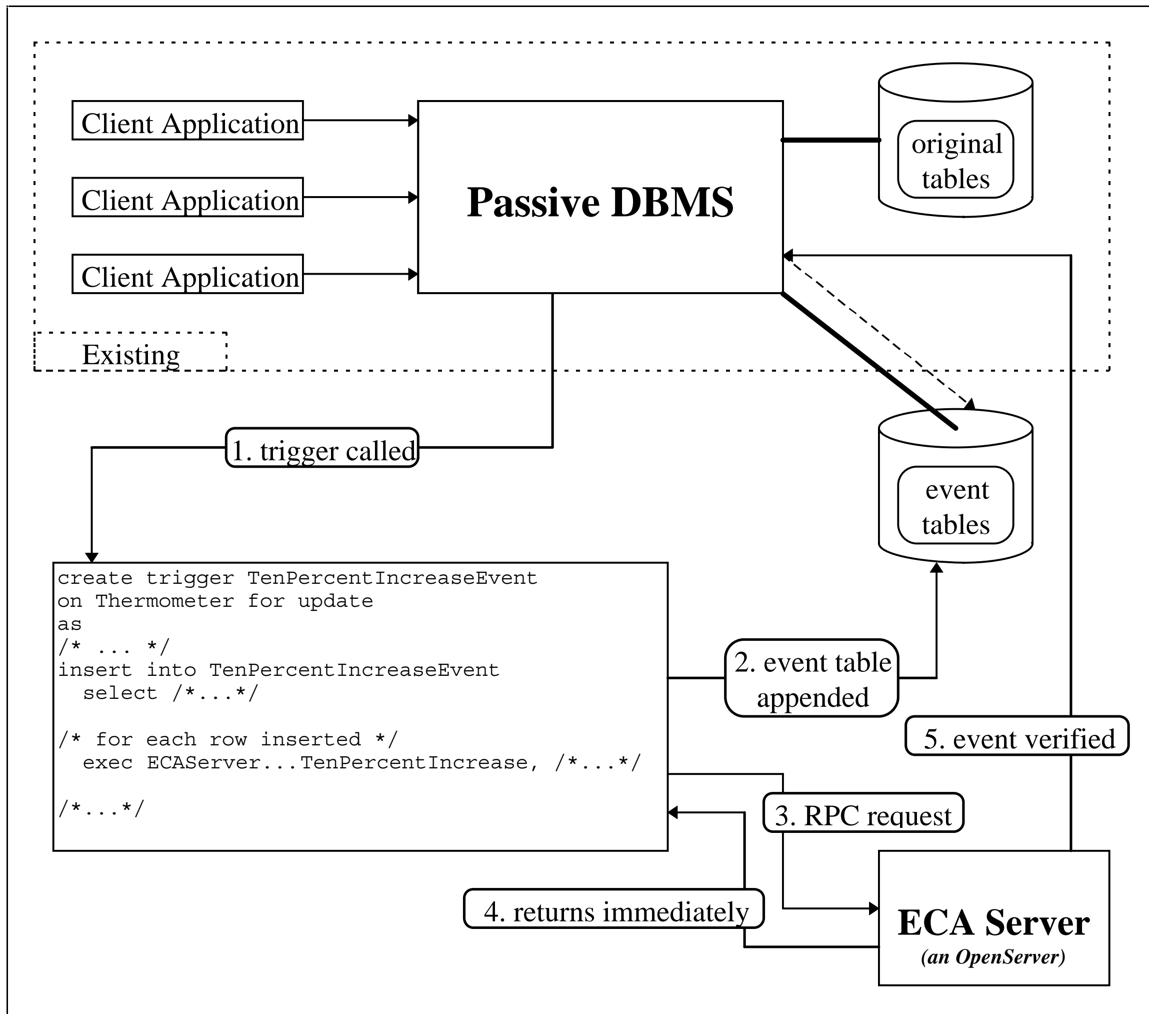


Figure 4-5: The Twice Revised "Asynchronous Trigger/OpenServer RPC" Architecture

<u>Features</u>

1. There is no latency in reporting events.

   The ECA server is immediately notified of any modifications.

2. No polling is necessary.

Since the dataserver is now actively reporting events, periodic polling is not necessary.

3. Existing applications remain unchanged, only minor database changes are necessary.

This method requires only some minor dataserver configuration and the addition of the triggers to the database. Changes are not required to the existing applications.

4. All modifications will be reported by the trigger.

Since a trigger generates the events, no modifications will be missed.

Limitations

This architecture seems very attractive, but there are some important limitations.

1. Events must be verified.

There is no way to roll-back an RPC event notification if a transaction aborts. Verification of events is required. It is not always possible to verify that an event has occurred.

2. Deferred semantics are not always achievable.

By the time the ECA Server logs back into the SQL Server, another modification may have taken place obscuring the previous consistent database state.

3. Poor performance.

The dataserver must reconnect to the OpenServer each time the RPC is made. While this is a fairly quick process, there may be a noticeable impact on performance if a large number are

required. Also, the transaction will have to wait for the non-preemptive ECA Server to accept a connection request.

To implement this architecture, the OpenServer should be a separate process in the operating system whose main purpose is to accept and process RPC calls. Otherwise, clients could experience long delays.

4. The ECA Server may be prevented from making additional inquiries.

If the transaction that has generated the RPC request is holding locks, the ECA Server will not be able to view the transaction's dirty data. In other words, if the ECA Server must make its own database inquiries during condition evaluation, it must obviously use a separate connection to the dataserver. Since it is a separate process, it will have to wait for the original transaction to release its locks before the server can access any of the transaction's dirty pages.

5. There is no way to determine the order of events.

We have a somewhat more accurate indicator than the previous architecture, however. In the process of verifying the results of a transaction, the ECA Server will block. The pages are not released until the completion of the transaction. Therefore, we can use the verification time to order the events. This is not always correct, but it is certainly a better estimate than we had previously. If the true ordering of events is important to the ECA system, this architecture is not suitable.

6. Poor robustness.

If the OpenServer fails, the events must be stored locally, like in the previous architecture. The trigger should check the value of @@error after the RPC has been made. If the SQL Server could not connect, the insert would still be made as usual. When the OpenServer restarts, it must check to see if any events have been queued at the server.

Even though all of the events can be recovered this way, the SQL Server RPC request will have to time-out each time. Even though the time-out period is configurable, the delay may be unacceptable to the client applications. A better solution is to maintain the state of the OpenServer on the SQL Server. If the Open server fails, the server can make a note and insert the events in the queue. When the OpenServer recovers, it can reset the flag. Figure 4-6 shows a trigger which can accomplish this.

```
create trigger RecordTenPercentIncreaseEvent
on Thermometer
for update
as

  declare @thermometerId int,
          @temperature   float,
          @serverStatus  int

  insert into TenPercentIncreaseEvent
    values (@thermometerId, @temperature, getdate ())


  select  @serverStatus = status
  from    OpenServer
  where   name = "ECAServer"

  declare cursor eventCursor for
    select i.thermometerId, i.temperature
    from   inserted i, deleted d
    where  i.thermometerId = d.thermometerId and
           i.temperature >= d.temperature * 1.10


  fetch eventCursor into @thermometerId, @temperature

  while (@@sqlstatus = 0 and @serverStatus > 0)
  begin
    exec ECAServer...TenPercentIncrease
                @thermometerId = @thermometerId,
                @temperature   = @temperature

    if (@@error > 0)
    begin
      select @serverStatus = 0

      update OpenServer
      set    status = 0
      where  name = "ECAServer"
    end

    fetch eventCursor into @thermometerId, @temperature
  end

  close cursor eventCursor
  deallocate cursor eventCursor
```

Figure 4-6: Trigger Which Calls An RPC With Error Handling

Note that this will lock the page containing the ECA Server's entry in the OpenServer table until this transaction completes. This will have the effect of blocking all of the other transactions that use the OpenServer. They will be prevented from selecting or updating the status until the transaction commits. Also, the change will not be permanent if the transaction aborts.

We can create a different OpenServer with an RPC that connects back into the server and changes the status asynchronously. This would successfully solve the locking problem, since the RPC call would not be part of the original transaction. Now if that server fails, we should reflect this in the OpenServer table. Of course, we are now back in the same position where we started. It would provide an additional level of robustness, however.

Conclusion

This architecture eliminates the latency of polling, but at the expense of delaying the initiating transaction. If the number of RPC calls is high, this will not be suitable.

CHAPTER 5
REPLICATION SERVER TO OPEN SERVER

Description

Sybase offers a product called Replication Server which is capable of re-creating the results of transactions on remote servers. While this is usually used to replicate data, it can also be used to generate events.

Replication server works by reading the SQL Server's transaction logs. It determines whether a transaction has been committed or rolled back, so only committed transactions are forwarded. SQL statements which modify more than one record are rewritten to modify one record at a time. For example, a delete statement which originally deleted two records will be turned into two delete statements which will delete one row each by primary key.

The replication server guarantees never to lose any modifications. The transactions are applied in the order they were committed at the primary. Transactions are applied serially in each database to minimize contention. Since this is a detached architecture, there is almost no impact on the performance of the SQL Server. The database only maintains a pointer to the last log record processed by the Replication Server's Log Transfer Manager process.

The SQL which the Replication Server generates can be sent to an OpenServer. The OpenServer can parse the SQL and generate events based on the modifications that were performed. The architecture is shown in Figure 5-1.

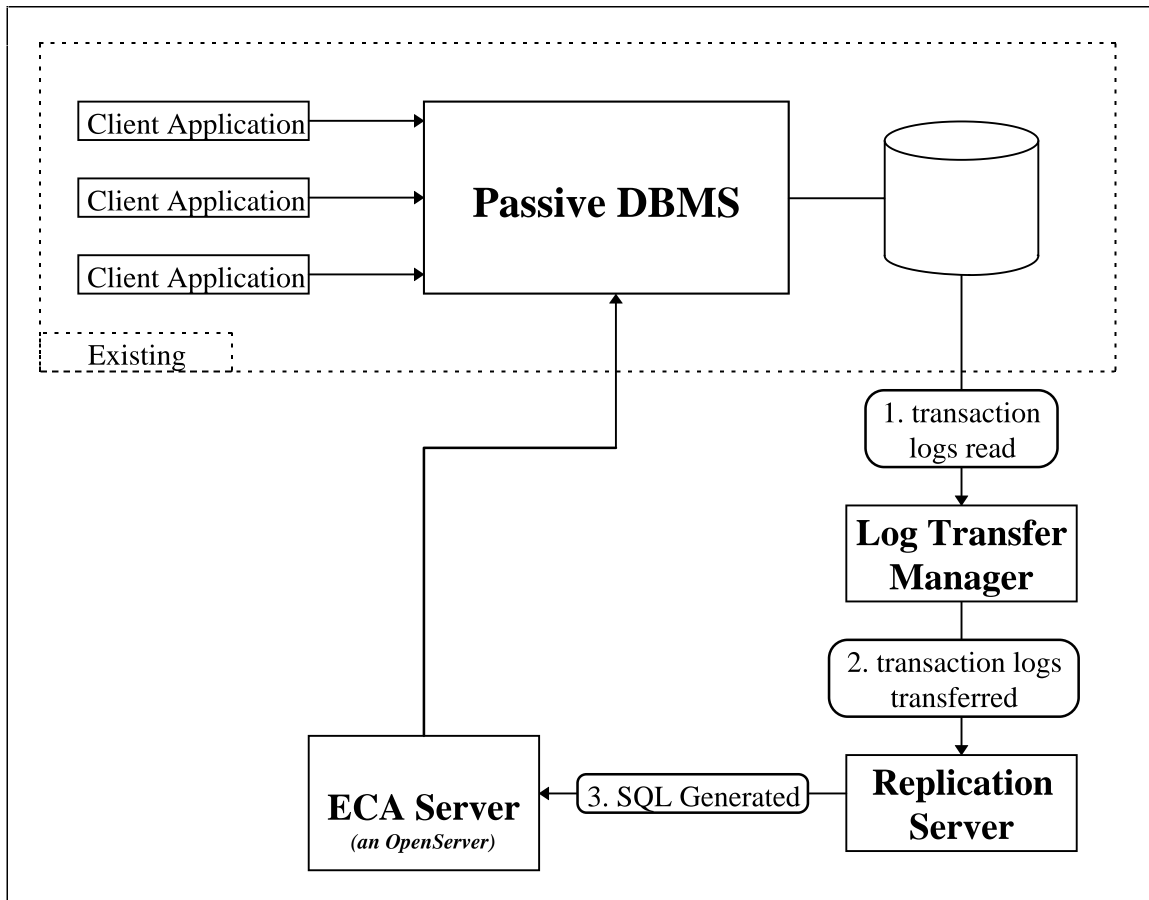


Figure 5-1: The “Replication Server To OpenServer” Architecture

Features

1. Excellent performance

The Replication Server architecture does not significantly degrade the performance of the SQL Server. The latency delay is very low, usually less than a second for small changes.

2. Events are generated in the correct order.

The ECA Server will be able to correctly order events because the SQL is sent in the order in which the transactions completed.

3. Intermediate modifications can be detected and compensation can be made.

Since the transactions that are sent by the Replication Server are all wrapped in begin transaction/commit transaction pairs, the ECA Server can compensate for any intermediate modifications. By examining the net result of the transaction, deferred events can be properly generated.

4. Events can not be lost.

The replication server will never lose any modifications.

5. No bogus events are generated.

Aborted transactions are not forwarded by the Replication Server

6. Robustness

Even if the Replication Server fails, the recovery is automatically handled.

7. Scaleability

Replication has been proven to efficiently handle hundreds of transactions per second.

Limitations

1. Latency

Although the Replication Server is very efficient, there is still a delay between the time that data is modified and the time the event can be generated. A maximum latency time can not be guaranteed.

Conclusion

This is the best architecture for implementing the detached active database semantics.

CHAPTER 6
SYNCHRONOUS TRIGGER TO OPEN SERVER RPC

Overview Of Immediate Semantics

While the previous four architectures supported only detached semantics, the next two additionally support immediate semantics. In immediate semantics, the ECA Server can decide whether to commit or abort a client's transaction. The events can be generated immediately as the transaction is executing, deferred until immediately before a transaction commits, as well as decoupled entirely to execute after a transaction completes.

Description

In the previous section, a trigger called an RPC on the ECA Server to inform the server of a possible event occurrence. In Sybase, it is possible to pass parameters both directions. A user-defined return status for the RPC may also be returned to the caller. A simple extension to the asynchronous architecture is for the trigger to wait for a confirmation from the ECA Server.

```
/* Execute sp_who on the MathDataServer */

declare @status int

exec @status = MathDataServer...sp_who


/*
**  Execute sharma's fibonacci calculator in the cop5555
**  database on the server SYBASE
*/

declare @value int


exec @status = SYBASE.cop5555.sharma.computeFibonacci
           @term = 10, @result = @value output
```

Figure 6-1: Examples Of Passing Parameters And A Return Status To A Remote Server

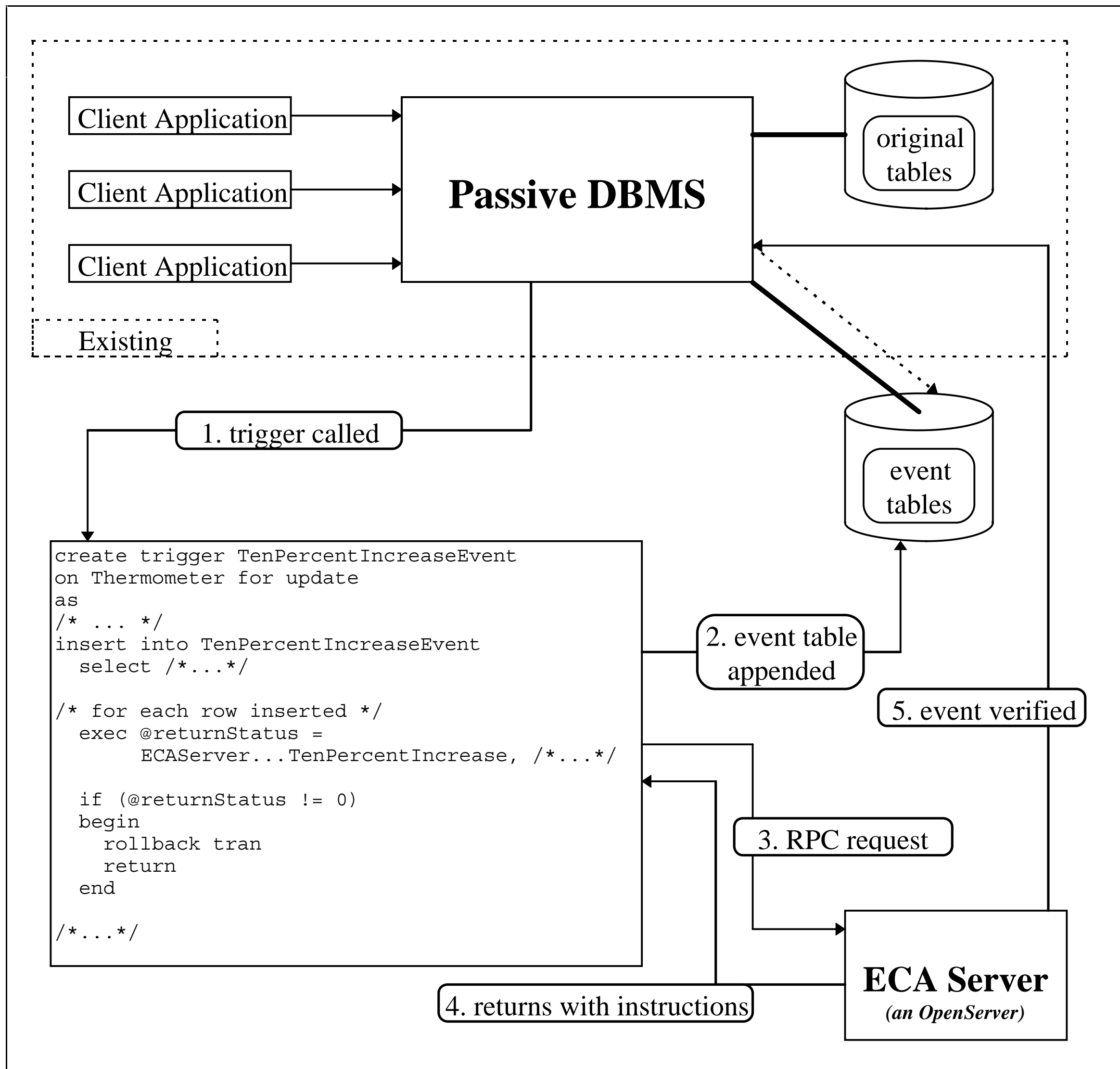


Figure 6-2: The "Synchronous Trigger To OpenServer RPC Architecture"

The return status from the ECA Server's RPC call should be checked to determine whether to abort or continue the transaction.

```
create trigger RecordTenPercentIncreaseEvent
on Thermometer
for update as
  declare @thermometerId int,  @temperature   float,
          @serverStatus  int,  @returnStatus  int

  insert into TenPercentIncreaseEvent
    select i.thermometerId, i.temperature
    from   inserted i, deleted d
    where  i.thermometerId = d.thermometerId and
           i.temperature >= d.temperature * 1.10

  select  @serverStatus = status
  from    OpenServer
  where   name = "ECAServer"

  declare cursor eventCursor for
    select i.thermometerId, i.temperature
    from   inserted i, deleted d
    where  i.thermometerId = d.thermometerId and
           i.temperature >= d.temperature * 1.10

  fetch eventCursor into @thermometerId, @temperature

  while (@@sqlstatus = 0 and @serverStatus > 0)
  begin
    exec @returnStatus = ECAServer...TenPercentIncrease
                           @thermometerId = @thermometerId,
                           @temperature   = @temperature
    if (@@error > 0)
    begin
      select @serverStatus = 0
      update OpenServer
      set    status = 0
      where  name = "ECAServer"
    end
    if (@returnStatus != 0)
    begin
      rollback tran
      return
    end
    fetch eventCursor into @thermometerId, @temperature
  end

  close cursor eventCursor
  deallocate cursor eventCursor
```

Figure 6-3: Trigger Which Calls An RPC With Error Handling And Return Status

It is also true in this architecture that the ECA Server must wait for the transaction to abort before it can access the transaction's dirty data. However, this limitation is much more serious in the synchronous architecture than in the asynchronous one. If the ECA Server even attempts to access pages held by the calling transaction, the system will be in a state of deadlock which is not detectable by the SQL Server. The ECA Server must constantly check itself with a separate connection to determine whether or not it is in a deadlock state. This is entirely possible to implement, but what status should be returned to the trigger if this happens?

Perhaps the ECA Server should allow the transaction to continue and then take remedial action using the previous architecture's callback semantics. The select request will obviously block until the transaction has either committed or aborted. After the transaction has committed or aborted, it will be too late to change the outcome of the transaction. The ECA may not even be able to correct the results of the previous transaction if another modification is already in progress.

In this architecture, only immediate trigger semantics may be used with confidence. Deferred semantics, like the asynchronous architecture, must be implemented using a database callback, which is prone to error.

Features

1. Time-constrained requirements are met.

If the system must take action within a specified time frame, this architecture provides for immediate response.

2. Existing applications remain unchanged, only minor database changes are necessary.

This method requires only some minor dataserver reconfiguration and the addition of the triggers to the database. Changes are not required to the existing applications.

3. All modifications will be reported by the trigger.

Since a trigger generates the events, no modifications will be missed.

Limitations

1. Immediate trigger semantics require verification.

There is no way to roll-back an RPC event notification if a transaction aborts. Verification of events is required.

2. Reliable deferred trigger semantics are difficult or impossible to achieve.

For the ECA Server to affect the outcome of a transaction based on a deferred event, it must take remedial action on the database. This is unacceptable.

3. Temporarily inconsistent states within transactions can generate events.

Again, without deferred triggers or the ability to determine the current transaction's transaction ID#, there is no way to guarantee the final result of offsetting modifications can be determined. If deferred semantics are required, this solution will not provide the correct results.

4. Deadlocks are very likely to occur

The ECA Server must maintain its own deadlock checking since the SQL Server can not be aware of the problem.

5. Poor performance

The dataserver must reconnect to the OpenServer each time the RPC is made. While this is a fairly quick process, there may be a noticeable impact on performance if a large number of RPC's are generated. Also, the dataserver will have to wait for the non-preemptive ECA Server to accept a connection request.

To implement this architecture, the OpenServer should be a separate process in the operating system whose primary purpose is to accept and process RPC calls. Otherwise, clients could experience long delays.

6. The ECA Server may be prevented from making additional inquiries.

The ECA Server needs to make its own database inquiries during condition evaluation, it must obviously use a separate connection to the dataserver. It will have to wait for the initiating transaction to release its locks before the server can access any of the transaction's dirty pages.

7. There is no way to determine the order of events.

The verification time is a good, but not an accurate indicator to determine order the commit times of the transactions. If the true ordering of events is important to the ECA system, this architecture is not suitable.

8. Poor robustness.

As in the asynchronous architecture, if the OpenServer fails, the SQL Server will experience delays waiting for RPC time-outs. Updating the status of the OpenServer can also have the effect of suspending other transactions which rely on the OpenServer until the transaction completes.

This problem is not unique to this architecture. Multiple points of failure are common with any tightly-coupled distributed architecture. It would be possible to run a backup OpenServer to handle RPC requests, but it may be impossible to transfer the state information of the failed OpenServer to its backup.

Conclusion

This architecture is only practical for using immediate trigger semantics with an ECA Server which does not need to make additional inquiries.

CHAPTER 7
OPEN SERVER GATEWAY

Description

When an OpenServer is placed as an intermediate between a client and a server, it is called a gateway. If we add a gateway between each client application and the SQL Server, we can solve many of the limitations in the previous architectures.

To reiterate, OpenServers look exactly like SQL Servers to application programs, so the programs would not have to be modified to run with this architecture.

The gateway server accepts a connection from a client and passes the authentication information along to the SQL Server. After the connection has been established, the gateway can take control of the client process. Language (SQL) and procedure requests from the client are pre-processed by the ECA Server to determine if events should be generated. Notice that in this architecture, a select event can be defined.

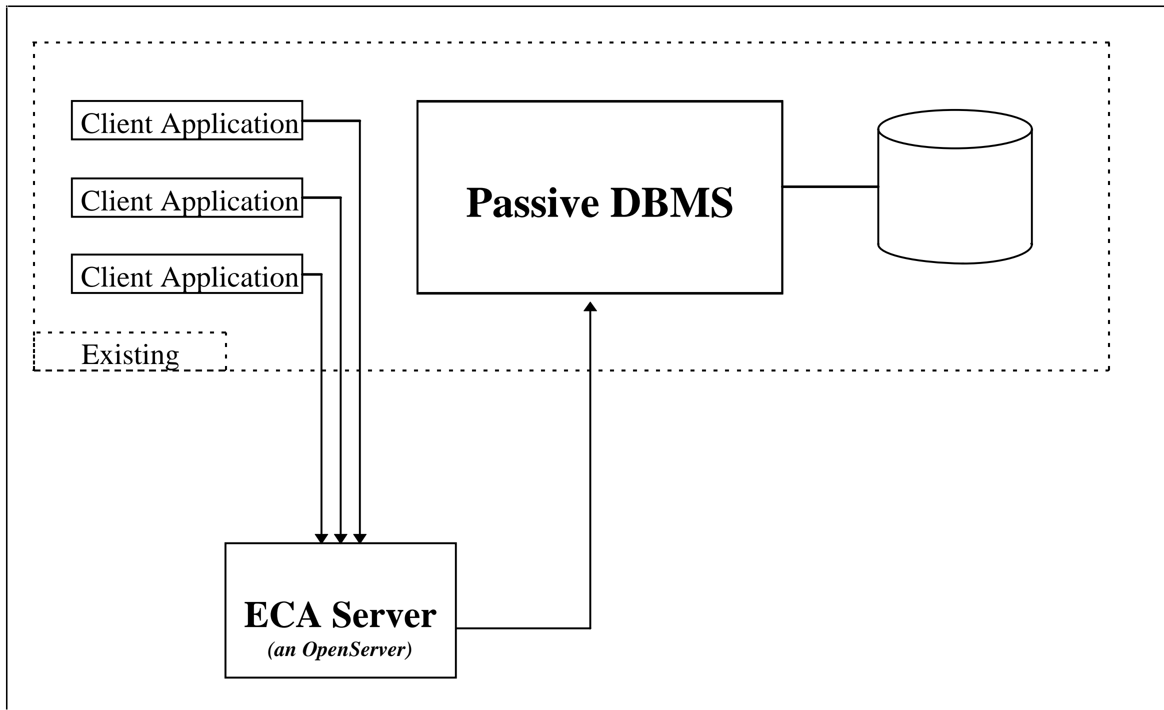


Figure 7-1: The "OpenServer Gateway" Architecture

This architecture is similar to a layered API approach, except the layer is truly transparent to the client. By using the gateway architecture, modularity is achieved and maintenance costs are reduced.

It is finally possible to use deferred trigger semantics in this architecture. Before any transaction is committed, the ECA Server can interrupt the transaction and perform inquiries on the SQL Server.

There are some limitations to generating events by only using the SQL Requests. For example, the ECA Server may not be able to determine which rows will be modified by a complex update statement. It may also be impossible to check afterwards. To solve this problem, triggers can be added to report the results of modification statements. Before sending back the event notifications, the SQL Server must notify the ECA Server that the results are for internal use and

are not to be returned to the client application. This may be easily done with a user-definable raiserror call which can be trapped at the ECA Server.

```
create trigger RecordTenPercentIncreaseEvent
on Thermometer
for update
as
  if (right (host_name (), 1) = '*'))  /* indicates gateway */
  begin
    /*
    **  Notify the gateway that an event has a occurred
    **  so it can intercept the output.
    */

    if exists (select i.thermometerId, i.temperature
               from   inserted i, deleted d
               where  i.thermometerId = d.thermometerId and
                      i.temperature >= d.temperature * 1.10)
    begin
      raiserror 200001 "TenPercentIncreaseEvent"

      select i.thermometerId, i.temperature
      from   inserted i, deleted d
      where  i.thermometerId = d.thermometerId and
             i.temperature >= d.temperature * 1.10
    end
  end
```

Figure 7-2: Trigger Which Returns Events To A Gateway

This solves the problem for immediate trigger semantics. For deferred semantics, the ECA Server can interrupt the client before committing and check the final state of the system based on the events it received.

Also, since the ECA Server is now the same process as the client application, the Server may make additional inquiries into the state of the system without blocking.

Features

1. Both immediate and deferred trigger semantics can be implemented.

By being able to interrupt the transaction before committing, this architecture achieves deferred semantics.

2. The ECA server may make additional inquiries without blocking

Since the ECA Server is the same process as the client application, no livelock or deadlock can occur.

3. Events are generated in the correct order.

Since the ECA Server is responsible for executing transaction commits, the ECA Server will be able to correctly order events in order of transaction completion.

4. No latency.

If the system must take action within a specified time frame, this architecture provides for immediate response.

5. Existing applications remain unchanged, only minor database changes are necessary.

This method requires only the addition of the triggers to the database. Changes are not required to the existing applications.

6. All modifications will be reported by the trigger.

Since a trigger generates the events, no modifications will be missed.

7. Excellent performance

This architecture does not significantly degrade the performance of the SQL Server. The gateway can be run on the same machine as the SQL Server to minimize the cost of sending event notifications.

8. Events can not be lost.

The ECA Server must process all events synchronously.

9. No spurious events are generated.

Aborted transactions do not generate bogus events because the ECA Server can implement a two-phase commit.

10. Scaleability

Multiple gateway instances can communicate with a central ECA Server since the gateway is only really responsible for event generation. By using two-phase commit for communication between the gateway and the central ECA Server, total ordering of transactions can still be achieved. If a single gateway fails, it will not affect the overall state of the ECA system.

11. Parallel execution of ECA Server and SQL Server

For events that are generated by SQL parsing, the ECA Server can process ECA rules while the SQL Server computes results. This was not possible with the previous architecture.

12. Portability.

It is possible to use the gateway to connect to other DBMS's besides Sybase which support triggers.

Limitations

There are no significant limitations in this architecture.

Conclusion

This is the best architecture for implementing the immediate semantics of an active database.

CHAPTER 8
RULE EVALUATION AND EXECUTION IN THE GATEWAY ARCHITECTURE

Introduction

In this section, the capabilities and limitations of the gateway architecture are described with respect to capturing events, evaluating conditions, executing actions, and supporting the HiPAC coupling modes.

Capturing Events

By raising events in triggers, all database modification events which take place through insert, update, and delete commands may be captured. In addition, the gateway can also detect and generate transaction commit, abort, and prepare-to-commit events. The ECA server can generate additional events by interpreting the SQL requests. For example, an event could correspond to the invocation of a stored procedure. Since the gateway is responsible for forwarding all of the SQL, an event could actually modify a request before submitting it to the server. Therefore it is possible to support a query-rewrite mechanism similar to POSTRGES. Detecting composite events must be accomplished by the ECA server itself by collecting and evaluating the atomic events it receives from the server. Temporal events may be specified, and must be managed by the ECA server.

While capturing modification events is straight-forward, capturing retrieval or selection events is impossible. Without integrating the active component into the DBMS, there is no way to tell which rows in a table were selected in either the result set or the predicate of the query.

A major limitation to Sybase's triggers is that they cannot execute commands outside the server, except by RPC. By evaluating and executing rules at the ECA server, the events, conditions, and actions can step outside the boundaries of Sybase. Hence, it is possible for the ECA server to monitor external systems and generate events based on their state.

Evaluating Conditions

The condition part of a rule may, of course, specify a query over the entire database. In addition, transition events may be specified, since Sybase triggers provide access both to the modified data and to the data that existed before the modification.

Conditions need not be limited to evaluating the state of the database. For example, a condition might inquire into the state of an external system during its evaluation. It could even create a dialog box at the user's console and ask for confirmation before proceeding. Conditions may be as sophisticated as necessary and are not restricted to SQL.

Executing Actions

Actions at the ECA server may be any database or non-database operation. When an action is executed, it can make additional modifications to the database and thus trigger additional rules. Since the ECA server manages the client connections, actions have the ability to abort the current transaction. Like conditions, actions may invoke operations outside the database, such as sending events to the user's application, communicating with another system, or accessing another database altogether.

In POSTGRES, an action can replace the triggering operation by using the keyword "instead." It is not possible to provide this functionality with the gateway architecture, since the

ECA server is dependent upon Sybase triggers to generate events. Since the operation has already occurred when the trigger is executed, the triggering operation cannot be replaced. An action must be expressly written to compensate the effect of the operation if these semantics are required.

Rule Execution Semantics

Since Sybase triggers only support set-oriented operations, namely insert, update, and delete, the minimum granularity of rule processing is a set of tuple-level operations. It is impossible to capture events at the level of the individual tuple without changing Sybase's semantics.

Sybase supports the concept of transaction save-points, so error recovery is possible during rule processing. If a particular rule fails during execution, or the results of a particular action need to be undone, the ECA server can roll back the transaction to the proper place.

Coupling Modes

Like HiPAC, the gateway architecture can support the immediate, deferred, and decoupled modes of coupling between events and conditions, as well as between conditions and actions. Immediate semantics are easy to implement. The ECA server can use the application's connection, or another connection if desired, to execute commands in the middle of the transaction. If, because of locking or other reasons, the rules must use the client application's connection to Sybase, the rules must obviously run serially. If locking is not a concern, another database connection can be used the rules may run concurrently.

For deferred and detached semantics, the events must be collected and stored until the end of the transaction. When the ECA server determines that a transaction is ready to commit or has

committed, it can then perform its operations on the server. In this case, the rules are considering the net effect of the transaction on the database. However, the before and after image of the modified tuples are available to the server.

CHAPTER 9
GATEWAY DESIGN AND IMPLEMENTATION

Introduction

To illustrate the power of the gateway architecture, this section describes the design and implementation of an actual active database gateway component. Although this implementation does not support every possible active database feature, it does provide a substantial amount of flexibility and illustrates the potential of this design.

A very simple and powerful gateway implementation is to separate the ECA server into a communication module and a rule module. The communication module is responsible for negotiating all of the communication between the clients and the Sybase SQL Server. In addition, this component must route events which are generated by the SQL Server to the rule module and allow the rule module to control a user's transaction. The rule module is responsible for handling the events, evaluating the conditions, and executing the actions with the proper semantics.

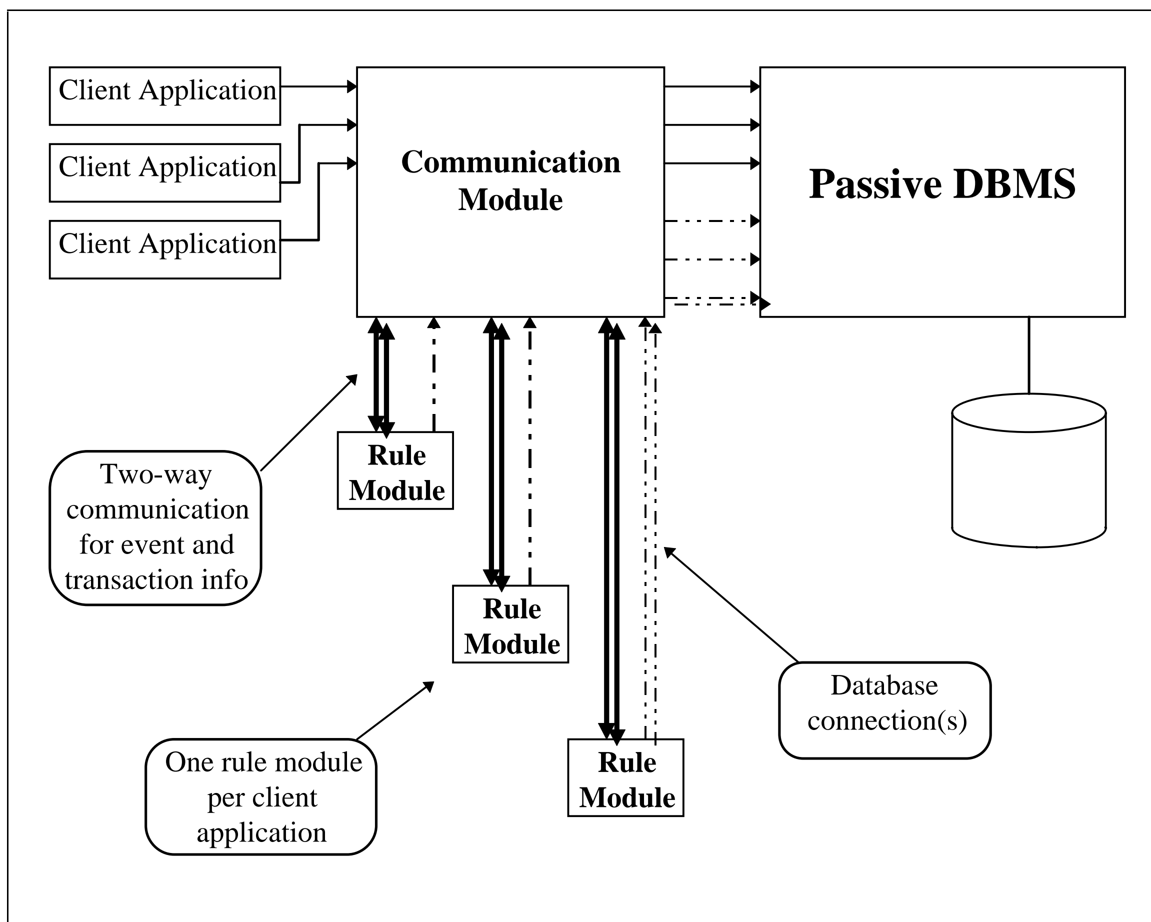


Figure 9-1: A Possible Gateway Implementation

The Communication Module

One multi-threaded communication module stands between the Sybase SQL Server and all of its client applications. When it receives a connection request from a client, it opens a connection on the Sybase server with the user's login and password. Through this connection will pass all of the user's SQL commands and all of the server's results. After successfully connecting to the SQL Server, the communication module starts a rule module by creating a heavyweight process, retaining control of the rule module's stdin, stdout, and stderr file descriptors.

When a database event occurs, the communication module receives the event parameters from the SQL Server. The event and the associated parameters are then forwarded to the rule module for evaluation. The rule module may either use the same database connection as the client application or may establish its own connection through the gateway to evaluate and execute the rules on the server. For example, the rule module might need to connect to the Sybase as the system administrator to perform a special operation which the user does not have permission to run. Alternatively, the rule module might want to evaluate rules in parallel. All of the event handling takes place behind the scenes, and the client application sees only a Sybase SQL Server connection.

When a client disconnects, the communication module closes the database connection and terminates the rule module.

The Rule Module

One rule module is created for every user connection. When a user connects to the communication component, the user's connection profile, including the name and password, are sent to the rule module so that the rule module may make additional database connections as that user. The rule server does not connect directly to Sybase, but rather connects back through the communication module so events may be captured. As mentioned before, the rule module may make additional connections if necessary, either with the user's connection profile or with another profile.

All of the conditions to be evaluated when an event occurs are stored in a table in the Sybase database. For this particular implementation, the condition and action code is written in TCL, but any interpreted language can be substituted. When an event occurs, the rule module

retrieves all of the conditions associated with an event and evaluates them in turn. The conditions which make additional database inquiries are evaluated through a default database connection unless otherwise specified. Since the events are written in TCL, any legal command may be executed on the rule module's interpreter. This gives the architecture tremendous power and flexibility. Conditions and actions may be activated or deactivated on the fly, and the very condition and action code may be changed if desired. Whether or not this is a good feature, it does show what is possible with this implementation. Although it is contrary to the von Neuman architecture model, self-modifying code is occasionally used in practice.

If a condition evaluates to be true, the rule component retrieves all of the actions associated with that event and condition from a table in the Sybase database. With this design, a particular action may be taken whenever a certain condition is true, no matter what event precipitated the evaluation of the condition. Actions may generate other events, send message to running applications, or perhaps start new ones.

Conditions and actions are prioritized in the database so a particular order of evaluation and execution may be enforced. These conditions and actions may also be active or inactive depending upon the state of the system, and may be toggled on the fly. It is possible that new conditions and actions could be generated by the rule handler at run-time as well.

For detached coupling mode, the rule module asks the communication module to send a notification upon completion of the transaction. The detached condition or action code resumes execution after the notification is sent. The rule module is responsible for collecting and executing the appropriate code, and the communication module is only responsible for informing the rule module of the transaction state.

For deferred trigger semantics, the rule server asks the communication server to suspend the transaction before completion and allow the rule server to take control. The rule server then either commits or aborts the transaction after the production rules have been evaluated and executed.

Conclusion

This implementation of the gateway concept is very powerful and usable. Unfortunately, the flexibility is a double-edged sword. A more robust design would not give the rule programmer quite so much liberty and attempt to prevent infinite loops in execution. Nevertheless, this architecture illustrates many of the features of the gateway architecture and provides an example of the practicality of the design.

CHAPTER 10
CONCLUSION

By decoupling the active component from the database, production rule support may be added to an existing Sybase system without any loss of functionality. Nearly the full range of active semantics can be supported without using an integrated active database architecture. The gateway concept described earlier is a particularly powerful example of how events at the SQL Server can be captured in a way which is transparent to the database clients.

Considering the recent proliferation of Sybase database installations and the large number of existing systems which have already been built on this technology, the prospect of seamlessly integrating an active component is very attractive indeed. For many users, this may be the only practical way to begin using production rules today. Additionally, because the active database component is external to Sybase, its power is not limited to what the database can provide, but only by what the architect can design and implement.

Since the client-server paradigm is rapidly being surpassed by the multi-tiered architecture, a decoupled active database architecture may be the best prospect for the future of active databases. Perhaps database researchers should work to specify an industry-standard protocol by which the many different DBMS's could communicate with a middle-tier active database component.

REFERENCES

[ANW93] E. Anwar, L. Maugis. A New Perspective on Rule Support for Object-Oriented Databases. In Proceedings of the ACM SIGMOD International Conference on Very Large Databases, 1993.

[CHA89] S. Chakravarthy, et al. HiPAC: A Research Project in Active, Time-Constrained Database Management (final report). Technical Report XAIT-89-02, Xerox Advanced Information Technology, Cambridge, Massachusetts, 1989.

[DAY94] U. Dayal, E. N. Hanson, and J. Wisdom. Active Database Systems. In Modern Database Systems: The Object Model, Interoperability, and Beyond. Addison-Wesley, Reading, Massachusetts, 1994.

[DAY88] U. Dayal, et al. The HiPAC project: Combining Active Databases and Timing Constraints. SIGMOD Record. 17(1):51-70, 1988.

[HAN92] E. Hanson, Rule condition and action execution in Ariel. Proceedings of the ACM SIGMOD International Conference on Management of Data, 1992.

[ING92] INGRES. INGRES/SQL Reference Manual, Version 6.4. ASK Computer Co., 1992.

[ORA92] ORACLE. ORACLE7 Reference Manual. ORACLE Corporation, 1992.

[STO90] M. Stonebreaker, A. Jhingran, J. Goh, and S. Potamianos. On Rules, Procedures, Caching, and Views in Database Systems. In Proceedings of the ACM SIGMOD International Conference on Management of Data, 1990.

[STO91] M. Stonebreaker and G. Kemnitz. The POSTGRES Next-Generation Database Management System. Communications of the ACM. 34(10):78-92, 1991.

[SYB96] Sybase. Sybase SQL Server Reference Manual: Volume 1. Sybase, Inc., 1996.

[WID91] J. Widom, R. Cochrane, and B. Lindsay. Implementing Set-Oriented Production Rules as an Extension to Starburst. In Proceedings of the Seventeenth International Conference on Very Large Databases, 1991.

BIOGRAPHICAL SKETCH

David Vance was born in Waynesboro, Virginia in 1969. He received his undergraduate degree in Computer Science from the University of Delaware, Newark, Delaware, in January 1993. He expects to receive his Master of Science degree in Computer and Information Science from the University of Florida, Gainesville, Florida, in August 1996. He has been working as a Sybase database administrator since receiving his undergraduate degree in 1993. His current research interests lie in integrating databases with multi-tiered architectures, particularly in the Java/Neo/Joe environment.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Sharma Chakravarthy, Chair
Associate Professor of Computer and Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Eric N. Hanson
Assistant Professor of Computer and Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Douglas D. Dankel, II
Assistant Professor of Computer and Information Science and Engineering

This thesis was submitted to the Graduate Faculty of the Department of Computer and Information Science and Engineering in the College of Liberal Arts and Sciences and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Master of Science.

August 1996

Dean, Graduate School