Flexible and Recoverable Interaction Between Applications and Active Databases

Eric N. Hanson **I-Cheng Chen** **Roxana Dastur**
Kurt Engel **Vijay Ramaswamy** **Chun Xu**

CIS Department
University of Florida
Gainseville, FL 32611
hanson@cis.ufl.edu

September 28, 1994
CIS-TR-94-033
(revised November 15, 1994)

Abstract

The need for active database rules to be able to interact with running application programs has been recognized for some time. In addition, technology that supports recoverable, transaction-consistent flow of requests from client applications to the database server and of responses from the server back to the client is maturing. The work described here unifies a flexible mechanism for interaction between applications and an active DBMS with recoverable messaging services. The resulting system, an extension of the Ariel active DBMS, allows messages to be transmitted by an active rule action to one or more applications. These messages can be made persistent, via a recoverable queue, so they will not be lost due to server, client, or communication failure. In addition, messages transmitted by rules to applications can be delayed until transaction commit to prevent applications from taking action on the results of an uncommitted transaction. This paper describes the design and implementation of the system, including the system architecture, application/server communication facilities, event processing mechanisms, and security and authorization techniques.

1 Introduction

This paper describes the design of a facility that supports reliable interaction between application programs and the Ariel active database system. Ariel is an active database system based on a production system model [HCKW90, Han92]. A new command called **raise event** is introduced. This command can be called from the action of a rule or can be submitted directly by a user or application. A companion of **raise event** is **register for event**. Client programs can register for an event and will receive the event and process it if it is subsequently raised. A distinguishing feature of the system is that event processing is recoverable in the face of failures of the Ariel system, the client application, or the network.

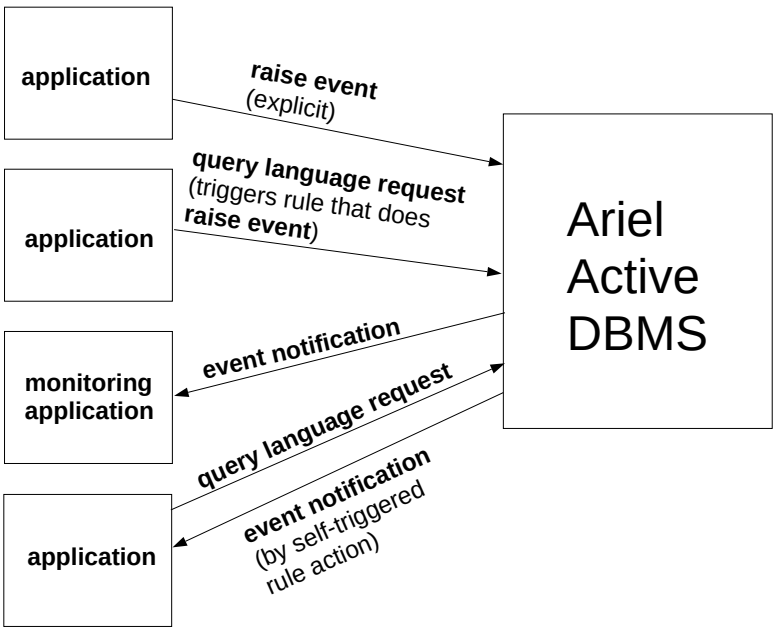

Figure 1: Examples of event processing in extended Ariel DBMS.

If application developers so desire, it can be guaranteed that if an event is raised, each client program registered for the event will process the event at least once. In other words, it is guaranteed that the system will not lose an event due to a failure. The problem whereby events may be lost and not acted on by clients is called the *lost dependent operation* (LDO) problem. The new version of Ariel being developed prevents this problem from occurring.

Also, it is often important that clients not act on events generated as a result of uncommitted transactions. A system that may let a client process an uncommitted event signal suffers from the *dirty dependent operation* (DDO) problem. The extended version of Ariel being developed avoids this problem by allowing event transmission to depend on commit of the transaction raising the event. As part of the solution to these problems, recoverable queues [BHM90] are used to help communicate events from servers back to clients.

Of course, the primary way applications will interact with an active database server is to send the server requests and get back responses. This style of interaction must be well supported. The main goal of this work is to provide an integrated, flexible framework for interaction between an active DBMS and applications. This includes standard communication of application requests and server responses, as well as the new form of communication where the server sends events to applications.

An example of different ways event processing can be carried out in the extended Ariel architecture is shown in figure 1. The figure illustrates that applications can raise events explicitly, raise events indirectly via a rule, monitor events raised by other applications, and monitor events raised by themselves.

The extended Ariel system now under development will support the following styles of interaction between the active DBMS server and application programs:

- direct, conversational transaction processing,

- queued transaction processing based on recoverable queues,
- an event mechanism where event messages can be:
  - **immediate** or **deferred**, i.e. they can be transmitted immediately after they are raised, or deferred until the raising transaction commits,
  - **direct** or **queued**, i.e. they can be sent with fast, volatile messages, or non-volatile messages sent via a recoverable queue that are slower but still normally arrive in a timely manner.

Supporting an event mechanism whereby the active database server can signal client applications adds a new dimension to the way applications and the server communicate. This work shows how event communication can be made flexible enough to allow application developers to choose among varying levels of performance and resilience to failure, depending on their needs. Moreover, it is shown how the communication mechanisms used for events can be made to dovetail with the existing communication mechanisms that support processing of application requests by the server.

2 Background

Traditionally, all interaction between an application program and the database server has been driven by the application. The application submits requests and receives responses, normally in a synchronous, conversational fashion. Many commercial systems also support asynchronous transaction processing, where request messages from applications are placed in stable storage, the server processes them and saves the reply on stable storage, and later the application retrieves the reply [GR93]. The stable storage used for message transmission is sometimes managed by a recoverable queue mechanism [BHM90] tailored to support this type of asynchronous transaction processing. A number of commercial systems support this type of processing, including, to name a few, IMS/DC and CICS from IBM, Tandem's Guardian 90, Digital's DECdta, and the TOP END TP-monitor from AT&T GIS [GR93, Ber91, ATT94].

The emergence of active databases has added a new dimension to interaction between applications and database servers. The action of a rule can invoke an operation in a running application program [MD89, ASK92]. This is a very useful facility because it allows any action desired to be taken based on the firing of a database rule. Actions that might be taken as a result of a rule firing, such an electronic purchase of stock market securities, transmission of an electronic mail message, or ringing someone's beeper, all require execution of application code written in a high-level programming language. It is not feasible to put into the database server all the programming mechanisms needed to carry out these kinds of actions. An application must be involved. Moreover, carrying out these actions in an application process separate from the DBMS provides an added degree of reliability and security compared with running user procedures inside the DBMS server process.

Providing a facility to let rules communicate with a running application is clearly a good idea. However, a simple design of such a system is vulnerable to errors due to transaction,

client, and server failures. Simple mechanisms to allow rules to signal applications have been developed for HiPAC [MD89] as well as the commercial INGRES system, versions 6.4 and higher [ASK92], and the Borland Interbase product [Das94].

The HiPAC mechanism was implemented as a main-memory prototype, and served more to validate the concept that a rule could signal an application than as a tool for application development. The INGRES mechanism allows an application program to create events and register for events. A database procedure invoked by the action of a rule can raise an event. If an application is registered for an event, the application is notified if the event occurs, and can take appropriate action. Applications can get information about events either by polling or by an interrupt-driven notification mechanism. The INGRES mechanism transmits the event notification to the application directly, without waiting for the transaction that raised the event to commit. This may be desirable in many circumstances, but can lead to the DDO and LDO problems. The event mechanism of Interbase is comparable to that if INGRES.

3 Commands For Event Processing

First, the commands used for event processing are described. Then the commands are used in examples, and potential uses of some of the features of the commands are discussed.

3.1 Command Description

The following commands and procedures are provided as the interface to the event system:

define event Declares the existence of an event with a specific name and parameter format. The general form of the command is:

```
define event <eventname> ( <parameters> )
[ mode [ is ] [ deferred | immediate ] ]
[ handler [ is ] [ all | raiser | any ]
[ grant register access [ to ] [ all | <auth-list> ] ]
[ grant raise access [to] [all | <auth-list> ] ]
```

Square brackets indicate optional clauses. A vertical bar indicates that exactly one of the symbols inside the enclosing brackets will be used. The **mode** clause allows specification of a default coupling mode to be used for the defined event when it is raised. The **deferred** mode defers transmission of the event to applications until the triggering transaction commits. The **immediate** mode allows the event to be transmitted immediately after it is raised. An event with no **mode** clause will have mode **deferred** by default, since this avoids transmission of uncommitted events, though it will delay event transmission.

The **handler** clause specifies how the event is to be processed by the application(s). If **all** is specified then all applications registered for the event will receive it. If **raiser** is

specified then only the application that caused the event to be raised will be notified. If **any** is indicated then any one (and only one) of the applications registered for the event will be selected to process it. If the handler clause is not present the default will be **all**.

Not every user can register for an event or raise an event. The **grant register access** and **grant raise access** clauses allow an event to be defined with raising and registration rights granted to all users or a restricted set of users. The right to raise or register for an event can be limited to a set of users by specifying an authorization list (`auth-list`) consisting of user ids or names identifying groups of user ids. Additional commands not shown here are provided to change the access rights associated with an event, and to create and manipulate user groups.

raise event This command allows an application to generate an event. The general format of the command is:

```
raise event <eventname> ( <parameters> )
[ from <from-list> ]
[ where <qualification> ]
[ mode [ is ] [ deferred | immediate ] ]
[ handler [ is ] [ all | raiser | any ]
```

An event is raised for a *set* of tuples that match the qualification in the **where** clause. The from-list allows specification of tuple variable bindings, just as is possible in the POSTQUEL **retrieve** command. If no tuples qualify, then the event is not raised. It is possible to override the default coupling mode and handler specification when raising an event.

RegisterForEvent This procedure signals the application runtime system that this application wishes to receive the specified event. When a client registers for an event, the registration is first validated to make sure that the event exists and the client has the privilege to register for it. If validation succeeds, the registration is stored in a persistent event catalog. The server processes the event registration inside a database transaction. The RegisterForEvent operation is a procedure rather than a POSTQUEL command since the client runtime system must do some local work to prepare for processing the event. The format of the procedure is:

```
int a_RegisterForEvent(char* eventName,
                       int eventNotificationMode,
                       int handlerFunc(a_Event*))
```

Procedures and types defined in the Ariel runtime library begin with `a_` to avoid naming conflicts with user-defined procedures and other procedure libraries. In `a_RegisterForEvent`, the eventNotificationMode can be **DIRECT** or **QUEUED**. It determines the way in which the notification is sent to the client. If it is **DIRECT**, an Ariel server will send

the notification to the client via RPC when the event is raised. If it is **QUEUED**, the server will place the notification in a recoverable queue and do an RPC to the client telling it that it has an event notification in the queue. A client must connect to their recoverable queue before registering for any events in QUEUED mode. The handlerFunc is a pointer to a user-supplied function which will be called when the client is notified. The parameter of type `a_Event` of handlerFunc points to a structure that contains information the handlerFunc uses to do event processing. If the handlerFunc is null, the client application will poll for event notifications, otherwise, an interrupt-driven environment is assumed, where the client acts immediately when notified.

Since the event registration catalog is persistent, if the server machine or an Ariel server process crashes and restarts, all registrations are retained. If a client crashes and restarts, it is expected to re-register for all events it is interested in. If a client previously registered for an event with the QUEUED event notification mode, then when it re-registers, it can begin processing any event notifications that are queued in the recoverable queue for it. If an application is registered for an event with DIRECT notification mode, and a server process attempts to notify the application without success, by default it will retry up to three more times at intervals of ten seconds. The number of retries and the interval between them can be changed by the system administrator. If none of the notification attempts succeed, the server assumes it has lost touch with the application and invalidates the registration for that event for that application.

Wildcard matching

An event name can be specified in the form `"prefix*"` where `prefix` is zero or more alphanumeric characters. This allows registration for a group of events with a single a_RegisterForEvent call. E.g., to register for all events, one could say:

```
int a_RegisterForEvent("*",DIRECT,genericHandler);
```

To register for all events starting with the string `SECURITY` one could say:

```
int a_RegisterForEvent("SECURITY*",DIRECT,genericHandler);
```

In the above examples, genericHandler is an application function that is written to be able to process any kind of event. Events that match the wildcard pattern are transmitted to the application only if the application has the necessary privileges. If the application does not have the needed privileges, the application is not notified when the event occurs, and is not even given an error message.

link confirm event This links one event as the confirming event for another. The command format is:

```
link confirm event <eventname2> with <eventname1>
```

The confirming event <eventname2> is raised on completion of the transaction that raised the original event <eventname1>. Every event has a unique ID. Event <eventname2> must take two arguments, the ID of the original instance of <eventname1> that was raised, and a CommitFlag indicating whether the transaction raising <eventname1> committed or aborted.

drop event Deletes the definition of an event from the system. The general form of this command is:

```
drop event <eventname>
```

drop confirm event Removes association of <eventname2> as the confirm event for <eventname1>. The general form of the command is:

```
drop confirm event <eventname2> with <eventname1>
```

The commands described above provide a general framework for defining, raising, and processing of event signals. The mechanism whereby a client application registers for and processes events will be discussed later.

3.2 Examples and Discussion of Command Usage

As an example, consider a database application suite designed to monitor and control the flow of natural gas in a gas pipeline network. A diagram of an event processing setup that could be used to perform this task is given in figure 2. If the flow of gas last recorded for the main pipeline is too low, the desired result might be to raise the main pump pressure by 10%. The gas flow may be logged in this table:

```
flowLog(time,sensor,pipeline,flowRate)
```

A set of monitoring applications would record records in flowLog periodically. An event to signal low main pipeline flow could be defined as:

```
define event lowMainFlow(time=float,flowRate=float)
```

A rule to check for low flow in the main pipeline could be:

```
define rule MainFlowTest
on append to flowLog
if flowLog.pipeLine="main"
and flowLog.sensor = 100
and flowLog.flowRate < 500
then raise event lowMainFlow(flowLog.time, flowLog.flowRate)
```

A monitoring application could register for lowMainFlow and if it received lowMainFlow, could increase the pressure generated by the main pipeline pump. The application code might look like this:

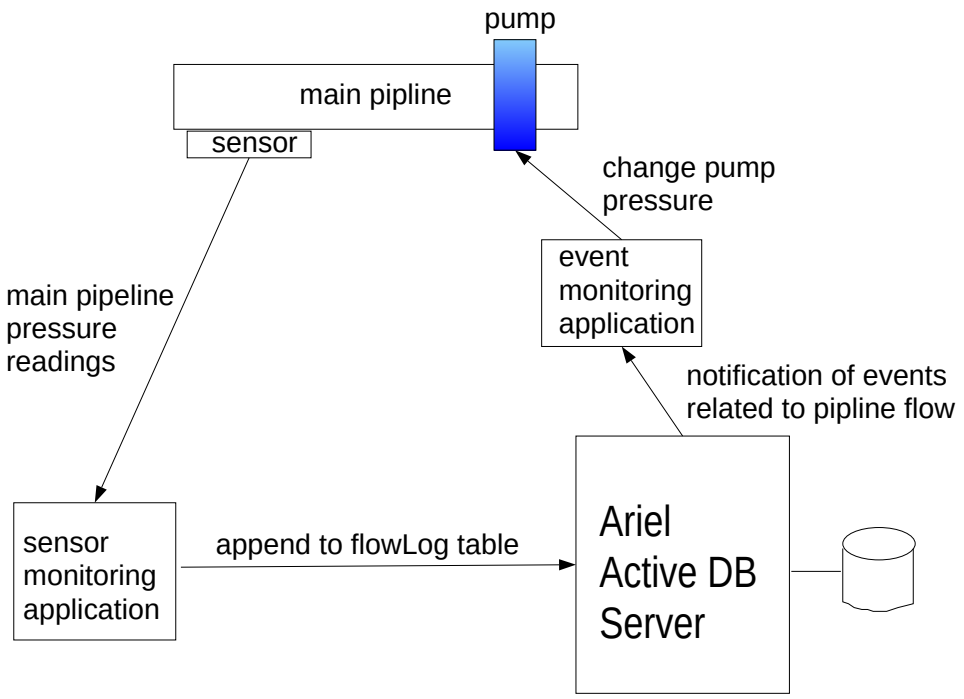

Figure 2: Example of event processing in extended Ariel DBMS.

```
...
int lowFlowHandler(a_Event *h) {
  /* Handle low flow event by
     increasing main pipeline pressure. */
  ...
}

main() {
  ...
  /* Register to handle lowMainFlow event, requesting
     to get event messages directly (not via a queue),
     using function lowFlowHandler to handle the event. */

  a_RegisterForEvent("lowMainFlow",DIRECT,lowFlowHandler);
  ...
}
```

This example illustrates how rules, events, and application programs can work together to carry out the intentions of an application system. A more detailed discussion of the client library functions that can be used to register for and process events is given later.

Regarding the options on the **raise event** command, An example of when the **handler** clause is useful is during application and rule testing. During testing, it might be useful for a rule action to have only the raiser as the handler, so programs other than a test program aren't subjected to a **raise event** from a rule that isn't yet debugged.

An example of when the **mode** clause is useful is a case like the following:

```
define event newAuditRecord(time=float,operation=c40,amount=float)
```

One rule may wish to raise newAuditRecord in deferred mode, and another may wish to raise it in immediate mode. One application may be interested when any records are appended to an Audit table, even in an uncommitted transaction. Another application may only want to know about new committed Audit records.

An example of how **link confirm event** can be useful is the following. Consider a database for a bank. The manager of the bank may wish to be notified of all attempted withdrawals of $10,000 or more, and whether they succeeded or failed. If a withdrawal fails due to a balance that is too low, the withdrawal transaction is rolled back. One rule could be set up to notify the manager when the withdrawal is attempted, and it could be linked to a confirming event. The confirming event would provide information that would be used to inform the manager whether the withdrawal succeeded or failed.

Another use of **link confirm event** is to improve performance by allowing an application notified of an event to work concurrently with a transaction that raised the event. If a long transaction is running, and it raises an event, the application listening for the event can prepare some work based on the event, but not take any irreversible actions until receiving a confirmation that the event-raising transaction committed. If the raising transaction rolled back, the work could be discarded.

Finally, **link confirm event** can also be used to allow an application to compensate for any action it took in response to an event raised by a transaction that later rolled back.

4 System Architecture

The system architecture of the Ariel event processing system is based on a client-server model. Client application programs can be written in a high-level language (C or C++) with embedded query language statements written in Ariel's version of POSTQUEL [SR86, Han92]. A client application establishes communication with server processes through a multi-threaded *data communications* (DC) process. When a client will have multiple interactions with a server, a session is established between them and the client is given a *server handle* to identify the server temporarily dedicated to the client. There are three types of servers:

Direct Ariel to execute direct requests from clients,

Queued Ariel to execute requests from clients sent through a recoverable queue, and

Queue Manager to perform operations on recoverable queues used for the flow of requests from clients and responses from servers back to clients, as well as the flow of event notifications from servers to clients.

More than one instance of each of these types of servers may be running simultaneously. The components of the system interact via remote procedure calls (RPC). This architecture is diagrammed in figure 3. The standard form of interaction is direct transactions. Client programs can call the following procedures to control direct transactions:

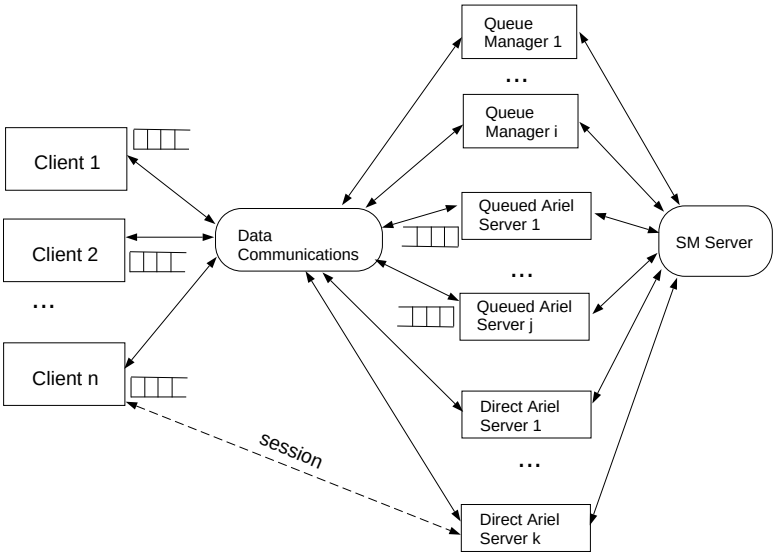

Figure 3: Ariel client-server architecture supporting event processing

`a_BeginWork()` starts a transaction,

`a_CommitWork()` commits a transaction,

`a_RollbackWork()` aborts a transaction.

When a client program calls `a_BeginWork` it establishes a *session* with a single Direct Ariel server process and that server becomes dedicated to the application until the application issues `a_CommitWork` or `a_RollbackWork` or the application suffers a *timeout*. A timeout occurs if the application does not communicate with the Ariel server for too long a time. Timeout causes the transaction in progress to abort. Establishing a session for the duration of a transaction is required because due to the way the Ariel server is implemented on top of the EXODUS system, there is no way one transaction can have work done by more than one Ariel server.

A BeginWork request is processed by the DC. The DC gets an Ariel server for the client and returns the server's identity in a server handle. If an Ariel is free, the original RPCs return value contains the handle. Otherwise, if an Ariel is not free, the RPCs return value contains a status code telling the client that an Ariel will be available later. Upon getting this code the client must be prepared to receive an RPC with the Ariel handle in it subsequently.

POSTQUEL commands, cursor operations, and other requests from the client to the Ariel server are done by having the client do RPCs directly to the Ariel server and get the responses back in the RPC return value. This design limits the interaction between the client and the DC or the client and the Ariel server to a single RPC and return in most cases, keeping down communication costs.

The multi-threaded DC process [Ram93] acts as both a server and a client. When it comes up it starts n ariel_threads to communicate with n Ariel servers (a one to one mapping), a Queue Manager thread to communicate with each Queue Manager, two high-priority RPC

servers – the Direct Ariel server thread and the Queued Ariel server thread, a wake-up thread, and a very high priority scheduler thread.

When a client does an RPC on the DC, a high priority server thread handles the RPC, preempting the lower priority ariel_threads. On getting a request the DC checks if an ariel_thread is free, which implies that an Ariel process is free. If so it returns the handle to the client. If no Ariel is free, the DC puts the request in a local data structure called the WorkQueue and returns a status code indicating this to the client. The client then must be prepared to receive an RPC later from the DC server when an Ariel process becomes free.

RPCs impose a limit of at the most one client inside the remote procedure. For this reason the DC server thread has the highest priority and executes the minimum required code so that it becomes available to other clients at the very earliest.

The process architecture described here is a variation of the "many servers, one scheduler" architecture [GR93], where the DC is the scheduler and the Ariel processes and QM are the servers. This architecture was chosen for simplicity, though in extreme cases, the DC could become a bottleneck, and failure of the DC process could bring down the whole system until the DC can be restarted. Since the focus of this work is on client-server interaction via the event mechanism, it was not felt to be worthwhile to complicate the design by using the more general "many servers, many schedulers" architecture [GR93].

5 Recoverable Message Queuing

Queued transactions are provided for application environments where an immediate response to a request is not essential, and messages must be able to flow reliably between clients and servers even if client, server, or network failures occur. Queued transaction processing involves three steps, each executed as a separate transaction:

1. A client puts a request into a durable input queue.

2. A version of the Ariel server called a Queued Ariel reads the request from the queue, processes the request, and enqueues a reply in a durable output queue.

3. The client gets the reply out of the output queue.

The same type of recoverable queues used for request and response transmission for queued transactions can also be used to provide recoverable transmission of event notifications to application programs. An economy of mechanism is achieved since durable queues are used for both purposes. When used for normal request and response transmission, Ariel's queue system ensures that requests and replies match. Of course, when a server asynchronously sends an event notification to a client through a recoverable queue, it doesn't correspond to any request, so request/reply matching isn't necessary. Our recoverable queue mechanism is similar to the one described by Bernstein et al. [BHM90]. However, we believe that the use of recoverable queues to transmit event notifications to clients is new.

The model of how clients interact with servers for non-interactive requests and queued event notifications is shown in figure 4. As illustrated in the figure, each client has a recoverable request queue, reply queue, and event queue. The request and reply queues are

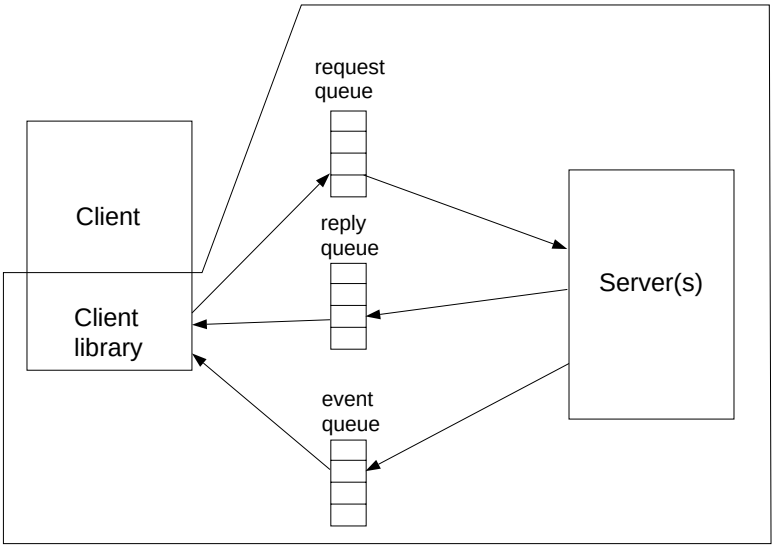

Figure 4: System model for queued request, reply, and event processing.

for the normal flow of requests and replies to those requests. The event queue holds event notifications sent for events the client registered for in QUEUED mode.

The interface to the queue system presented to the client application program consists of the following procedures:

cd = Connect(clientID) Connect to the queue system. The connection descriptor **cd** returned contains the following information:

LastSentRID ID of last request sent.

LastReceivedRID ID of last request received.

LastCommittedRID ID of last received request that was marked "completed."

LastReceivedEID ID of last event notification received.

LastCommittedEID ID of last event notification received that was marked "completed."

Disconnected True if client issued a Disconnect since previous Connect operation.

ReceiveCkpt Checkpoint data sent with last Receive operation.

EventCkpt Checkpoint data sent with last ReceiveQueuedEvent or CommitReceiveQueuedEvent operation.

Disconnect() Shut down the session with the Queue Manager by disconnecting from its queues. This procedure is successful only if there are no pending messages left in the reply or event queues.

Send(req,RID) Enqueue a request **req** with request ID **RID** in a single transaction.

r = Receive(ckpt) Receive the next reply **r** to a request. This implicitly does a "CommitReceive" (see below) of the previous message received if it hasn't been done explicitly.

r = ReReceive() Receive the reply returned by the last Receive operation.

CommitReceive(ckpt) Mark the last queue entry received as "completed."[1] Pass the client's current state related to request and reply processing as **ckpt**.

e = ReceiveQueuedEvent(ename,ckpt) Receive the next event with name **ename** in the event queue as **e**, and save the event state checkpoint information **ckpt**. The value of ckpt is the state of the client related to all events processed, up to and including the previous event received. This implicitly acknowledges the last event received, regardless of the name of that event. If **ename** is NULL, then the next event ready is received, regardless of event name.

e = ReReceiveQueuedEvent() Get the event returned by the last call to ReceiveQueued-Event.

CommitReceiveQueuedEvent(ckpt) Mark the last queued event received as "completed," and save the checkpoint information **ckpt**. The value of ckpt is the state of the client related to all event notifications processed so far, including the one being marked "completed" now.

The functional interface described here is similar to the one given by Bernstein [BHM90], with the addition of the function CommitReceive for normal request and reply processing and ReceiveQueuedEvent, ReReceiveQueuedEvent, and CommitReceiveQueuedEvent for queued event processing. The CommitReceive function allows an application to explicitly acknowledge that it is done processing the last Receive. This is not essential, but it can be used to make it less likely that a client will see pending replies when it restarts and connects to the system. If the CommitReceive function is not used, then the next Send implicitly acknowledges the last Receive. It is conceivable that there could be a long time between a Receive and the next send. Explicit use of CommitReceive can reduce the interval of uncertainty regarding whether the receive was processed.

The ReceiveQueuedEvent and ReReceiveQueuedEvent operations are analogous to the Receive and ReReceive, except they pertain to receiving event notifications from the event queue. Similar to CommitReceive, CommitReceiveQueuedEvent marks the last event notification received as "completed."

The client application goes through state transitions with respect to its request and reply queues and event queue as illustrated in figures 5 and 6, respectively. When a client connects to the system and wishes to start submitting requests, it must make sure it processed it's last reply. If the **disconnected** flag returned by Connect is true, then Send can be issued immediately. The client also knows that if it does ReceiveQueuedEvent that it will definitely not have processed the event received before. If the LastSentRID and LastReceivedRID returned by Connect are different, the client should do a Receive. If they are the same, and LastCommittedRID is different from LastReceivedRID, and the client wants to ensure at-least-once reply processing, the client should do a ReReceive and reprocess the reply. If the LastReceivedEID and LastCommittedEID are the same, then the

[1] This operation was called CommitClient in [Das93].

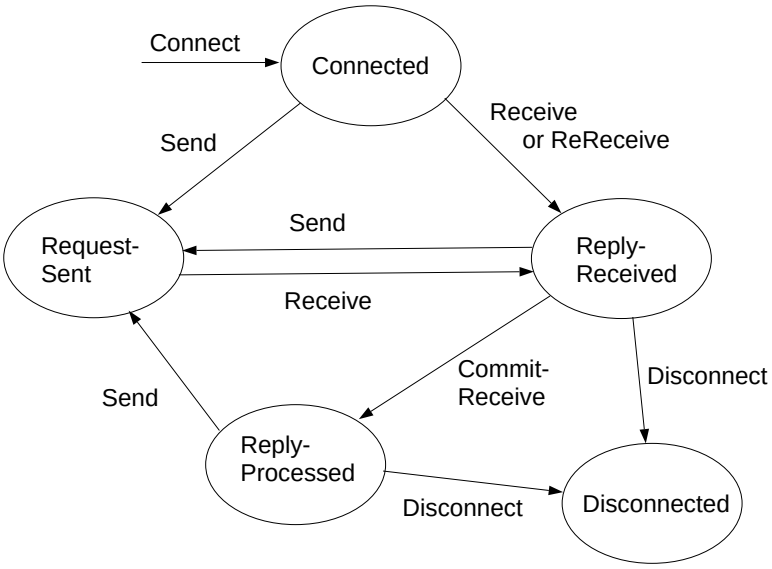

Figure 5: Client state transitions for queued request and reply processing.

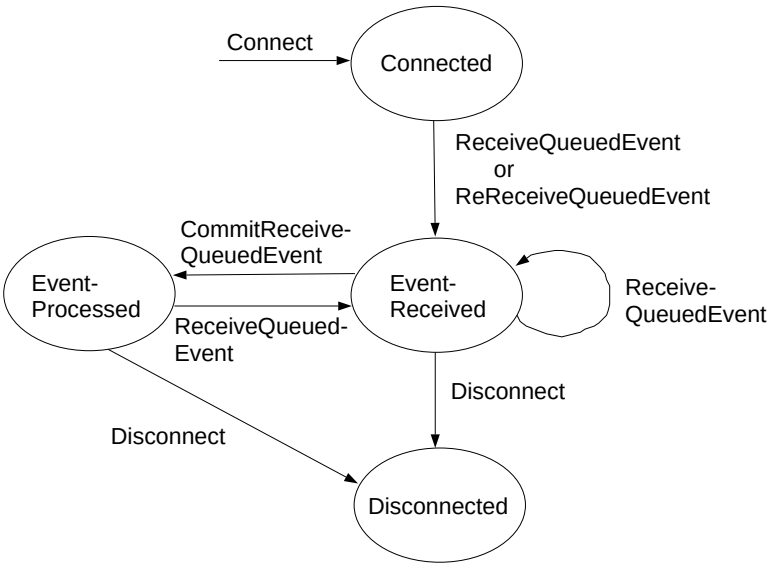

Figure 6: Client state transitions for queued event processing.

client can to a ReceiveQueuedEvent immediately. If they are different, the client should do a ReReceiveQueuedEvent and process the event to make sure it is processed at least once. If the client needs to make sure that it only processes a reply or event notification once because processing it more than once is not acceptable (e.g. it is not idempotent), then the application needs to be able to test its local environment to determine whether it actually processed the reply [BHM90].

If a client's state is small, it can use the checkpoint parameter of the Receive, CommitReceive, ReceivedQueueEvent and CommitReceiveQueuedEvent operations to save it's state to help it initialize itself after it connects. If the client has separate pieces of state information for different event names, it must checkpoint all of this state every time it receives or commits an event notification. An alternative design would have been to allow state to be checkpointed separately for different event names, but this was felt to be too complex, especially since the total event-related state for all event names will usually be small. See [BHM90] for a more detailed discussion of issues related to exactly-once message processing, and use of a client state checkpointing mechanism.

The queues are implemented as persistent C++ (E language) [RCS89] objects. The Queue Manager, Direct Ariel, and Queued Ariel processes are all allowed to access the queues directly, and perform queue operations as separate transactions or parts of other transactions as needed. The Queue Manager is the real owner of the queues. It generates persistent logical input and output queues and an event queue for a client when the client issues a **Connect** request to be connected to a Queued Ariel server. It stores requests from a client in the assigned queue in a transaction-consistent manner. It is responsible for seeing that the client receives a reliable response, if any, to a given request. The Queue Manager cleanly shuts down the queue when a session between a client and a Queued Ariel process ends.

The queue system provides functions **GetRequest** and **SetResponse** that enable the Queued Ariel server process to read requests from the queue and store responses in the queue. These two functions are executed directly by the Queued Ariel process. The QM process has a function called **DeleteTrans** which is used to delete completed transaction requests and their replies, as well as event notifications marked "completed," from the queue associated with a particular client. This is called by the QM when a client issues a **Disconnect.** It is also called as needed by the QM to garbage-collect completed messages.

The Queue Manager (QM) is set up as an RPC server, with an interface consisting of the client-library queue management procedures described above, as well as corresponding server-side procedures. In addition, the QM also acts as a client to the client application when the QM notifies the client that a response or event notification is waiting for the application. In this way, the QM supports asynchronous event processing by a client application.

6 Event Processing

6.1 Execution of RAISE EVENT in the Server

A **raise event** command can be executed directly by a user or application, or as part of a rule action. In either case, every **raise event** command is associated with one or more

event tuples. When **raise event** executes, this set of tuples is sent to the client, along with an event descriptor describing the event name, event parameter names and types, and other necessary information. When **raise event** is executed directly by a user or application, an implicit **retrieve** query is created with a target list equivalent to the parameter list of the **raise event** command, and an identical **from** and **where** clause. This query is run to find the event tuples to transmit to the client. When **raise event** is executed as part of a rule action, there is an implicit binding between tuple variables that appear in the rule condition, and tuple variables that are present in the **raise event** command in the rule action. A detailed discussion of rule condition-action binding is given in a previous paper on Ariel [Han92]. Essentially, the tuple variables that appear in both the rule condition and the rule action range over the set of new tuples that have just matched the rule condition. For example, consider this table format and rule:

```
emp(eno, name, sal)

define rule raiseAlerter
if emp.sal > 1.1 * previous emp.sal
then raise event bigRaise (emp.eno, emp.name, previous emp.sal, emp.sal)
```

Suppose that the following command is run to increase employee salaries:

```
replace emp (sal = emp.sal + 5000)
```

This command would raise many employee's salaries more than ten percent. Hence, a large number of tuples would be bound to the condition of the raiseAlerter rule. The bigRaise event would be raised once, and all the tuples bound to the rule condition would be logically included in the event notification sent to a client registered for the event. The actual implementation of event notification uses a stream mechanism so that a client can consume event tuples in medium-sized groups instead of having to receive all of them immediately. This avoids overwhelming the client. The same stream mechanism supports processing of tuples retrieved by a client as the answer to a query.

6.2 Transmission of Event Notifications to Client

In the new Ariel system, when an event is raised, a number of different mechanisms can be employed to process the event. The event catalog is always consulted to determine which client(s) are to receive the event. The other steps taken depend on whether:

- the event is raised in **immediate** or **deferred** mode,
- the client to which notification is to be sent registered for the event in **queued** mode or **direct** mode.
- the client has registered to process events in **polling** mode or **interrupt-driven** mode.

The mode in which an event is raised can be specified either as a default, or explicitly when the event is raised, but it is always either immediate or deferred. A client registered

for an event in queued mode will get notifications for that event from a recoverable queue. Otherwise, event notifications will be sent directly to the client via non-recoverable RPCs.

The following are the possible configurations for event notification, and the steps that must be taken:

1. Immediate mode, direct notification

```
Server: send notification to client RPC listening thread
        commit transaction

Client: if polling: enqueue notification locally on receipt
                    dequeue later, process
        if interrupt-driven: process notification on receipt
```

2. Immediate mode, queued notification:

```
Server: tell QM to enqueue notification in a separate transaction
        QM sends "queued event ready" message to client via RPC
        commit transaction

Client: if polling:  enqueue message locally
                     dequeue event from QM when ready
        if interrupt-driven:
                     dequeue response from QM on receipt of message,
                     process
```

3. Deferred mode, direct notification

```
Server: commit transaction
        send notification to client RPC listening thread

Client: if polling: enqueue notification locally on receipt,
                    dequeue later, process
        if interrupt-driven: process notification on receipt
```

4. Deferred mode, queued notification

```
Server: directly enqueue notification in durable queue
        commit transaction
        send "queued event ready" message to client via RPC

Client: if polling:  enqueue message locally
                     dequeue event from QM when ready
        if interrupt-driven:
                     dequeue response from QM on receipt of message,
                     process
```

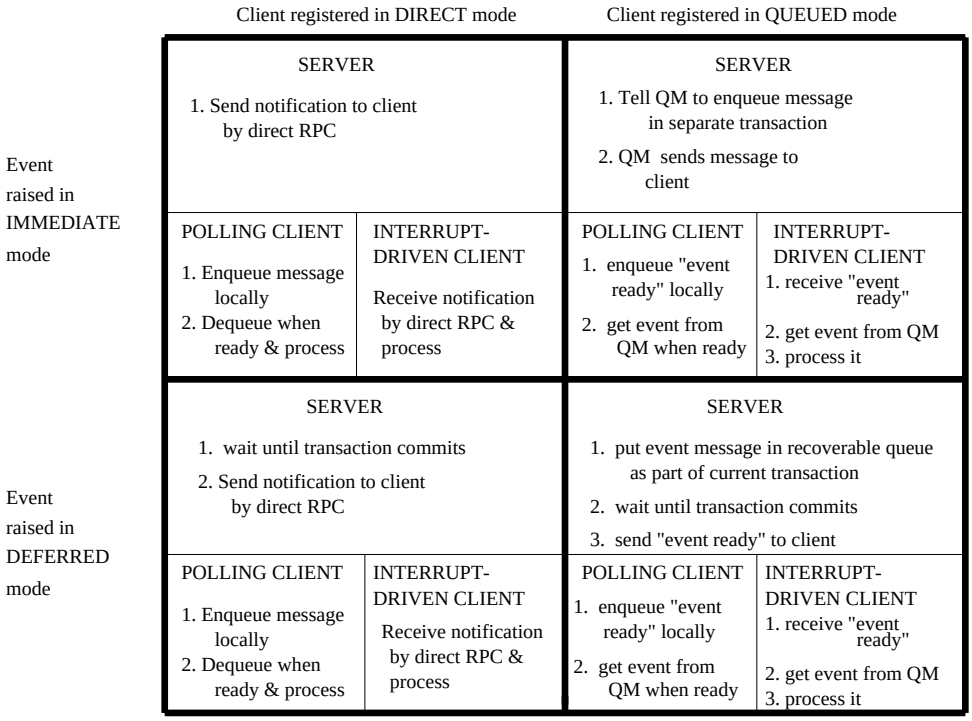

Figure 7: Illustration of different notification methods and responsibilities

The actions of the client and server in each of the four cases is summarized in figure 7. When the client is registered in queued mode, and notification is deferred, then the server itself enqueues the event notification as part of the current transaction. That way, if the current transaction rolls back, the event notification will be removed from the queue. As seen above, regardless of whether the client is using polling or interrupt-driven event handling, the client will need to be able to receive an RPC at any time. How this is accomplished is discussed below.

7 Client Runtime Library

A single client runtime library is provided to allow development of applications programs. This library supports both the functions of accessing the database directly, and processing event notifications. Applications are compiled and linked with the library. This library contains all the code needed for communicating with the system directly or via durable queues. The library functions make use of the Sun RPC system for communication. When the applications run, they must first call the function a_Initialize to initialize runtime library structures and communication with the DC server. Afterwards they can call other library functions to access the database and process events. The application programming interface provided by the library is described in appendix A.

The internal structure of a client application is shown in figure 8. There are two threads in the client application. One is the **user thread** which contains the actual application logic of the client. The other is the **RPC thread** which waits for incoming RPCs from the

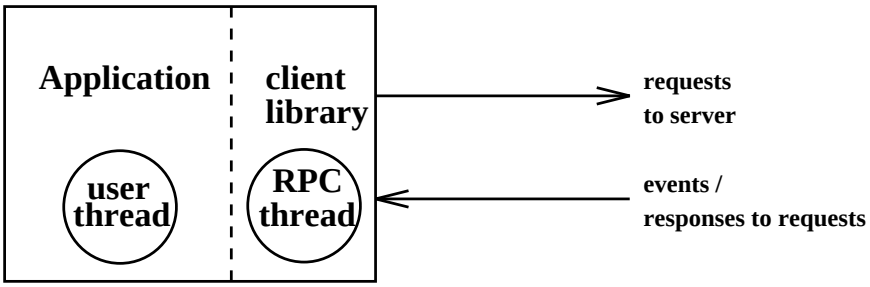

Figure 8: Client application internal structure.

DBMS, which may contain event notifications, or other messages from the database server such as callbacks delivering part of the answer to a query.

If events are to be processed by the application in an interrupt-driven mode, where event handlers are registered using the `a_RegisterForEvent` function, then the RPC thread executes the event handlers. If an interrupt-driven application monitors events and performs no other functions, then the user thread can simply sleep forever. If the application is structured so that the user thread polls for events, then the user thread must process the events. The only function of the RPC thread in this case is to enqueue event notifications in a volatile queue in the client's memory. These event notifications are dequeued and processed later by the user thread.

In interrupt-driven applications where the user thread is performing work concurrently with processing of event notifications by the RPC thread, the application programmer must use semaphores or some other synchronization mechanism to protect any data structures shared between the two threads.

If a client is implemented so that it polls for events, when it checks to see if an event has arrived, it does not send a message to the server. It can tell if an event has arrived by looking at its local volatile queue. This helps reduce the load on the server and cut down the number of network messages.

8 Authentication and Authorization

The Ariel system uses the Unix password mechanism for authenticating users. In order for a user to register for or raise events, they must have been granted authorization to do so. Each event has an owner. The owner of the event can modify the authorizations for register and raise access for an event.

For purposes of authentication and authorization, a user is described uniquely by:

userid@host.domain

where **userid** is a Unix userid, **host** is a machine name, and **domain** is a single administrative domain on the network. E.g., for the user

hanson@eel.cis.ufl.edu

The userid is hanson, the host is eel, and cis.ufl.edu is the domain.

For convenience, access can be granted to raise and register for an event to any of:

user@host.domain
user@*.domain
user@hostlist

The notation **user@*.domain** means the the user **user** at any host in **domain**. The notation **user@hostlist** means the user **user** at any host in **hostlist**. A hostlist can be defined separately as a list of hostnames.

An example use of the "user@*.domain" notation is:

```
define event sensitiveEvent(sensitiveInfo=c50)
grant raise access to all
grant register access to "hanson@*.cis.ufl.edu"
```

A single userid can be used to identify a user, and if used, such a userid is equivalent to "userid@*.current-domain" – e.g., "hanson" is equivalent to "hanson@*.cis.ufl.edu" if "cis.ufl.edu" is the current domain. Similarly, if a single hostname is used where "host.domain" is expected, the hostname is equivalent to "host.current-domain". E.g., "hanson@eel" would be equivalent to "hanson@eel.cis.ufl.edu". Identification of a user or host can be given as either an unquoted identifier (if it is a legal identifier) or as a string enclosed in double quotes.

Hostlists are useful for specifying a limited set of machines for which particular types of access are to be granted. An important use of hostlists is in systems where security in an entire domain can't be guaranteed, but a subset of the machines in the domain are known to be under firm administrative control. In such a situation, a hostlist can be created, and raise and register access can be granted to a user on any machine on the hostlist. An example of this is as follows:

```
define hostlist OURHOSTS as eel, mahimahi, carp
define event sensitiveEvent(sensitiveInfo=c50)
grant raise access to all
grant register access to "hanson@OURHOSTS"
```

An identifier that appears anywhere a hostlist can be used is first checked to see if it is a legal hostlist. If it is, it is processed as a hostlist. Otherwise, it is assumed to be a single hostname.

In general, if the "user@*.domain" or "user@hostlist" features are to be used, application developers and the DBA must make sure that they have administrative control over the whole domain or the listed hosts. For example, if there are many different "root" users on different machines, and not all these machines use the same password file, then the user "hanson" on one machine may not be the same as "hanson" on another machine. A rogue "root" user could create a user hanson on their machine solely for the purpose of being able to monitor sensitiveEvent. The "user@*.domain" feature can be disabled by the database administrator if administrative control over the whole domain cannot be ensured.

Ariel also supports user groups. A user group is a list of expressions of one of the forms:

user@host.domain
user@*.domain
user@hostlist

A user group can be defined as follows:

```
define user group <group-name> as <user-expression-list>>
```

For example:

```
define user group ariel as hanson, "ke@OURHOSTS", cx, icchen
```

Register and raise access for an event can be granted to a user group by referring to the user name in the appropriate "grant access" clause. An example of using a user group is:

```
define event sensitiveEvent(sensitiveInfo=c50)
grant raise access to all
grant register access to "msh@OURHOSTS", ariel
```

Identifiers that appear on an authorization list in a "grant access" clause are first checked to see if they are the name of an existing group. If they are, they are treated as group identifiers. Otherwise, they are treated as userids.

Normally in Ariel, the action of a rule runs as part of the same transaction that triggered the rule, and on behalf of the user who's transaction triggered the rule. This is normally adequate, but there are situations where the action of a rule needs more privileges than the user has. For example, consider a table AuditLog that has read and write access only for the user "auditor." The purpose of AuditLog is to store records of suspicious transactions. Triggers are used to identify such transactions, and their actions must append records to AuditLog. Clearly, the user submitting a suspicious transaction does not have write privilege for the AuditLog. For this reason, an Ariel rule can be made a "set userid" (setuid) rule, so that the rule action will run on behalf of the rule owner, not the user who triggered the rule. While the rule action is run, the effective userid (from Ariel's perspective) is temporarily switched to the userid of the rule owner. This setuid mechanism is analogous to the setuid capability for executable programs in UNIX [Gla93]. This setuid feature can also allow a rule to raise an event for which the current user does not have raise privilege.

To significantly improve security beyond what is described here, so that vulnerability to problems like rogue "root" users creating their own machine names and usernames is eliminated, an additional mechanism is required. These vulnerabilities can be handled using an encryption-based authentication service such as Kerberos [SNS88, Koh88, NS78]. An interesting topic for future work would be to study how a client-server database system with an event mechanism like the one described here could be made more secure using Kerberos or a Kerberos-like authentication service.

9 Recovery Issues

If a client application fails and then comes back up, then when the application restarts, it must reconnect to its recoverable queues if it is expecting to receive any replies or event signals via the queues. If the server computer system or a database or QM server process fails and comes back up, then the normal recovery mechanism will restore all the queues to a transaction-consistent state. Once the queues have been recovered, applications can start

taking replies and events out of them. If the network connecting the client and the server fails and then is restored to service, or the DC or entire server system fails and then comes back, the client application does not normally need to reconnect to its durable queues. If it is no longer connected, then it must connect using the normal mechanism. All replies and events sent via the durable queues remain transaction-consistent even if the network goes down temporarily. Replies and events will not be lost, nor will any events generated in deferred mode be processed.

If a server process crashes, the DC will notice this and restart the process. If the DC crashes, a daemon process whose only purpose is to monitor the DC will discover this and restart the DC. The new DC will locate the running server processes to initialize itself, ensure that the correct number of servers of each type are running, and then make itself available to service client requests.

10 Summary and Conclusion

The techniques described in this paper can be employed to build a reliable event mechanism for a distributed client-server transaction processing environment. This mechanism gives application programmers a convenient and reliable way to write programs that can be signaled when an event is raised by a rule action. We believe that facilities that support interaction between client applications and active database systems are extremely important, because they are crucial to allow active database rule firings to effectively interact with the real world. Active database/client interaction has not been studied by the active database research community in proportion to its practical significance.

An extended recoverable queue architecture has been developed to support the event mechanism presented. Recoverable queues provide the infrastructure needed to ensure that event signals from the server to client applications will never be lost, nor processed prior to the commitment of the transaction that generated the event. In contrast, event mechanisms in commercial active databases products, while very useful, are currently subject to failures where event notifications can be lost or processed prior to commit of the notifying transaction. In addition to recoverable event flow, traditional request/reply flow through recoverable queues is also supported.

The facilities described in this paper are currently under construction. The DC Server and Queue Manager have been implemented and tested on their own but not integrated with the rest of the system yet. An RPC interface has been created for the Ariel executable so it can serve both the role of Direct Ariel and Queued Ariel. The role of an Ariel process is determined by the setting of a flag when the executable is started up. The client library is currently being implemented. In late fall 1994 and spring 1995, the pieces of the system will be integrated.

We believe that the facilities described in this paper would be a useful addition to commercial on-line transaction processing products. New functionality is required both in the server (for client event registration and event notification) and in the client (for registering for and processing events). A nice property of the event mechanism proposed is that it is independent of the active database rule system implemented by the DBMS. It is conceivable that a standard for recoverable client-server programming in the presence of event notifications

could be developed, independent of the precise type of active rule mechanism implemented by specific database products. This could help make it possible to develop client application development tools that support recoverable event processing in conjunction with many different database products. This would provide an added dimension of interoperability in a heterogeneous database environment.

References

[ASK92] ASK Computer Systems. *INGRES/SQL Reference Manual*, 1992. Version 6.4, pages 6-67 to 6-79.

[ATT94] ATT Global Information Solutions. *TOP END Recoverable Transaction Queuing*, April 1994. DTP #93-386.

[Ber91] Philip A. Bernstein. DECdta - Digital's distributed transaction processing architecture. *Digital Technical Journal*, 3(1):10–17, Winter 1991.

[BHM90] Philip A. Bernstein, Meichun Hsu, and Bruce Mann. Implementing recoverable requests using queues. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 112–122, May 1990.

[Das93] Roxana K. Dastur. Using recoverable queues for reliable event communication between applications and active databases. Master's thesis, University of Florida, December 1993.

[Das94] Roxana K. Dastur. Correspondence regarding borland interbase. (personal communication), September 1994.

[Gla93] Graham Glass. *UNIX for Programmers and Users: A Complete Guide*. Prentice-Hall, 1993.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Han92] Eric N. Hanson. Rule condition testing and action execution in Ariel. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 49–58, June 1992.

[HCKW90] Eric N. Hanson, Moez Chaabouni, Chang-ho Kim, and Yu-wang Wang. A predicate matching algorithm for database rule systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 271–280, May 1990.

[Koh88] John T. Kohl. The evolution of the kerberos authentication service. In *Spring 1991 EurOpen Conference*, December 1988.

[MD89] Dennis R. McCarthy and Umeshwar Dayal. The architecture of an active data base management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 215–224, June 1989.

[NS78] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[Ram93] Vijay Ramaswamy. Data communication in the ariel client/server active dbms. Master's thesis, University of Florida, November 1993.

[RCS89] Joel E. Richardson, Michael J. Carey, and Daniel T. Schuh. The design of the E programming language. Technical report, University of Wisconsin, 1989. To appear, *ACM TOPLAS*.

[SNS88] Jennifer G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Usenix Conference Proceeedings*, December 1988.

[SR86] M. Stonebraker and L. Rowe. The design of POSTGRES. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1986.

A Client Application Programming Interface

In addition to the `a_RegisterForEvent` procedure described earlier, other functions are provided for a client application to use to interact with the event mechanism, as well as submit database commands and receive replies.

Most client library functions return an integer error code. If the code returned has the value a_NOERROR, which is zero, then the function executed successfully. If a non-zero code is returned, then an error may have occurred, and the global variable a_Code is set. The variable a_Code is undefined if the error code returned by the function is zero. A function that does not return an error code always sets a_Code. The possible values of a_Code are defined in the source file "errcodes.h" in the client library. The client library functions include:

a_Event* a_GetEvent(char* eventName, long timeout);

This function lets the application poll for an event with the given name. If eventName is NULL, then the next event among all the polling-type events registered for so far is returned. If the timeout is zero it will return a pointer to a descriptor for the next event if one is in the local event queue, and return NULL otherwise. If the timeout (unit = ms) is greater than zero, then if the next event is available anytime prior to timeout, a pointer to a descriptor for the next event is returned. If timeout occurs, a null pointer is returned. A negative timeout value causes the caller to block indefinitely until an event is available. It is an error to call a_GetEvent for an eventName that has an event handler function defined for it.

If the notification mode for the event registration is direct, the client will poll a local queue where notifications from the server are stored as they arrive. If the notification mode

is queued, the client will normally poll it's local queue looking for "queued event ready" messages from the QM. If a "queued event ready" message is in the local queue, the client will attempt to receive the message from the QM. If there has been no successful communication between the client and the QM for more than T time units (default T=5 minutes) then the client will send a "get new messages" request to the QM. This message will be repeated every T time units until communication with the QM is reestablished.

When the computer system on which the servers is running crashes and is restarted, and the entire Ariel system (DC, QM processes, and Ariel processes) is started, a designated QM sends "QM is alive" messages to all registered clients that have queued messages waiting for them. This is done once with no timeout and retry, and the clients do not acknowledge the message. This allows the clients to request new messages from the QM as soon as possible, without waiting for their own timeout to cause them to poll the QM.

The intent of this is to avoid unnecessarily frequent messages from the client to the QM when a client is using polling with a short local timeout frequency. The "get new messages" request to the QM is necessary to recover from communication failures. The "QM is alive" messages sent after restart of the QM are to allow clients to get pending messages sooner. The "QM is alive" messages are not necessary for correctness because the clients will time out eventually and poll the QM.

The body of a_GetEvent, which is always run in the user thread, is a critical section since it accesses the local event queue that is also updated by the RPC thread. A semaphore on the local event queue protects it from improper concurrent access.

char* a_GetEventName(a_Event*);

Get the name of an event given its handle.

int a_SetEventHandler(char* eventName, int handlerFunc(a_Event*));

This function lets the application set a handler for an event with name eventName. If a handler has been defined, then as soon as the notification arrives, the handler will be called. The handler function must take a single argument which is a pointer to an a_Event. If the application wishes, it can destroy the reference to a handler for an event using this function:

int a_UnsetEventHandler(char* eventName)

The application may then continue to get events for this eventName by polling with a_GetEvent.

The following functions are used to extract information from the event handle. There is always a *set* of one or more tuples associated with the occurrence of an event that has one or more parameters. There is an implicit cursor that initially is over the first tuple associated with the event. An exception to this is for events that have no parameters, in which case there are no tuples associated with an event occurrence.

long int a_getIntParam(a_Event* handle, char* paramName);
double a_getFloatParam(a_Event* handle, char* paramName);
void a_getCharParam(a_Event* handle, char* paramName, char *string);
void a_getParam(a_Event* handle, char* paramName, void *buffer);
void a_nextEventTuple(a_Event* handle);

int a_endOfEventTuples(a_Event* handle);

The first two functions above will return a parameter value for a given parameter name from the event. The third, getCharParam, extracts a character string from the event, and will store the result in the location passed as a third argument. The a_getParam function allows a parameter value of any type to be retrieved into an untyped buffer. Integers are represented as 4-byte quantities, floats as 8-byte double-precision quantities, and strings as null-terminated sequences of characters. The a_nextEventTuple function advances the cursor associated with the event so that it is over the next event tuple if there is one. If the cursor is not over an event tuple then it is an error to try to get a value from the current event tuple or advance the current event tuple. The a_endOfEventTuples function returns TRUE if the application has advanced past the last event tuple, and FALSE otherwise.

It is possible for an application to receive unanticipated events, e.g. if it registers for all events by specifying "*" as the event name when registering. To handle these unanticipated events, the application needs to be able to get information about the event, such as its name, the number of parameters it has, and the names and types of the parameters. The a_GetEventName function lets a program extract the name of any event. The following functions allow getting information about the parameters of an event:

int a_getNumParams(a_Event* handle);

Get the number of parameters of an event.

char* a_getParamName(a_Event* handle, int pnum);

Get the name of the parameter numbered pnum. Parameters are numbered 0 through $n-1$ when there are n parameters.

int a_getParamType(a_Event* handle, int pnum);

Get the type of a parameter (one of a_INT, a_FLOAT, a_CHAR) given its number.

int a_getParamTypeByName(a_Event* handle, char* pname);

Get the type of a parameter (one of a_INT, a_FLOAT, a_CHAR) given its name.

int a_DeleteEvent(a_Event* handle);

This function will free the memory of the Event.

The same functions are used for event processing by the client regardless of whether the events are sent directly, or sent via a recoverable queue. This allows applications to treat the event system as an abstraction and not worry about the details of event delivery. Also, the status of an event can be modified from immediate to deferred and it will not affect the application.

int a_ExecQUEL(char* queryString);

Send any query language string in Ariel's version of POSTQUEL to a Direct Ariel server. The names of bound variables can appear in queryString using a :ID notation to be described later. Binding is done with the a_Bind function. In this case, The queryString will be pre-processed and variables will be bound to the :ID values before the server carries out the string. The a_Code is set to indicate success or failure. No other information is available about the execution of the command.

a_Cursor* a_openCursorOnReply(a_Reply*);

This function opens a read-only cursor on the reply returned by a queued RETRIEVE command.

a_Cursor* a_openCursor(char *queryString)

This function supports opening a cursor on the answer to a RETRIEVE command. The query string may contain bound variables identified by :ID. Cursors are read-only.

int a_Bind(char* varName,void* value,int type);

Bind the memory location pointed to by value to the variable name varName. The type of the variable must be a_INT, a_FLOAT, or a_CHAR, which are defined as constants. All character strings must be null terminated. a_INT types must be 4-byte integers. a_FLOAT types must be C "double" values.

An example use of a_bind and a_OpenCursor is:

```
int anInt;
char* aName;
a_Cursor* c;
...
anInt = 17;
aName = "Bob";
a_Bind("X",&anInt,INT);
a_Bind("NAME",aName,CHAR);
c = a_openCursor("retrieve(emp.all) \
                 where emp.name = :NAME \
                 and emp.sal > :X");
```

Another example that retrieves a value into a bound variable is:

```
c = a_openCursor("retrieve(:X = emp.sal)\
                  where emp.name = :NAME");
```

int a_closeCursor(a_Cursor*);

Close a cursor. After this it can't be used.

int a_fetchNext(a_Cursor*);

Advance cursor to next record and get that record. The first call to this function gives the first tuple as the result. Falling off last record sets a_Code to a_EOF, and causes a_EOF to be returned. Values are retrieved from each record into variables bound using the :ID notation in the query on which the cursor is defined.

int a_fetchCurrent(a_Cursor*);

Get current record on which cursor is defined. Values from the record are fetched into bound variables. If cursor is not on a record, a_Code is set to a_NO_CURRENT_RECORD.

int a_BeginWork();

int a_CommitWork();

int a_RollbackWork();

These functions support standard transaction begin, commit and rollback. They set a_Code to indicate success or failure.

int a_Initialize();

This function must be called to initialize interaction with the Ariel system. It will create the RPC thread on the client side and send authentication information to the DC server. When the function returns, the flow of control on the next and subsequent statements is part of the user thread.

int a_StartEvents();

This function must be called after registering for an initial set of events. Any DIRECT events received before a_StartEvents is called are ignored. The a_StartEvents function processes any initial QUEUED interrupt-driven events and then makes the system ready to process any additional interrupt-driven events. The client application can register for additional events after calling a_StartEvents if desired.

Recoverable Queue Functions

int a_Connect(char* clientID);

Connect the client to the system. Associate the current client application with the name clientID. The client library maintains the queue ID numbers and other needed information in its own variables. The connection descriptor structure kept as a global variable by the client library contains the information described in section 5. This information allows the client to restore its state, and decide how and whether to process incoming replies and event notifications. Functions to be described later allow information to be obtained from the connection descriptor.

int a_Send(char* reqStr, a_reqID rid);

Send a request contained in the string reqStr and associate it with the given request ID rid. The rid must be obtained using the function **a_reqID newRID()**. The newRID function guarantees request ids won't be reused. The reqStr can contain any legal query language command or set of commands. The :ID notation may be used inside reqStr.

a_Reply* Receive(void* ckpt, int length);

Receive the next pending reply from the queue. This implicitly marks that last received reply as "completed." Some checkpoint data may be passed to the queue system. The ckpt parameter points to this data, and length indicates the number of bytes of checkpoint data to copy.

a_Reply* ReReceive();

Receive the last reply returned to this client by a Receive operation.

int a_Disconnect();

Disconnect the client from the system.

int a_CommitReceive(void* ckpt, int length);

Mark the last received queue entry as "completed." Some state checkpoint data may be passed to the queue system. The ckpt parameter points to this data, and length indicates the number of bytes of checkpoint data to copy.

a_Event* a_ReceiveQueuedEvent(char* ename);

This function is for internal use in the client library and is not to be called by the application (see instead a_GetEvent). If ename is not NULL, receive the next event with name ename queued by the QM in the recoverable event queue. Some checkpoint data is passed to the queue system if state handler functions are registered for event state (See a_SetEventStateFunc and a_SetEventStateLengthFunc below). The a_ReceiveQueuedEvent function returns NULL if no event with name ename is waiting for this client. The function also implicitly acknowledges the last event received unless it has been explicitly acknowledged with a_CommitReceiveQueuedEvent.

If ename is NULL when this function is called, then the QM is to return the next event ready for this client from among all those events the client is registered for in QUEUED mode.

a_Event* a_ReReceiveQueuedEvent();

Get the event returned by the last a_ReceiveQueuedEvent operation.

int a_CommitReceiveQueuedEvent();

Mark the queue entry associated with the last event received as "completed." Some checkpoint data can is passed to the queue system if state handler functions are registered (See a_SetEventStateFunc and a_SetEventStateLengthFunc below).

Other Functions

int a_SetEventStateFunc(a_ckpt EventState());

Register a function that returns a structure describing a client's event-related state information. The function registered must initialize this structure to point to the checkpoint data and contain the checkpoint data length. See the definition of the checkpoint abstract data type below for operations to set and get information from an a_ckpt structure.

The function registered will be called by the client library to get event state information whenever a_ReceiveQueuedEvent or a_CommitReceiveQueuedEvent are called. If an event state function is not registered, then no checkpoint data will be saved. The client must pass a single piece of event checkpoint data that satisfies the state checkpoint needs for *all* the different QUEUED events for which the client is registered.

int a_isReceiveCommitted(a_Reply *r);

Returns TRUE (1) if the receive operation associated with r has been marked as "completed," and FALSE (0) otherwise. Always sets a_Code.

int a_isEventCommitted(a_Event *e);

Returns TRUE (1) if the a_ReceiveQueuedEvent operation associated with e has been marked as "completed," and FALSE (0) otherwise. Always sets a_Code.

int a_isQueuedEvent(a_Event *e);

Returns TRUE (1) if e points to an event obtained from the recoverable event queue, and FALSE (0) otherwise. Always sets a_Code.

char* a_CharErrCode(int errorCode);

Returns a string with an English description of the error identified by errorCode.

int a_amConnected();

Returns TRUE (1) if client is currently connected, FALSE (0) otherwise.

Functions to Get Connection Information

These functions extract the information described in section 5 from the global connection descriptor kept by the client library. All these functions set a_Code.

a_reqID a_GetLastSentRID();

a_reqID a_GetLastReceivedRID();

a_reqID a_GetLastCommittedRID();

a_eventID a_GetLastReceivedEID();

a_eventID a_GetLastCommittedEID();

int a_DisconnectedLastSession();

True if client issued a Disconnect since last Connect operation.

a_ckpt a_GetReceiveCkpt();

Return checkpoint information associated with the last a_Receive or a_CommitReceive done by the client prior to last a_Connect operation.

a_ckpt a_GetEventCkpt();

Return checkpoint information associated with the last a_ReceiveQueuedEvent or a_CommitReceiveQue done by the client prior to last a_Connect operation.

Operations on Checkpoint abstract data type

The client library contains a typedef for a structure type a_ckpt. These operations can be performed on this type:

void a_SetCkptData(a_ckpt* ckpt, void* data);

Set checkpoint data.

void a_SetCkptLength(a_ckpt* ckpt, int length);

Set checkpoint length.

void* a_GetCkptData(a_ckpt* ckpt);

Return a pointer to the checkpoint data.

int a_GetCkptLength(a_ckpt* ckpt);

Return the length of the checkpoint data;

Example Application Program

```
/* This is an example event processing client application for the Ariel
   event system.  This client processes events using only the
   QUEUED mode with polling. */

#include<ariel.h>

/* handler functions and state functions for events */
void alertHandler(a_Event* e);
void* EventState();
int EventStateLength();

void InitSelfForEvents(a_ckpt*);

main() {
  a_Event* event;

  a_Initialize(); /* initialize interaction with Ariel */
  a_Connect("client1");/* Connect to persistent input & output queues
                          and event queue. */
  /* initialize self using state checkpoint information from queue system */
  InitSelfForEvents(a_GetEventCkpt());

  /* register for event */
  a_RegisterForEvent("alert",a_QUEUED,NULL);
  /* set state checkpoint functions */
  a_SetEventStateFunc("alert",AlertState);
  a_SetEventStateLengthFunc("alert",AlertStateLength);

  /* decide whether to do a re-receive of event "alert" */
  if(a_GetLastReceivedEID() != a_GetLastCommittedEID()) {
      /* process event again to make sure we do it at least once */
      event = a_ReReceiveQueuedEvent();
      alertHandler(event);
      a_commitReceiveQueuedEvent();
    };

  /* process polling-style events every 15 seconds */
  while(event = a_GetEvent("alert",15000)) {
    if (event == NULL)
      ; /* do nothing -- just wait again */
    else {  alertHandler(event);  /* process alert event */
            a_CommitReceiveEvent(); /* commits receive and checkpoints state */
    }
  }
}
```