An Architecture for Recoverable Interaction Between Applications and Active Databases (extended abstract)

Eric N. Hanson **Roxana Dastur** **Vijay Ramaswamy**

CIS Department
University of Florida
Gainseville, FL 32611
{hanson,vijay,rkd}@cis.ufl.edu

July 22, 1993

CIS-TR-93-024

1 Introduction

This paper describes the design of the event mechanism used to support reliable interaction between client applications 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. It is 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 not be 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 making event transmission 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. Recoverable queues and other techniques have been used in commercial products for many years to ensure recoverable message delivery in distributed systems [GR93]. The goal of this work is to provide an integrated framework for recoverable event communication between an active DBMS and applications.

At least one commercial product has an event mechanism similar in many ways to the one described here. This system is quite powerful and very useful for many applications. However, at least in its original form, that product is susceptible to the LDO and DDO problems. In addition, a mechanism whereby applications could be signalled by rules was conceived as part of the HiPAC project, and a main-memory prototype was done [MD89]. However, issues related to recoverable communication between applications and an active DBMS were not considered as part of that work.

Though extensions to Ariel described here can solve the LDO and DDO problems, it is not mandatory for applications to use the added mechanisms for solving these problems. If applications can tolerate lost or dirty event signals, they can get faster performance using a less reliable protocol.

2 Application Commands Supported

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 <eventname1> ( <parameters> )
[ mode [ is ] [ deferred | immediate ] ]
[ handler [ is ] [ all | raiser | any ]
[ grant access [ to ] [ all | raiser | <user-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 is "safer" though there will be some performance overhead.

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. The **grant access** clause allows an event to be defined with registration rights to all, to the raiser only, or to a set of users. Additional commands not shown here are provided to change the access rights associated with an event.

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

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

This format makes it possible to override the default coupling mode and handler specification when raising an event.

RegisterForEvent Signals the application runtime system that this application wishes to receive the specified event. It is only valid when issued by a running application. Since this command is interpreted by the application runtime system, it is a procedure rather than a POSTQUEL command.

```
int RegisterForEvent(char* eventName, void* eventHandlerProcedure)
```

The eventHandlerProcedure must be a procedure that accepts arguments of the same type as those specified in the definition of event <eventName>.

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 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, or *server classes*:

Direct Ariel to execute direct requests from clients,

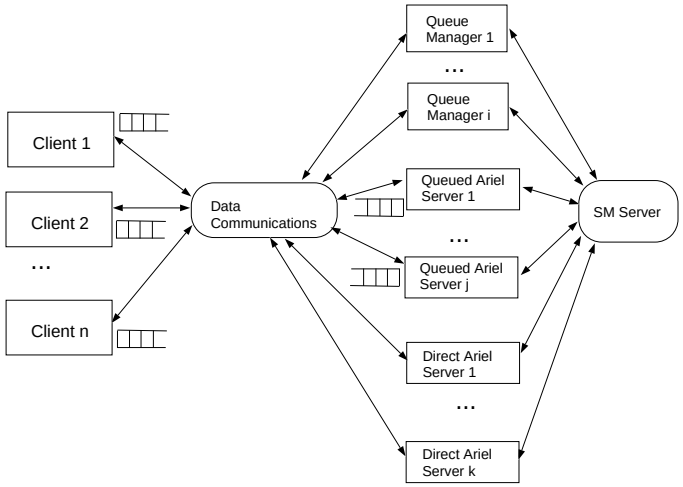

Figure 1: Ariel client-server architecture supporting event processing

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

Queue Manager to enqueue requests from clients in a recoverable queue to be processed later by a **Queued Ariel** server.

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 1. The standard form of interaction is direct transactions. Client programs can call the following procedures to control direct transactions:

BeginWork() starts a transaction,
CommitWork() commits a transaction,
RollbackWork() aborts a transaction.

When a client program calls BeginWork it establishes a *session* with a single Direct Ariel server process and that server becomes dedicated to the application until the application issues CommitWork or 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 RPC's return value contains the handle. Otherwise, if an Ariel is not free, the RPC's 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 DC is "a multi threaded server cum client." When it comes up it fires up n ariel_threads to communicate with n Ariel servers (a one to one mapping), a high priority dc_server thread on which the clients do RPC's, a high priority scheduler which schedules these threads and a sleep_destroyer thread.

When a client does an RPC on the DC, the high priority dc_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, or else it puts the request in a data structure called the WorkQueue and transmits a status code indicating this as the RPC return value. The client then must be prepared to receive an RPC later from the DC server.

RPC's 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 Ariel server process supports operations BeginWork, CommitWork, and RollbackWork as described before. In addition it supports the operation DoQUEL that takes a POSTQUEL command as an argument and sends back the results, such as the outcome of the command, any tuples retrieved etc. SQL-style cursor operations Open, Fetch, and Close are also supported by the Ariel server for direct transactions. DoQUEL and cursor operations are received via RPC directly from clients. This concludes the description of the features that support direct transactions.

4 Event Registration Service

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 durable event catalog, which is updated inside a transaction.

5 Recoverable Queues

Recoverable queues are implemented as a collection of queues, one queue per client, in the database. Access to these queues is limited to the Queue Manager and the Queued Ariel process. The Queue Manager is the real owner of these queues. It generates a persistent 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. For server-end management and administration, the Queue Manager has functions that enable the server to read requests in from the queue, store responses in the queue, delete transaction entries from the queue, and commit transactions. Reading and storing requests and responses can follow request-reply matching, or be asynchronous. This asynchronous mode of client notification allows the server to notify the client that an event occurred, and this notification need not be in response to a request from the client. This is different from the standard use of recoverable queues, where requests and replies are always matched.

The Queue Manager (QM) is set up as an RPC server, with client-end and server-end procedures as its interface. In addition, the QM also acts as a client to the client application itself in the following circumstances: it has to explicitly inform the client to receive a response, either as a reply to a request or as an event. In this way, it supports asynchronous processing by a client application.

As far as transaction boundaries are concerned, all **Connect** and **Send** requests are transaction consistent, since the QM writes to a persistent queue within these procedures. Queued Ariel processes call the server-end procedures within their own transactions. Isolated responses, which are messages from the

server back to a client that don't correspond to a request, but instead represent the raising of an event by a rule, carry an explicit **Commit** request for the queue manager. This makes the QM flag the non-existent requesting transaction in the queue as "done." The QM functions offer a low level abstraction for event processing, as well as a higher level request/reply communication protocol.

6 Queued Transactions

Queued transactions are provided for application environments where an immediate response to a request is not essential. A version of the Ariel server called a Queued Ariel reads an input request from a durable queue, processes the request, and enqueues a reply, all inside a transaction. The same type of durable queues used for request and response transmission for queued transactions are also used to provide recoverable transmission of events to application programs. An economy of mechanism is achieved since durable queues are used for both purposes. Many transaction processing systems already provide durable queues for requests and replies. With relatively little effort they could likely use those same queues to transmit events.

7 Event Processing

When an event is raised, the event catalog is consulted to determine which client(s) are to receive the event. If the event is declared as "immediate" then the Ariel server that recognized the event does a direct remote procedure call to the application. No recoverable queue is used. If the event is "deferred" then the Ariel server enqueues the event in the durable input queue of the application. This is done inside the same transaction that raised the event. If the raising transaction aborts, the event disappears from the queue due to rollback. The event cannot be read by the application until the raising transaction commits because the event is write-locked by the raising transaction prior to commit.

8 Client Application Development Facilities

In addition to the RegisterForEvent procedure described earlier, some other functions are provided for a client application to use to interact with the event mechanism. These include:

eventHandle* GetEvent(char* eventName, int timeout);

This function lets the application poll for an event with the given name. If the timeout is zero it will return a pointer to a descriptor for the next event if one is in the queue, and return null otherwise. If the timeout 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. It is an error to call GetEvent for an eventName that has an event handler function defined for it (see below).

void SetEventHandler(char* eventName, handlerType* handlerFunc);

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 event arrives, the handler will be called. The handler function must take a single argument which is a pointer to an eventHandle. If the application wishes, it can destroy the reference to a handler for an event using this function:

void UnsetEventHandler(char* eventName)

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

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.

9 Client Runtime Library

The functions described in the previous section are part of the client runtime library that is linked with each client when it is compiled. 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.

10 Recovery Issues

If a client application fails and then comes back up, then when the application restarts, it must reconnect to its durable event queue if it is expecting to receive any event signals via this queue. Reconnecting is done

via a variation of the normal mechanism used for durable queues [BHM90]. If the database system fails and comes back up, then the normal recovery mechanism will restore all the durable queues to a transaction-consistent state. Once the durable queues have been recovered, applications can start taking events out of them. If the network connecting the client and the server fails and then is restored to service, the client application does not need to do anything if it is still connected to its durable event queue. If it is no longer connected, then it must reconnect using the normal mechanism. All events sent via the durable queues remain transaction-consistent even if the network goes down temporarily. Events will not be lost, nor will uncommitted events be processed.

11 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. A variation of Bernstein's recoverable queue architecture [BHM90] has been developed to support this event mechanism. These 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.

References

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

[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 *Proc. 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.

[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.

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