Implementing Recoverable Requests Using Queues

Philip A Bernstein*
Meichun Hsu†
Bruce Mann‡

Digital Equipment Corp

Abstract

Transactions have been rigorously defined and extensively studied in the database and transaction processing literature, but little has been said about the handling of the *requests* for transaction execution In commercial TP systems, especially distributed ones, managing the flow of requests is often as important as executing the transactions themselves

This paper studies fault-tolerant protocols for managing the flow of transaction requests between clients that issue requests and servers that process them We discuss how to implement these protocols using transactions and *recoverable queuing systems* Queuing systems are used to move requests reliably between clients and servers The protocols use queuing systems to ensure that the server processes each request exactly once and that a client processes each reply at least once We treat request-reply protocols for single-transaction requests, for multi-transaction requests, and for requests that require interaction with the display after the request is submitted

*Address Digital Equipment Corp , One Kendall Square – Building 700, Cambridge, MA 02139, pbernstein@crl dec com

†Address Aiken Computation Lab , Harvard Univ , Cambridge, MA 02138, hsu@harvard harvard edu

‡Address Mail Stop ZKO1-3/J35, Digital Equipment Corp , 110 Spit Brook Road, Nashua, NH 03062-2642, mann@star dec com

1 Introduction

A transaction is a unit of work, performed on shared data, which preserves atomicity (i e , all-or-nothing), serializability, and durability Transactions have been rigorously defined and extensively studied in the database and transaction processing (TP) literature [Bernstein et al 87] By comparison, only a few articles explain how to handle *requests* for transaction execution [Gray 78, McGee 77] This emphasis in the literature does not reflect the relative importance of the problems In commercial TP systems, especially distributed ones, managing the flow of requests is often as important as executing the transactions themselves

This paper studies (1) protocols for reliably managing the flow of transaction requests, and (2) implementations of those protocols that use *recoverable queuing systems* A queue in a TP system serves as an intermediary between a *client* and *server* A client enters a request into a queue that is served by the desired server A server obtains the request by removing it from the queue Queuing facilities are embedded in many existing TP systems [DEC 88, McGee 77, Wipfler 87]

A queue manager (QM) is a type of database system It stores queue *elements*, which are usually uninterpreted by the QM (i e , the QM doesn't know that they represent requests) The QM may support content-based retrieval of the elements Operations on elements may be performed within a transaction, and must therefore be recoverable

A QM is also a type of communication system, since it moves messages between processes that run asynchronously It allows for *indirect* communication between processes, that is, these processes do not directly bind to one another to communicate, but instead, each binds to a queue This indirect communication can mask process failures, by allowing communicating processes to execute and fail independently For example, a process that enqueues a request can communicate with one that executes the request even if they are not both operational simultaneously Indirect communication via queues can also mask com-

munication failures if a client enqueues its requests to a local queue, and periodically moves its local requests to the remote input queue of a server process, then the server appears to provide a reliable service to the client even if the client and server nodes are frequently partitioned by communication failures These features improve availability and efficiency

Queues facilitate *batch input* of requests Requests can be captured reliably in a queue, and processed later in a batch Queues also provide *load sharing* Since many processes can dequeue requests from a single queue, this automatically shares the workload among these processes Moreover, queues provide a buffer that mitigates the effects of bursts of requests

The purpose of this paper is to present fault-tolerant protocols for managing the flow of transaction requests, and to show how to implement these protocols using queuing systems The paper is organized as follows Section 2 presents an overview of the main idea of the paper how to use queues to process requests in a fault-tolerant manner Section 3 defines the client's model of fault-tolerant request-reply processing Section 4 describes a queue abstraction, which is used in Section 5 in a system model that implements the client's model Section 6 extends the model to requests that execute as a sequence of transactions Section 7 extends it to support cancellation of requests Section 8 treats interactive requests, in which the client provides input to the request while the request is executing Section 9 discusses related work, including the queuing facilities offered in commercial systems Section 10 discusses implementation issues Section 11 is the conclusion

2 Reliable Request-Processing

A *request* is a data structure (e g , a record) that describes some work that the system should perform Typically, a user interacts with *a client* program to prepare a request (e g , by filling in a form), and then the client sends the request to a *server* for processing The server *processes* the request by executing one or more transactions After it finishes processing the request, the server may send a *reply* back to the client, which delivers it to the user

The client typically doesn't need to access shared updatable data Therefore, in a distributed system, it often runs in a "front-end" process close to the user's display, where processor cycles are inexpensive and interactive feedback is easy to attain The server typically needs access to a shared updatable database, often a large one Therefore, it executes on a large "back-end" system, which is accessible to many clients Since the client and server reside in different processes, often on different processors, interprocess communication primitives are needed to exchange requests and replies

If ordinary messages are used to move requests and replies between clients and servers, then an untimely system failure (e g , when the request or reply is in transit) may cause either the request or the reply to be lost The client may be unable to determine whether the request or reply has been lost Moreover, since many requests are not idempotent, the client cannot resubmit the request unless it's sure that the original request hasn't been and won't be processed But determining the state of the request is difficult If the client and server don't carefully maintain this state on stable storage, it may be impossible

In principle, we can solve the problem of untimely failures by having the client execute the sequence {send a request, receive the reply, process the reply} within a transaction In practice, there are two problems some clients execute on systems that don't support a transaction mechanism or don't support one that's compatible with the server's mechanism, and processing the reply may be slow, which creates contention for resources (e g , locks) that the server must hold until the transaction commits We can solve the latter problem by only executing {send a request, receive the reply} within a transaction But if the client fails after receiving the reply and before processing it, the reply may be lost

We can improve matters by executing the client's sequence as three transactions and using two *recoverable queues* as intermediaries between the client and server – a client queue and a server queue (We use the word *queue* for compatibility with commercial TP terminology It's a misnomer, since these queues are often not FIFO) To send the request to a server, the client executes a transaction that enters the request into the appropriate server's queue To process a request, the server starts a transaction, removes an element from its queue, "executes the request," enters a reply into the client's queue (if a reply is desired), and commits To process a reply, the client starts a transaction, dequeues the reply, and after it has finished processing the reply (e g , the user acknowledges having seen the reply on the display), commits Note that the reply processor (e g , user) is just another "resource manager" that participates in the transaction

Each enqueue and dequeue operation executes within a transaction If the invoking transaction aborts, the operation is undone This enables the system to cope with failures of clients and servers If the client fails while submitting a request, the request is forgotten, so there is no danger that a partial request is submitted If the server fails while executing a request, all of its database updates are undone and the request is returned to the server's queue for reprocessing If the client fails while processing a reply, the reply is returned to the client's queue, so the client can reprocess the reply after recovery Overall, this approach avoids the resource contention of the one-transaction approach, but still requires a client to use transactions

We can eliminate the client's transactions altogether by regarding it as a fault-tolerant sequential program That is, it sends (i e , enqueues) a request and receives (i e , dequeues) a reply outside of a transaction At recovery time it determines the last non-idempotent operation it executed before the failure and reconstructs its internal state that was current when it issued that operation Since the client wants the system to execute its request exactly once, the operation "send-the-request" is not idempotent So if the client loses connectivity with the system, when communication is restored the client must determine whether the system received the last request that it sent A queue can help here, since the client can tell whether the request was enqueued It can also help by making "receive-the-reply" idempotent, by retaining the reply until the client says to delete it (i e , even after the client has dequeued it) A queue can even help with checkpointing If the client's state is small, the client can piggyback its state with its enqueue and dequeue operations, the queue manager can log this client state along with the queue operation, thereby performing the client's checkpoint This, in essence, is the solution developed in this paper

Notice that the client accesses queues outside of a transaction, while the server accesses queues within transactions In this sense, the queue is a gateway between the non-transaction world of front-ends and the transactional world of back-ends

3 Client Model

A *non-interactive request* is a request which, after being submitted by a client, is processed without further input from the client The Client Model consists of a *client* and a *system* From the client's viewpoint, the client submits a request to the system, the system processes the request and sends a reply, and the client receives the reply and processes it [1]

Before sending any requests, the client must *connect* to the system While connected, the client offers requests, one-at-a-time The system processes each request and sends a reply to that request before it processes the next request Each request implicitly acknowledges that the client has received and processed the reply to the client's previous request When the client has no more requests to offer, it *disconnects* from the system

The client attaches a *request-id (rid)* to each request In response to the Connect operation, the system returns two rids — the rid of the last request it received from the client, and the rid of the request that corresponds to the last reply it sent to the client After recovering from a failure, the client can use these rids to resynchronize with the system, that is, it can determine whether to (possibly again) send the next request or (possibly again) receive the next reply

[1] Some applications don't need a reply They are not treated here, but our model could be modified to handle them

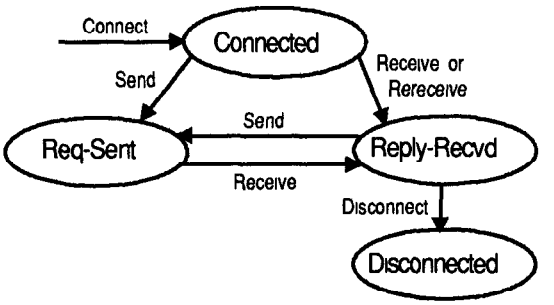

Figure 1 Client's State Transition Diagram for a Non-Interactive Request

The Client Model is summarized in fig 1 by a state transition diagram, and in fig 2 by a client program The client sees a non-interactive request pass through two states Req-Sent (the client has sent a request), and Reply-Recvd (the client has received the reply, and may enter a new request) A Connect operation puts the client into a Connected state, from which it branches to Req-Sent or Reply-Recvd, depending on the rids returned by the Connect A Disconnect operation causes the client to break its connection with the system

The client's interface to the system consists of five operations

- *s-rid, r-rid, ckpt = Connect(client-id)* Connect the client to the system If the client has not executed a Disconnect since the last time it executed Connect, then Connect returns the rids associated with the last Send and Receive it executed (*s-rid* and *r-rid* respectively) It also returns the value of the *ckpt* parameter of the last Receive operation executed by the client
- *Disconnect(client-id)* Disconnect the client from the system
- *Send(r, s-rid)* Send a request r with rid *s-rid* to the system When this returns, the request and rid have been stably stored
- p = *Receive(ckpt)* Return the next reply p for that client The client may put some checkpoint information in the *ckpt* parameter (see the Connect operation above)
- p = *Rereceive()* Return the reply that was returned by the last Receive executed by the client

If the client conforms to the above model, then the system makes the following guarantees, despite failures and recoveries

- *Request-Reply Matching* — Each reply is the reply for the corresponding request

```
Client(user-id)  {
  s-rid, r-rid, ckpt = Connect(client-id),
  if ( not(s-rid = NIL) and
     not(s-rid = r-rid) )
   { reply = Receive(ckpt),
     process the reply },
  if ( not(s-rid = NIL) and
      (s-rid = r-rid) and
      (client didn't process reply) )
    { reply = Rereceive(),
      process the reply };
  While (there's work to do)
   { construct request and s-rid,
     Send(request, s-rid),
     /* construct ckpt parameter */
     reply = Receive(ckpt);
     process reply },
  Disconnect(client-id)          }
```

Figure 2 The Client Program for a Non-Interactive Request

- *Exactly-Once Request-Processing* — The system processes each request exactly once The system may process the request by unsuccessfully attempting to execute the request, and then returning a reply that indicates that fact, the reply is a promise that it will not attempt to execute the request any more
- *At-Least-Once Reply-Processing* — The client processes each reply at least once

The value of Request-Reply Matching and Exactly-Once Request-Processing is obvious To examine At-Least-Once Reply-Processing, consider the client's behavior at connect time If the client receives a non-NIL s-rid and a non-NIL r-rid and they match, then it must determine whether it should process the reply again There are several cases to consider

First, suppose "process reply" is an atomic operation and the client can detect whether it processed the reply Then the client processes the reply exactly once Exactly-once is important if reply processing is not idempotent, e g , if it involves printing a ticket or dispensing cash This is easy if the output device is *testable*, meaning that the client can read the state of the device, such as the next ticket to be printed [Pausch 88] The client reads the state (e g , the ticket number) *before* receiving the reply, and uses that state as part of the ckpt parameter in the Receive If the client or client-system communication fails, then when the client reconnects, it can compare the device's state with the ckpt value returned by Connect If they don't match, then it knows the reply was already processed

Now suppose that processing the reply is not atomic and/or the client cannot detect whether it already processed the reply In this case, if we want to ensure that each reply is processed at-least-once, then the client should assume that it didn't process the reply, and it should process the reply again If reply processing is idempotent, then this is acceptable For example, if the client is communicating with a display, and the user supplies a unique id for each request, then the client can display the id with each reply and the user can detect and ignore duplicate replies If this is not acceptable (e g , if reply processing is not idempotent), then other techniques are needed (e g , ask someone to find out if the reply was already processed, or regard lost replies as acceptable)

4 Queue Operations

We will implement the client model of the previous section using a queue abstraction In this section, we describe the abstraction, for the most part without motivation We limit our attention to features that are needed for the Client Model in Section 3 and other models described later in the paper (The abstraction would be incomplete for a full-function queue manager product) We use the abstraction to implement the Client Model in Section 5

Most features of the abstraction are in existing products However, we believe the persistent registration and operation tag features described here are new

Queue operations are all-or-nothing and are serializable with respect to each other If invoked from within a transaction, a queue operation obeys transaction semantics

4.1 Objects

A *queue* is a stable memory area that contains *elements* (Some systems also offer volatile queues, which we do not discuss here) Each element has a unique *element identifier (eid)*

A *queue repository* is a set of queues Repositories are disjoint Each queue has a name that is unique within its repository Each repository has a system(or network)-wide unique name

Data definition operations are needed to create, destroy, and modify a queue or repository, and to start and stop queues These operations are straightforward and are not discussed here

4.2 Data Manipulation Operations

Programs access queues using the following data manipulation operations (see fig 3)

The *Enqueue* operation creates an element and stores it in a queue If Enqueue is issued within a

stroys all registration information about a previously registered registrant

The QM allows each registrant to supply a registrant-defined tag with each Enqueue or Dequeue operation (We will use tags to implement rids and ckpts in the Client Model) The QM maintains a stable copy of eid, tag, and element contents associated with the last operation performed by each registrant as a part of the registration information

If a client that calls Register is already registered (e g , it is recovering from a failure), then the QM returns the tag and eid associated with the operation most recently executed by this registrant A registrant may Read the element identified by this eid, even if the last operation was a Dequeue, or if it was an Enqueue and the enqueued element was dequeued by another registrant A "stable-flag" parameter indicates whether the client wants the QM to maintain the most recently executed operation

Queue operations can be used to support a variety of client models, not just the model of Section 3 Different models will need to tag different client operations, which in turn requires tagging different queue operations Therefore, we believe it's desirable to allow tagging of all data manipulation operations, to support the flexible construction of client models In this case, the QM must maintain the *type* of the last operation executed by each registrant, in addition to the eid of the element operated upon and the last operation's tag However, in this paper we won't discuss applications of this generalization to other client models

5 System Model

We define a System Model for processing non-interactive requests that supports the Client Model of Section 3 In this model, each request executes as a single transaction

The system consists of a clerk program, a request queue, a reply queue, and a server process (see fig 4 and 5) For each request, the server dequeues the request, processes it, and enqueues the reply, all within a transaction The client's operations are translated into queue operations This translation is performed by a *clerk program* that is local to the client (i e , it is a runtime library) The clerk communicates with the QM(s) that manage the request and reply queues If the QM is remote from the client, then we assume that the clerk invokes QM operations using remote procedure call [Birrell and Nelson 84] The clerk behaves as follows

- *Connect(client-id)* – Register the client with the request and reply queues Each Register returns a tag These tags are returned to the client as s-rid, r-rid, and ckpt
- *Disconnect* – Deregister with the request and reply queues
- *Send(r, rid)* – Enqueue request r into the request queue, tagging the Enqueue with *rid*
- *Receive(ckpt)* – Dequeue the next reply from the reply queue and return it, tagging the Dequeue with *ckpt* and the rid of the previous Send
- *Rereceive()* – Read the reply most recently dequeued by the client and return it

When Send returns, the client knows that the request was stably stored If we are willing to forego this knowledge, then Send need not receive an acknowledgement that the Enqueue executed That is, it can invoke Enqueue using a one-way message, instead of a remote procedure call If the Enqueue fails, the client will time out waiting for its Receive to dequeue the reply and can determine what happened when it reconnects This saves a message from the QM to the client in the common case that the reply arrives within the client's timeout period Alternatively, we can merge Send and Receive into a single Transceive operation, which blocks the client until the reply arrives

This algorithm is correct in that it preserves Request-Reply Matching, Exactly-Once Request Processing, and At-Least-Once Reply-Processing, even in the presence of client, server, and communication failures Request-Reply Matching follows from the fact that the server replies to each request before dequeuing the next request, and that the client receives the reply to each request before entering the next request In the case of client, server, or communication failures, the client's connect-time resynchronization (i e , lines 2-11 of the client program in fig 2) ensures the property is re-established at recovery time

Exactly-Once Request-Processing follows from the fact that the request is processed within a transaction If the transaction aborts, the Dequeue of the request is undone, thereby returning the request to the request queue To avoid cyclic restart of the request (i e , to guarantee termination), the server should use the error queue facility of the Dequeue operation (see Section 4 2)

At-Least-Once Reply-Processing follows directly from the Client Model semantics of the client-to-system operations and the client's connect-time resynchronization activity (in fig 2)

There are many useful ways to extend and modify the simple Client Model Each extension affects the System Model, and the underlying queue abstraction that supports it

One extension is to allow multiple client processes We can accommodate multiple clients by giving each client a private reply queue, and passing that queue's name with the request, so the server knows where to Enqueue the reply

Another extension is to allow concurrency within a client This amounts to identifying a client by both a

```
h,t,e = Register(qname,client,stable-flag)
/* Registers client with queue qname, and returns
handle h, tag t, and eid e, where t is the tag of the
client's last Enqueue or Dequeue, and e the eid of the
element enqueued or dequeued (t and e may be NIL)
stable-flag indicates whether a tag should be main-
tained */

Deregister(h, client)
/* Deregisters client from the queue identified by han-
dle h */

e = Enqueue(h, element, t)
/*Enqueues element into the queue identified by h,
tags the operation with t and returns eid e */

element = Dequeue(h, t, eh)
/* Dequeues the next element from the queue iden-
tified by h, returns it and tags the operation with t
If the element is dequeued successively by n trans-
actions, each of which aborts, then the n-th abort
returns the element to the queue identified by eh */

element = Read(h, e)
/* Reads element with eid e from the queue identified
by h */
```

Figure 3 Operations on Queues

transaction, its effect becomes visible when the transaction commits If issued outside a transaction, its effect is visible before the operation returns to the caller

The *Dequeue* operation deletes an element from a given queue, and returns it to the caller If invoked from within a transaction, and if the transaction aborts, then the element is marked with an *abort code* and returned either to the given queue or to a separate error queue (specified by a parameter in the call) If an element is dequeued successively by n transactions, each of which aborts, then the *n-th* abort returns the element to an error queue (where n is an attribute of the queue)

The *Read* operation returns the contents of an element with a given eid from a given queue without modifying it

4.3 Persistent Registration

A QM can help its clients recover from failures by keeping persistent information about clients

The *Register* operation associates an authenticated, uniquely-named *registrant* with a named queue, and returns a handle Information about a registration is guaranteed to be stable when the Register operation completes The registration information is maintained until the registrant explicitly deregisters In particular, the failure of a registrant does *not* implicitly deregister it The *Deregister* operation de-

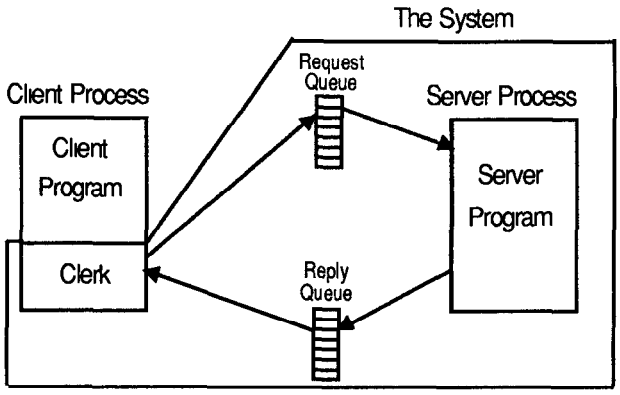

Figure 4 System Model for Non-Interactive Requests

```
Connect(user_id) {
    q_in, rid_tag, qe1 =
      Register(req_q,  client_id, TRUE),
    q_out, reply_tag, qe2 =
      Register(reply_q,client_id, TRUE),
    /* Variables q_in, q_out, qe1, qe2, and
      rid_tag are global to the client */
    return(rid_tag, reply_tag[rid-piece],
          reply_tag[ckpt-piece])        }

Disconnect(user_id) {
    Deregister(q_in,  client_id),
    Deregister(q_out, client_id) }

Send(request, rid) {
    rid_tag = rid,
    e = Enqueue(q_in, request, rid_tag) }

Receive(ckpt) {
    return(Dequeue(q_out,[rid_tag,ckpt],eh))}

Rereceive() {
    return(Read(q_out, qe2)) }

Server() {
  q_in, x,y=Register(req_q,  ap_id,FALSE);
  q_out,x,y=Register(reply_q,ap_id,FALSE),
  While (true) {
    start_transaction,
    request = Dequeue (q_in, NIL),
    process request and prepare reply;
    Enqueue (q_out, reply, NIL),
    commit_transaction }            }
```

Figure 5 Algorithm for Non-Interactive Requests Connect, Disconnect, Send, Receive, and Rereceive reside in the Clerk

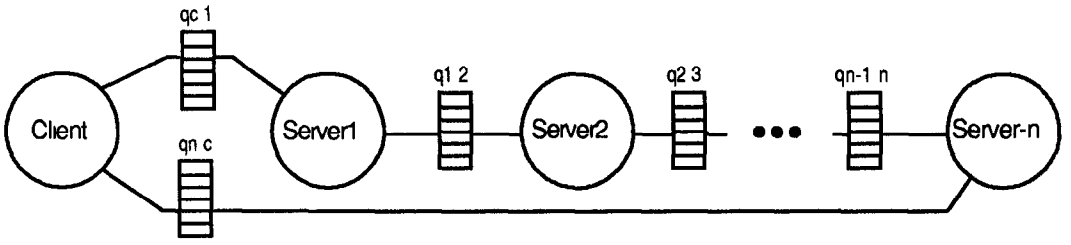

Figure 6 System Model for a Multi-transaction Request

client-id and a "thread"-id The system now maintains an array of [req-tag, reply-tag] pairs for the client, one for each thread-id The entire array is returned by a Connect operation To support this, the underlying QM needs a comparable facility in the Register operation

We discuss several other extensions to the Client Model in Sections 6-8 requests that execute as multiple transactions, the cancellation of in-flight requests, and requests whose execution must interact with the client Given space limitations, we'll only sketch the client model, system model, and queue management considerations for each extension

6 Multi-Transaction Requests

It is often convenient to execute a request using several transactions For example, a *funds transfer request* may be processed as three separate transactions debit source bank account, credit target bank account, and log the transfer with a clearinghouse This approach may be chosen to avoid executing one long transaction, which can lead to lock contention It may be required in a distributed system, if the nodes that process the request either are not all available at the same time or do not use the same transaction protocol (e g , two-phase commit) (In the latter case, a QM may need to support multiple transaction protocols) In principle, this multi-transaction implementation should be transparent to the client, who still sees the request as a single unit of work, as in the model of Section 3

We modify our model of Section 5 to accommodate multiple transactions that are performed *serially* to service a non-interactive request There is a sequence of server processes, which executes the sequence of transactions for the request (see fig 6) Each server registers with a different pair of queues for req_q and reply_q (e g , server1 uses qc 1 and q1 2, and server2 uses q1 2 and q2 3) The clerk and server algorithms are unchanged from fig 5

An application programmer (i e , one who writes server programs) cannot rely on local program variables to record the *state* of the request across multiple transactions, because the contents of such variables may be lost in a failure If any information must be passed across transaction boundaries, a server must store it either in a database or in the next request (i e , the request for the next transaction in the sequence)

The sequence of transactions that processes the request cannot be broken by a failure, since any failure will result in a transaction being aborted, thereby returning the request for that transaction to the transaction's input queue Thus, the argument that this multi-transaction implementation preserves Request-Reply Matching, Exactly-Once Request-Processing, and At-Least-Once Reply-Processing is the same as the single-transaction case

This method can be extended to include *concurrent* execution of multiple transactions servicing a user request The main issue is forking a request into multiple requests and rejoining the requests when the concurrent branches complete This can be handled by extending the QM with a trigger mechanism A trigger is set to send a request when all of the replies to earlier concurrent requests have been received

One disadvantage of multi-transaction requests is that the execution of requests is not serializable Only the execution of the component transactions is serializable That is, a transaction that executes for one request may execute in between two transactions that execute for another request

If all database systems accessed by the multi-transaction request use locking for concurrency control, then one can achieve serializability of request executions by coaxing the database systems into holding locks across transaction boundaries for all of the transactions that implement the multi-transaction request That is, each transaction's database locks are inherited by the next transaction in the sequence Of course, this may create lock contention, which is one reason why some requests are executed as multi-transaction requests in the first place But when lock contention is not an issue, serializability of requests is possible even if database systems won't hold locks across transactions the *application* can mimic database system locking by creating a persistent database of locks, setting the appropriate locks for each database object it accesses, and releasing all of these "application locks" just before the final transaction of the multi-transaction request commits Unfortunately, the performance of this approach will be

limited, due to the high overhead of setting locks and the coarseness of lock granularity

7 Request Cancellation

A client may send a request, and later decide to cancel that request To implement this in the System Model, we need a queue operation that deletes a specified element

The *Kill_element* operation takes an eid and tries to delete the given element from its queue If the element has not yet been dequeued, it is deleted If it was dequeued by a transaction that has not yet committed, the transaction is aborted and the element is deleted Kill_element returns an indication whether it deleted the element

The client operation is Cancel-last-request The clerk processes this operation by invoking the queue operation Kill_element on the eid of the last request The clerk should maintain this eid, which is returned by each Enqueue operation and by Register when the client recovers from a failure

With multi-transaction requests, the cancellation request fails once the first transaction in the sequence has committed Later cancellation can still be arranged by supporting compensating transactions and sagas [Garcia and Salem 87, Klein and Reuter 88] That is, one cancels the request by compensating for the committed transactions that executed on behalf of the request This can be done by executing the compensations as a serial multi-transaction request

8 Interactive Requests

8.1 Client Model

This section considers *interactive requests* In the client model of an interactive request, the executing request periodically sends *intermediate output* and asks the client to supply *intermediate input* This adds a new *Intermediate-I/O* state to the client's state transition diagram (see fig 7) As before, the client transitions to Req-Sent after sending a request The client then asks to receive an intermediate output message, thereby transitioning to Intermediate-I/O After receiving this output, the client sends intermediate input to the system, thereby transitioning from Intermediate-I/O back to Req-Sent The client may cycle between Req-Sent and Intermediate-I/O many times (in this over-simplified model, we assume the client knows how many times) Eventually, the client supplies all the intermediate input, the system finishes processing the request, and the client receives the reply

One could implement interactive transactions just like non-interactive ones, using one-way messages to exchange intermediate I/O between the client and system However, if an interactive transaction aborts, intermediate I/O may be lost (unlike requests, which are never lost due to an abort) We now describe ways to avoid losing intermediate I/O

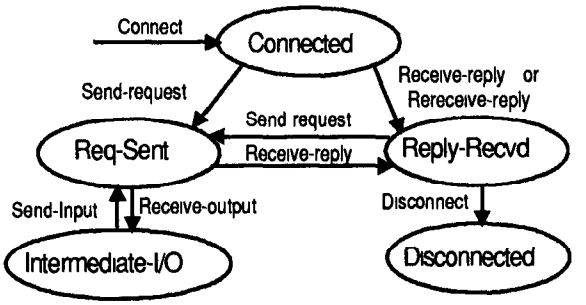

Figure 7 Client's State Transition Diagram for an Interactive Request

8.2 Pseudo-Conversational Transactions

One traditional way to implement an interactive request is to map it into a serial multi-transaction request This approach is called *pseudo-conversational* transactions [McGee 77] Each intermediate output is a reply, and each intermediate input is a request for the next transaction in the sequence The reply to the final transaction in the sequence is interpreted as the reply to the original interactive request Since this implementation is exactly the same as that of Section 6, Request-Reply Matching, Exactly-Once Request-Processing, and At-Least-Once Reply-Processing are preserved

In this implementation, each time the client receives an intermediate output, it knows that its previous input (either the request or its previous intermediate input) was reliably captured, and will not need to be re-sent in the event of a failure However, the approach has the two weaknesses of any multi-transaction request the difficulty of late cancellation and the loss of serializability of request execution

The problem of cancellation is particularly annoying After the first intermediate output is received, the client cannot cancel the request, unless sagas are supported The client is in an awkward position if it is unable to send intermediate input it can't complete the request and it can't cancel it This is not a problem in the non-interactive case, since the client has already supplied all of the input needed to fully execute the request

8.3 Alternative Approaches

An alternative to pseudo-conversational transactions is to have the request execute as one transaction,

which solicits all the intermediate inputs by exchanging ordinary messages with the client This enables the client to cancel the transaction until it sends the last intermediate input that's required It also guarantees that request executions are serializable However, unless special precautions are taken, it does not ensure that intermediate input is reliably captured

One method to ensure intermediate I/O is reliably captured is logging The client logs all intermediate I/O, labeling each log entry with the eid of the request for which the intermediate I/O applies If the interactive transaction aborts, the server starts another transaction for the request, and labels all intermediate I/O with the eid of the request During this replay, as long as the client receives intermediate output that is identical to the request's previous incarnation, it can re-use the intermediate input that it logged during the previous incarnation However, once the client receives intermediate output that differs from the previous incarnation, it must discard the remaining logged intermediate input and must calculate or solicit intermediate input from scratch

9 Related Work

A succinct overview of queuing issues appears in Section 4 of [Gray 78]

Terminal I/O in TP systems is explored in [Pausch 88] It proposes an *External Operation Server (EOS)* to handle reliable input and output of transactions An EOS-process records the values of the I/O operations in persistent data structures It corresponds roughly to a combination of a client and QM *Transaction processes* (which correspond roughly to *servers*) perform I/Os using an EOS The work contains many insights on the types of external operations and implementation of EOS, including the notion of "testable device," which we used in this paper However, it does not abstract QMs separate from EOSs, nor requests separate from transactions, with all that these distinctions imply

Unprotected operations are discussed in [Gray 80, Gray 81] An unprotected operation issued within a transaction does not participate in transaction semantics, it can neither be undone nor redone If the client issues {send request, receive reply, process reply} within a transaction, then "process reply" is usually unprotected In a sense, our Client Model is a way of coping with such unprotected operations

Many existing TP monitors support queues As examples, we mention DECintact [DEC 88], IBM's IMS/DC [McGee 77], and IBM's CICS [IBM 86, Wipfler 87, Wipfler 89] [2]

The DECintact monitor supports recoverable queues, including priority-based Enqueue and Dequeue, queue sets (a view of a set of queues), alert thresholds, and queue redirection (to automatically forward elements from one queue to another) However, persistent registration and recovery on reregistration are not supported

In IMS/DC, queues are the only means of communication between *transaction programs* and *terminals* IMS/DC distinguishes between *terminal queues*, which are dequeued by forms managers (analogous to clients), and *transaction queues*, which are dequeued by transaction programs (analogous to servers) It recommends the pseudo-conversational implementation of interactive requests The system supports a *scratch pad*, which is a field in a queue element that is used to store data to be passed from one transaction to the next for the same multi-transaction request A dequeue (Get_Unique) call returns both the element and the scratch pad Unlike IMS/DC, IMS/FP (Fast Path) disallows conversational mode transactions

IBM's CICS has Transient Data queues, which can be in memory without transaction control, or on disk with transaction control A queue can be associated with a transaction-type name and some numerical limits, which are used to start transaction-tasks when elements arrive in the queue

CICS also has internal work queues of requests that are waiting for a task (process) to be started up Elements are placed in these queues as a result of explicit START commands in other transactions or of users entering requests at their terminals One can place these queue elements under transaction control, so that they reappear after system recovery

CICS includes a feature called Transaction Routing This allows a CICS system A to receive a request and forward it to another CICS system B The request contains information that allows B to create a communication binding with the display that produced the request, so B can effectively communicate directly with the display

10 Implementation Issues

A QM is a type of database system It could be implemented on a file system or a database system, or as an independent database system However, it differs from conventional database systems in some respects

One difference is that most of the data it stores is deleted (dequeued) shortly after being inserted This reduces the importance of checkpointing data to stable storage, although there is still the need to log updates That is, queues can be managed as a main memory database Still, applications that require queue elements to be stored for long periods must be supported

Another difference is in the amount of concurrency between data manipulation operations, such as between enqueue and dequeue operations on the same

[2] DEC and DECintact are trademarks of Digital Equipment Corp

queue For example, it should be possible for one transaction to dequeue (e g , SQL Delete) the top element of a queue, and for a second transaction to do the same before the first transaction commits or aborts Of course, if the first transaction aborts and the second commits, then the Dequeues won't be FIFO ordered But this anomalous ordering is tolerable, when compared to the performance degradation that strict ordering would imply This amounts to allowing readers to scan the queue and ignore (not be blocked by) write-locked elements Another extension is needed to allow a transaction that Dequeues from an empty queue to become blocked (e g , a "notify" lock, similar to "blocking asynchronous system traps" in VMS) Such features are present in some relational database systems such as Tandem's Non-Stop SQL [Tandem 89]

In this paper, we ignored the important issue of scheduling requests Requests may be scheduled for the server by priority, request contents (highest dollar amount first), submission time, etc The server itself is subject to scheduling policy, which determines when it should run and how many instances (threads) it should run The request scheduler is a major component of most TP monitors, and usually requires a QM with content-based retrieval capability

As in object-oriented database systems, element identity is an issue for queues For sequential multi-transaction requests, it is convenient if an element retains its identity as it moves from queue to queue (DECintact makes this guarantee in a centralized system) It would be nice to guarantee this across nodes in a distributed system But this has some implementation challenges since a queue element does not permanently reside in one data server This is closely related to *object mobility*, which has been investigated in other contexts [Jul et al 88][Black and Artsy 89]

A *volatile queue* is one whose contents is lost by a node failure Volatile queues have a useful role in some systems For example, suppose a client redirects its volatile output queue to the volatile input queue of a server at a different node The reliability of the two volatile queues may be as high as that of a single stable queue.

Queue replication can be made explicit Indeed, given the importance of reliably managing requests in a distributed system, queues are a good candidate for being stored as a replicated database that guarantees one-copy serializability, despite the cost of such strong synchronization

Many of these implementation issues have been treated in commercial products A careful survey of these products and their capabilities would be a valuable contribution to the open literature

11 Conclusions

In this paper we described the use of recoverable queues to manage the flow of requests in a TP system We described fault-tolerance requirements of request flow from the client's viewpoint And we showed how TP applications can use queues to meet those requirements

Our motivation for this study was the importance of obtaining exactly-once semantics for processing requests in a TP system, without losing replies However, the solutions we described are more general than that The approach is also valid for obtaining fault-tolerant request-reply semantics in any distributed system For example, it could be applied to remote procedure call systems In this context, an interactive request is one that has call-backs from the remote procedure to the client One could extend the Client Model to support streaming of requests and replies, as in the Mercury system [Liskov et al 88]

We believe that persistent registration with operation tags is an important capability in realizing fault-tolerant client models The idea is quite general it can be used to create an arbitrary fault-tolerant session between the client and system Persistent registration is a session in that it contains state that is shared by the client and system By making it persistent, the session can be recovered when the client reconnects after a failure

The client-system protocol we described also applies to the end user's interaction with the client The user submits a request to the client (program), and later expects a reply, just as the client submits a request to the system and later expects a reply There are, of course, no queues between the user and client So, we need another protocol to give the user the semantics he or she wants For example, we may require that the user enters a unique identifier with each request, and that the client returns this identifier with the corresponding reply The client may fail, or the user may have a memory lapse (another kind of failure) So the user should checkpoint that identifier (e g , on a piece of paper), so the user can figure out where the user and client left off Like the client in its relationship to the system, the user behaves like a fault-tolerant sequential program, at recovery time he or she must determine his or her last non-idempotent operation (i e , the last request submission) and must checkpoint his or her state when issuing this operation Just as the client asks the system to checkpoint its state in a tag, the user can ask the client to checkpoint his or her state in the request

It is interesting to compare the reliability of replies in our model with that of transaction commit or abort In transaction management, once a transaction has committed, and all of the data managers that participated in the transaction have acknowledged receiving the commit directive, the transaction manager forgets about the transaction Under the presumed abort

protocol, the transaction manager also forgets about a transaction as soon as it learns that the transaction aborts The user is not a participant in the commit protocol So a transaction can be forgotten by the transaction manager, and then the system may fail, before the user sees the result of the transaction After the system recovers, there is no evidence in the transaction manager of whether or not the transaction ran There must be another mechanism for retaining the disposition of a transaction for long periods, so the user can find out what the transaction did, long after the transaction disappears At-least-once reply processing provides such a mechanism

Like other popular features of TP monitors, such as multi-threaded processes and transactions, queuing is a general-purpose feature that should be incorporated in all distributed computing architectures

Acknowledgements

Many people contributed to discussions that led to this paper, including Andrew Black, Mark Brown, John Clarke, Bill Emberton, Dieter Gawlick, Jim Gray, Rivka Ladin, Bill Laing, Rich Larson, Dave Lomet, Barry Rubinson, Paul Shrager, Diogenes Torres, and Mark Tuttle We thank them all for their help We especially thank Mike Rosen, who helped us crystalize aspects of persistent registration and many implementation issues

References

[Bernstein et al 87] Bernstein, P , V Hadzilacos, and N Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, MA, 1987

[Birrell and Nelson 84] Birrell, A D , and B J Nelson "Implementing Remote Procedure Calls," *ACM Trans on Computer Sys* , Vol 2, No 1 (Feb 1984), pp 39-59

[Black and Artsy 89] "Implementing Location Independent Invocation," *Proc 9th Int 'l Conf on Dist'd Computing Systems*, June 1989, pp 550-559

[DEC 88] Digital Equipment Corp , *DECintact Transaction Processing System Application Programming Guide,* Maynard, MA, 1988 Order Number AA-KZ03A-TE

[Garcia and Salem 87] Garcia-Molina, H and K Salem, "Sagas," *Proc ACM SIGMOD Conf* , May 1987

[Gray 78] Gray, J , "Notes on Database Operating Systems," *Operating Sys , An Adanced Course* (Ed by R Bayer, R M Graham, G Seegmuller), Lecture Notes in Computer Science 60, Springer-Verlag, N Y , 1978, pp 393-481 Section 4 on "Data Communications" is on pp 415-420

[Gray 80] Gray, J , "A Transaction Model," Technical Report RJ2895, IBM Research Laboratory, San Jose, CA, 1980

[Gray 81] Gray, J , "The Transaction Concept Virtues and Limitations " *Proc Int'l Conf on Very Large Data Bases*, 1981, pp 144-154

[IBM 86] International Business Machines, "CICS/OS/VS Intercommunications Facilities Guide," Form SC33-0230, White Plains, N Y , 1986

[Jul et al 88] Jul, E , H Levy, N Hutchinson, and A Black, "Fine-Grained Mobility in the Emerald System," *ACM Trans on Computer Sys* , Vol 6, No 1 (Feb '88), pp 109 - 133

[Klein and Reuter 88] Klein, J and A Reuter, "Migrating Transactions," *Future Trends in Distributed Computer Systems in the '90s*, Hong Kong, 1988

[Liskov et al 88] Liskov, B , T Bloom, D Gifford, R Scheifler, and W Weihl, "Communication in the Mercury System," *Proc 21st Hawaii Conf on System Sciences*, Jan 1988 (also, MIT Laboratory for Computer Science Programming Methodology Group Memo 59, Oct 1987)

[McGee 77] McGee, W C , "The Information Management System IMS/VS Part V Transaction Processing Facilities," *IBM Sys Journal*, Vol 16, No 2 , 1977, pp 148-169

[Pausch 88] Pausch, R , "Adding Input and Output to the Transaction Model," Ph D Thesis, Computer Science Dept , Carnegie Mellon Univ , August, 1988 (CMU-CS-88-171)

[Tandem 89] The Tandem Database Group, "NonStop SQL A Distributed, High Performance, High Availability Implementation of SQL," in *High Perf Transaction Sys* (Ed by D Gawlick, M Haynie, A Reuter), Lecture Notes in Computer Science 359, Springer-Verlag, N Y , 1989, pp 60-104

[Wipfler 87] Wipfler, A J , "CICS Application Development and Programming," Macmillan, N Y , 1987

[Wipfler 89] Wipfler, A J , "Distributed Processing in the CICS Environment," McGraw-Hill, N Y , 1989