9. Queued Transaction Processing

CSEP 545 Transaction Processing
Philip A. Bernstein
Copyright © 2005 Philip A. Bernstein

Outline

1. Introduction
2. Transactional Semantics
3. Queue Manager
Appendices
A. Marshaling
B. Multi-Transaction Requests (Workflow)
C. (Appendix) Microsoft Message Queue

9.1 Introduction

• Direct TP - a client sends a request to a server, waits (synchronously) for the server to run the transaction and possibly return a reply (e.g., RPC)
• Problems with Direct TP:
  - Server or client-server communications is down when the client wants to send the request
  - Client or client-server communications is down when the server wants to send the reply
  - If the server fails, how does the client find out what happened to its outstanding requests?
  - Load balancing across many servers
  - Priority-based scheduling of busy servers

Persistent Queuing

• Queuing - controlling work requests by moving them through persistent transactional queues

Other Benefits

• Queue manager as a protocol gateway
  - need to support multiple protocols in just one system environment
  - can be a trusted client of other systems to bridge security barriers
• Explicit traffic control, without message loss
• Safe place to do message translation between application formats

9.2 Transaction Semantics Server View

• The queue is a transactional resource manager
• Server dequeues requests within a transaction
• If the transaction aborts, the dequeue is undone, so the request is returned to the queue

Server Program

Start
Dequeue(Req, Q1)
process request Req
Enqueue(Reply, Q2)
Commit
Transaction Semantics
Server View (cont'd)
- Server program is usually a workflow controller
- It functions as a dispatcher to
  - get a request,
  - call the appropriate transaction server, and
  - return the reply to the client.
- A abort-count limit and error queue to deal with requests that repeatedly lead to an aborted transaction.

Transaction Semantics - Client View
- Client runs one transaction to enqueue a request and a second transaction to dequeue the reply.

Client Recovery
- If a client times out waiting for a reply, it can determine the state of the request from the queues:
  - request is in Q1, reply is in Q2, or request is executing
- Assume each request has a globally unique ID
- If client fails and then recovers, a request could be in one of 4 states:
  A. Txn1 didn't commit – no message in either queue.
  B. Txn1 committed but server's Txn2 did not – request is either in request queue or being processed
  C. Txn2 committed but Txn3 did not – reply is in the reply queue
  D. Txn3 committed – no message in either queue

Client Recovery (2)
- So, if the client knows the request id R, it can determine state C and maybe state B.
- What if no queued message has the id R?
  Could be in state A, B, or D.
- Can further clarify matters if the client has a local database that can run 2-phase commit with the queue manager:
  - Use the local database to store the state of the request.

Transaction Semantics - Client View
- Client runs one transaction to enqueue a request and a second transaction to dequeue the reply.
Client Recovery (3)

- If client fails and then recovers, a request R could be in one of 4 states:
  - A. Txn1 didn't commit - LocalDB says R is NotSubmitted.
  - B. Txn1 committed but server's Txn2 did not - LocalDB says R is Submitted and R is either in request queue or being processed.
  - C. Txn2 committed but Txn3 did not - LocalDB says R is Submitted and R's reply is in the reply queue.
  - D. Txn3 committed - LocalDB says R is Done.

- To distinguish B and C, client first checks request queue (if desired) and then polls reply queue.

Persistent Sessions

- Suppose client doesn't have a local database that runs 2PC with the queue manager.
- The queue manager can help by persistently remembering each client's last operation, which is returned when the client connects to a queue ... an entry to a persistent session.

Client Recovery with Persistent Sessions

- Now client can figure out:
  - A - If last enqueued request is not R
  - D - If last dequeued reply is R
  - B - No evidence of R, and not in states A, C, or D.

// Let R be id of client's last request
// Assume client ran Txn0 for R before Txn1
Client connects to request and reply queues;
If (id of last request enqueued ≠ R) { resubmit request }
elseif (id of last reply message dequeued ≠ R) { dequeue (and wait for) reply with id R }
else // R was fully processed, nothing to recover

Non-Undoable Operations

- How to handle non-undoable non-idempotent operations in txn3?

  Txn3: Start
  Dequeue(Reply for R, Q2)
  decode reply & process output
  State(R) = "Done"
  Commit

  Crash!

  => R was processed.
  But Txn3 aborts.
  So R is back on Q2.

- If the operation is undoable, then undo it.
- If it's idempotent, it's safe to repeat it.
- If it's neither, it had better be testable.

Testable Operations

- Testable operations:
  - After the operation runs, there is a test operation that the client can execute to tell whether the operation ran.
  - Typically, the non-undoable operation changes the state of the device (before-state) and then changes the state of the device.
  - The test operation returns a description of the state of the device.
  - E.g., State description can be a unique ticket/check/$form number under the print head.

Recovery Procedure for State C

- To process a reply
  1. Start a transaction
  2. Dequeue the reply
  3. If there's an earlier logged device state for this reply and it differs from the current device state, then ask the operator whether to abort this txn
  4. Persistently log the current device state with the reply's ID. This operation is permanent whether or not this transaction commits.
  5. Perform the operation on the physical device
  6. Commit
Optimizations

- In effect, the previous procedure makes the action "process output" idempotent.
- If "process output" sent a message, it may not be testable, so make sure it's idempotent!
  - if box3 is sending a receipt, label it by the serial number of the request, so it can be sent twice.
- Log device state as part of Dequeue operation (saves an I/O)
  - i.e., run step 3 before step 2

Queue Manager (cont'd)

- A box has some common communication types of operations
  - start and stop queue
  - volatile queues (lost in a system failure)
  - persistent sessions (explained earlier)
- System manager operations
  - monitor load
  - report on failures and recoveries

Example of Enqueue Parameters (IBM MQ Series)

- System-generated and application-assigned message IDs
- Name of destination queue and reply queue (optional)
- Flag indicating if message is persistent
- Message type - datagram, request, reply, report
- Message priority
- Correlation ID to link reply to request
- Expiry time
- Application-defined format type and code page (for I18N)
- Report options - confirm on arrival when enqueued?, on delivery when dequeued?, on expiry?, on exception?

Priority Ordering

- Prioritize queue elements
- Dequeue by priority
- A box makes strict priority-ordered dequeue too expensive
  - could never have two elements of different priorities dequeued and uncommitted concurrently
- But some systems require it for legal reasons
  - stock trades must be processed in time stamp order

Routing

- Forwarding of messages between queues
  - transactional, to avoid lost messages
  - batch forwarding of messages, for better throughput
  - can be implemented as an ordinary transaction server
- Often, a lightweight client implementation supports a client queue
  - captures messages when client is disconnected, and
  - forwards them when communication to queue server is re-established
- Implies system management to display topology of forwarding links
State of the Art

- All app servers support some form of queuing
- A new trend is to add queuing to the SQL DBMS
  - Oracle has it: Avoda 2PC for Tnx2, allow s queues...
- Queuing is hard to build well: It’s a product or major subsystem, not just a feature.
- Lots of queuing products with small market share.
- Some major ones are:
  - IBM’s MQ Series
  - BEA Systema’s Message Q
  - Microsoft Message Queuing

Appendix A: Marshaling

- Caller of Enqueue and Dequeue needs to m arshal and unmarshal data into variables
- Instead, use the automatic marshaling of RPC
- Here’s how RPC works:

A adapting RPC Marshaling for Queues

- In effect, use queuing as a transport for RPC
- Example - Queued Component in M SM Q

Appendix B: Multi-Transaction Requests

- Some requests cannot execute as one transaction because
  - It executes too long (causing lock contention) or
  - Resources don’t support a compatible 2-phase commit protocol.
- Transaction may run too long because
  - It requires display I/O with user
  - People or machines are unavailable (hotel reservation system, manager who approves the request)
  - It requires long-running real-world actions (get 2 estimates before settling an insurance claim)
- Transaction may be required to run independent ACID transactions in subsystems (placing an order, scheduling a shipment, reporting commission)

Workflow

- A multi-transaction request is called a workflow
- Integrated workflow products are now being offered.
  - IBM MQSeries Workflow, MS BizTalk Orchestration, TIBCO, J2EES, BEA Weblogic Process Integrator, Action,...
  - See also www.workflowsoftware.com, www.wfmc.org
- They have special features, such as
  - Flow graph language for describing processes consisting of steps, with preconditions for moving between steps
  - Representation of structural and roles (a role step can be performed ed by a person in a role, with complex role resolution procedures)
  - Tracing of steps, locating in-flight workflows
  - Ad hoc workflow, integrated with e-mail (casemgmt)

Managing Workflow with Queues

- Each workflow step is a request
- Send the request to the queue of the server that can process the request
- Server outputs request(s) for the next step(s) of the workflow

Submit expense claim → Validate claim → Get Manager Approval

Email notification → Request Automatic Deposit → Authorize Payment
Workflows Can Violate Atomicity and Isolation

- Since a workflow runs as many transactions,
  - it may not be serializable relative to other workflows
  - it may not be all-or-nothing

Consider a money transfer run as 2 txns, T1 & T2
- Conflicting money transfers could run between T1 & T2
- A failure after T1 might prevent T2 from running
- These problems require application-specific logic
  - E.g. T2 must send ack to T1's node. If T1's node times out waiting for the ack, it takes action, possibly compensating for T1.

Automated Compensation

In a workflow specification, for each step, identify a compensation. Specification is called a saga.

- If a workflow stops making progress, run compensations for all committed steps, in reverse order (like transaction abort).
- Need to ensure that each compensation's input is available (e.g. log it) and that it definitely can run (enforce constraints until workflow completes).
- Concept is still at the research stage.

Pseudo-conversations

- Simple solution in early TP system products
- A conversational transaction interacts with its user during its execution
  - This is a sequential workflow between user & server.
  - Since this is long-running, it should run as a single request
- Since there are exactly two participants, just pass the request back and forth
  - request carries all workflow context
  - request is recoverable, e.g. send/receive is logged or request is stored in shared disk area
  - This simple mechanism has been superseded by queues and general-purpose workflow systems.

Maintaining Workflow State

- Queue elements and pseudo-conversation requests are places for persistent workflow state. Other examples:
  - Browser cookies (files that are read/written by http requests), containing user profile information
  - Shopping cart (in web server cache or database)

This simple mechanism has been superseded by queues and general-purpose workflow systems.

Appendix C: Microsoft Message Queuing (MSMQ)

- Clients enqueue/dequeue to queue servers
  - API - Open, Close, Send & receive
  - Each queue is named in the Active D Directory
  - Additional functions: Create/Destroy queue, Locate queue, Get/Set queue properties, Set/Get queue security

Send & receive can be
- Transactional or persistent queues (transparently gets transaction context), using DTC
- Non-transactional or persistent/volatile queues

Independent client has a local persistent queue store.
- Processes ops locally, asynchronously sends to a server
- B dependent clients issue RPC to a queue server (leader to admin nodes, few or resources required)

MSMQ Servers

- Stress test messages
- Dynamic, in-context routing
- Volatile or persistent (local) store and flow and
- Support local/independent clients and flow existing transport servers/independent clients
- Provides MSMQ Explorer
  - Topologies, routing, etc.
- Security via ACLs, journals, public key authentication
MS MQ Interoperation

- Exchange Connector - Send and receive messages and forms through Exchange Server and MS MQ
- MAPI transport - Send and receive messages and forms through MAPI and MS MQ
- Via Level 8 System s,
  - Clients - MVS, AS/400, VM S, HP-Unix, Sun-Solaris, AIX, OS/2 clients
  - Interoperates with IBM MQ Series