4. Queued Transaction Processing

CSE593 Transaction Processing

Philip A. Bernstein

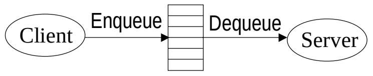
Outline

4.1 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



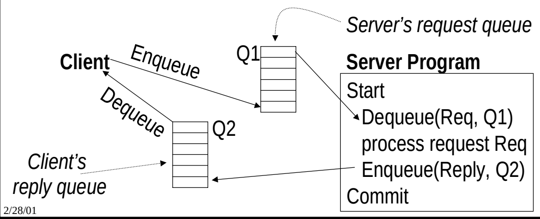
- Benefits of queuing
  - client can send a request to an unavailable server
  - server can send a reply to an unavailable client
  - since the queue is persistent, a client can (in principle) find out the state of a request
  - can dequeue requests based on priority
  - can have many servers feed off a single queue

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

4.2 Transaction Semantics Server View

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



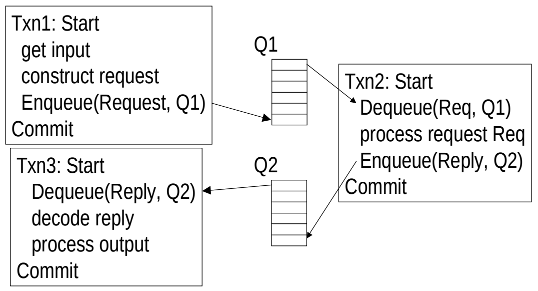
```
Start
  Dequeue(Req, Q1)
  process request Req
  Enqueue(Reply, Q2)
Commit
```

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.
- 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



```
Txn1: Start
  get input
  construct request
  Enqueue(Request, Q1)
Commit
```

```
Txn2: Start
  Dequeue(Req, Q1)
  process request Req
  Enqueue(Reply, Q2)
Commit
```

```
Txn3: Start
  Dequeue(Reply, Q2)
  decode reply
  process output
Commit
```

Transaction Semantics Client View (cont'd)

- Client transactions are very light weight
- Still, every request now requires 3 transactions, two on the client and one on the server
  - Moreover, if the queue manager is an independent resource manager (rather than being part of the database system), then Transaction 2 requires two phase commit
- So queuing's benefits come at a cost

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 (cont'd)

- 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
  - client is "single-threaded" (≤ one request in flight), and
  - queue server persistently remembers client's last operation, which is returned when client connects to a queue … amounts to a <u>persistent session</u>

Client Recovery (cont'd)

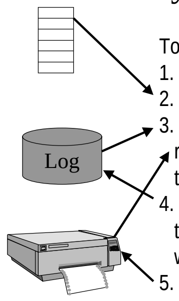
- Now client can figure out
  - A – if last enqueued request is not R
  - D – if last dequeued reply is R
  - B – no evidence of R and not in states A, C, or D.

```
// Let R be id of client's last request
// Assume client persistently stores R before submitting request
Connect to request and reply queues;
If (id of last request message 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 ?
- Require that such operations be testable
  - After the operation runs, there must be a test operation that can tell whether it ran
  - Typically, the non-undoable operation returns a description of the state of the device (before-state) and then changes the state of the device, and
  - 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

- Work hard to make operations idempotent
  - if txn3 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)

4.3 Queue Manager

- A queue supports most file-oriented operations
  - create and destroy queue database
  - create and destroy queue
  - show and modify queue's attributes (e.g. security)
  - open-scan and get-next-element
  - enqueue and dequeue
    - next element or element identified by index
    - inside or outside a transaction
  - read element

Queue Manager (cont'd)

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

Example of Enqueue Parameters (IBM MQSeries)

- 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

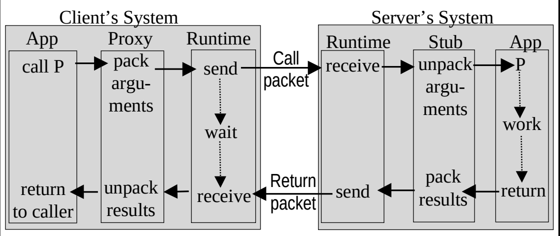
- Prioritize queue elements
- Dequeue by priority
- Abort makes strict priority-ordered dequeue too expensive
  - could never have two elements of different priorities dequeued and uncommitted concurrently
- But some systems require it for legal reasons
  - stock trades must be processed in timestamp 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 mgmt requirement to display topology of forwarding links

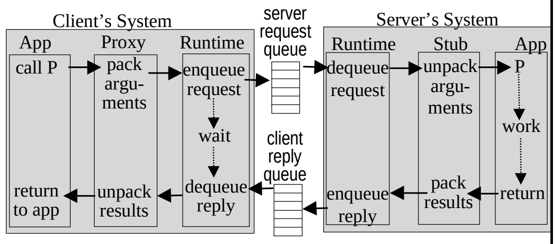
Marshaling

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



Adapting RPC Marshaling for Queues

- In effect, use queuing as a transport for RPC
- Example – Queued Component in MSMQ



4.4 Multi-Transaction Requests

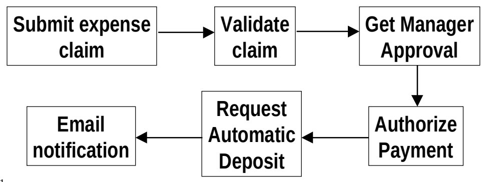
- Some requests cannot execute as one transaction because
  - It executes too long (causing lock contention) or
  - Resources don't support a compatible 2-phase commit protocol.
- Transaction may run too long because
  - It requires display I/O with user
  - People or machines are unavailable (hotel 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, JetForm, BEA WebLogic Process Integrator, Action, ...
  - See also www.workflowsoftware.com, www.wfmc.org
- They have special features, such as
  - flowgraph language for describing processes consisting of steps, with preconditions for moving between steps
  - representation of organizational structure and roles (manual step can be performed by a person in a role, with complex role resolution procedure)
  - tracing of steps, locating in-flight workflows
  - ad hoc workflow, integrated with e-mail (case mgmt)

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



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, T_1 & T_2
  - Conflicting money transfers could run between T_1 & T_2
  - A failure after T_1 might prevent T_2 from running
  - These problems require application-specific logic
  - E.g. T_2 must send ack to T_1's node. If T_1's node times out waiting for the ack, it takes action, possibly compensating for T_1

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 multiple requests
- 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 superceded by queues and general-purpose workflow systems.

Maintaining Workflow State

- Queue elements and pseudo-conversation requests are places 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)
- Such state management arises within a transaction too
  - Server scans a file. Each time it hits a relevant record, return it.
  - Issue: later calls must go to the same server, since only it knows where the transaction's last call left off.
  - Sol'n 1: keep state in the message (like pseudo-conversation)
  - Sol'n 2: first call gets a binding handle to the server, so later calls go to it. Server needs to release state when client disappears

State of the Art

- All TP monitors support some form of queuing
- Queuing is hard to build well. It's a product, not just a TP monitor component.
- Lots of queuing products with small market share.
- Some major ones are
  - IBM's MQSeries
  - BEA Systems MessageQ
  - Microsoft Message Queuing

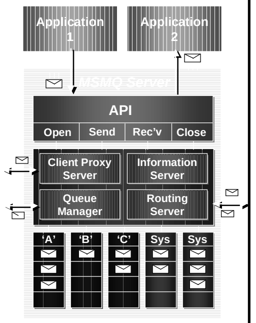
Appendix: Microsoft Message Queuing (MSMQ)

- Clients enqueue/dequeue to queue servers
  - API - Open/Close, Send/Receive
  - Each queue is named in the Active Directory
  - Additional functions: Create/Delete queue, Locate queue, Set/Get queue properties, Set/Get queue security
- Send/Receive can be
  - Transactional on persistent queues (transparently gets transaction context), using DTC
  - Non-transactional on persistent/volatile queues
- *Independent client* has a local persistent queue store.
  - Processes ops locally, asynchronously sends to a server
  - Dependent client issues RPC to a queue server (easier to administer, fewer resources required)

2/28/01 31

MSMQ Servers

- Stores messages
- Dynamic min-cost routing
- Volatile or persistent (txnal) store and forward
- Support local / dependent clients and forwarding from servers / independent clients
- Provides MSMQ Explorer
  - Topologies, routing, mgmt
- Security via ACLs, journals, public key authentication



2/28/01 32

MSMQ Interoperation

- Exchange Connector - Send and receive messages and forms through Exchange Server and MSMQ
- MAPI transport - Send and receive messages and forms through MAPI and MSMQ
- Via Level 8 Systems,
  - Clients - MVS, AS/400, VMS, HP-Unix, Sun-Solaris, AIX, OS/2 clients
  - Interoperates with IBM MQSeries

2/28/01 33