9. Queued Transaction Processing

CSEP 545 Transaction Processing

Philip A. Bernstein

Outline

1. Introduction
2. Transactional Semantics
3. Queue Manager
4. Message-Oriented Middleware

Appendices

A. Marshaling

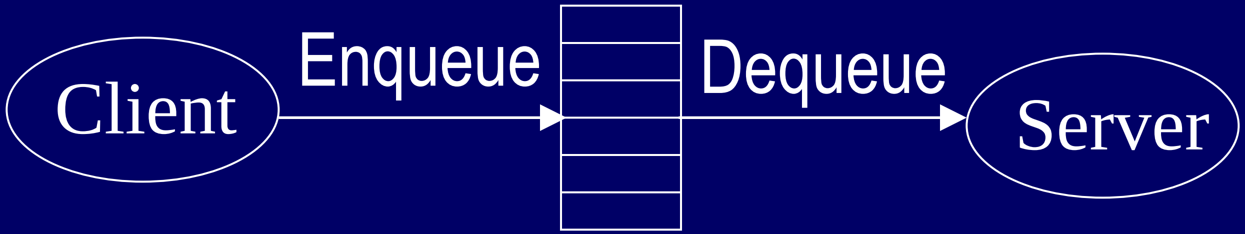
B. Microsoft Message Queue

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



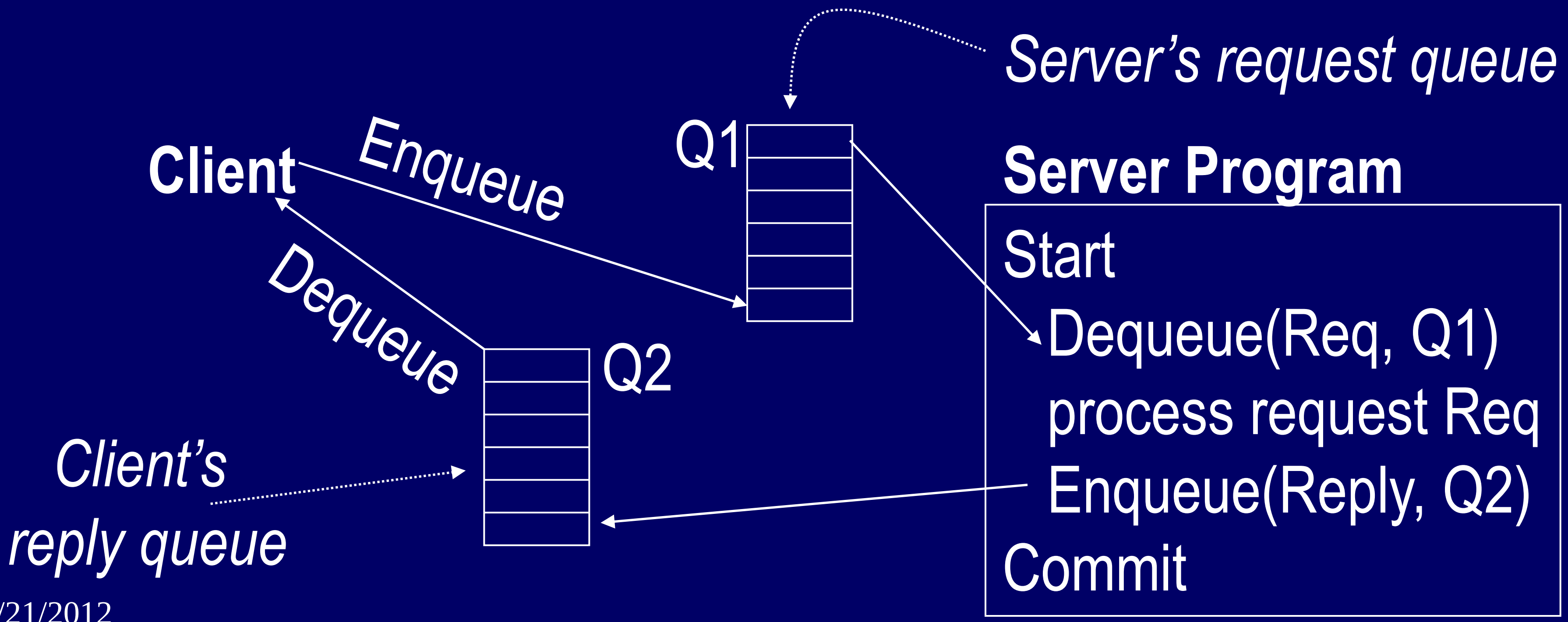
- Benefits of queuing
  - Client can send a request to an unavailable server
  - Server can send a reply to an unavailable client
  - Since the queue is persistent, a client can (in principle) find out the state of a request
  - Can dequeue requests based on priority
  - Can have many servers feed off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

9.2 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

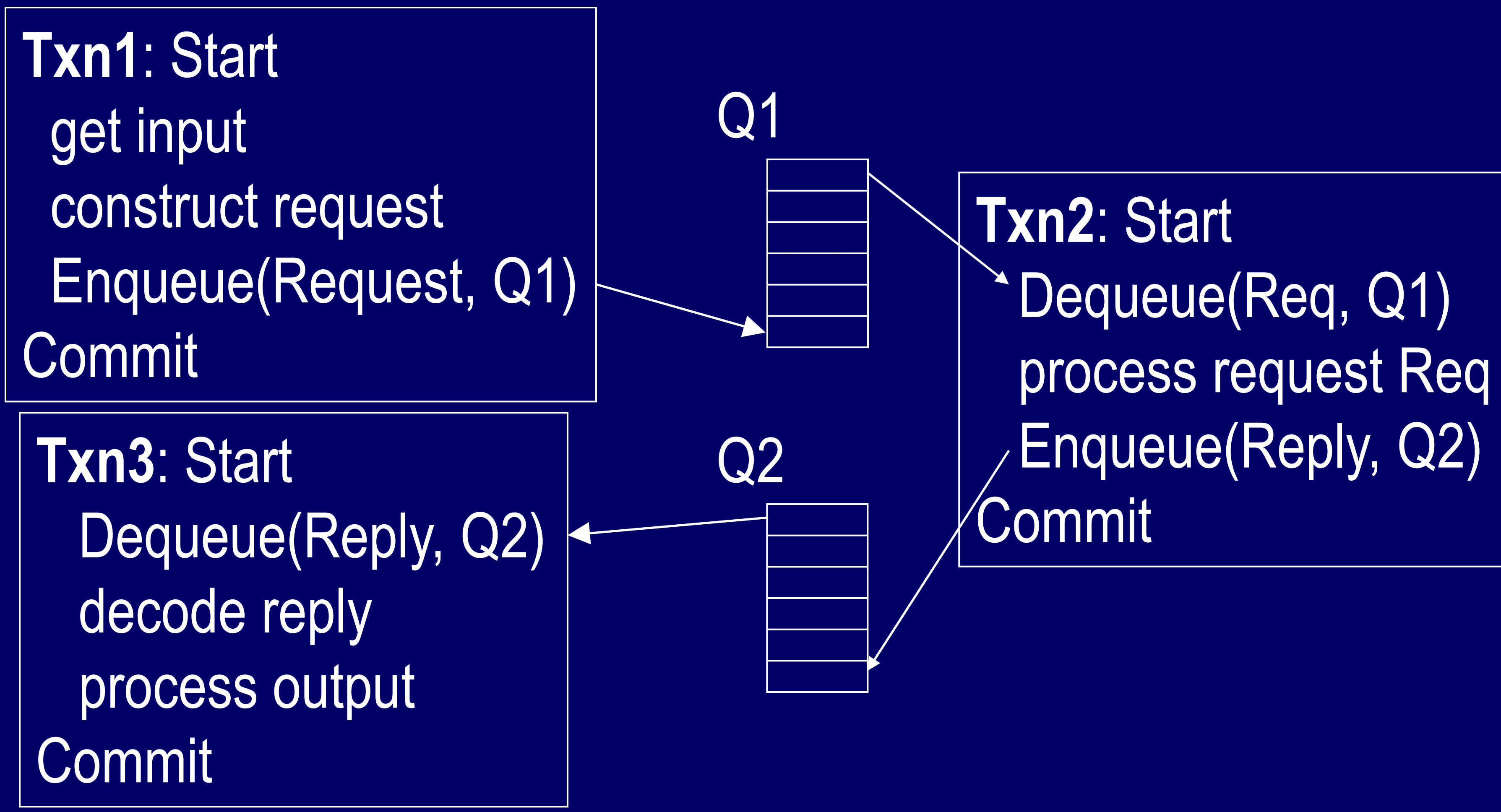


Transaction Semantics Server View (cont'd)

- 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



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

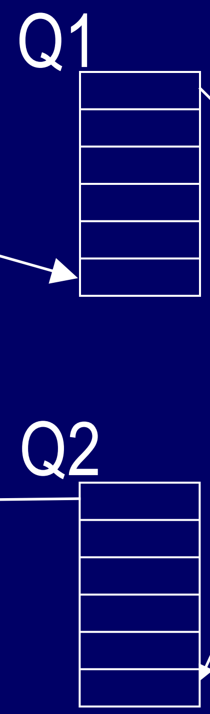
Client Recovery (2)

- 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

Txn0: Start
get input & construct request R
LastRequest = R
Commit

Txn1: Start
R = LastRequest
Enqueue(Request Q, R)
LastEnqueuedID=R.ID
Commit



Txn2: Start
Req=Dequeue(RequestQ)
process request Req
Enqueue(ReplyQ, reply)
Commit

Txn3: Start
R = Dequeue(Reply Q)
decode reply & process output
LastDequeuedID=R.ID
Commit

Client Recovery (3)

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

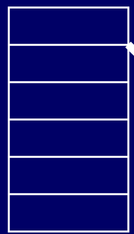
=> R was processed.
  But Txn3 aborts.
  So R is back on Q2.

- 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

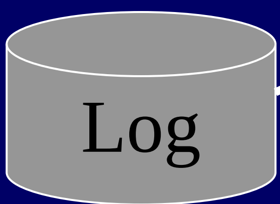
- Testable operations
  - After the operation runs, there is a test operation that the client can execute to tell whether the operation ran
  - Typically, the non-undoable operation returns a description of the state of the device (before-state) and then changes the state of the device
  - The test operation 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

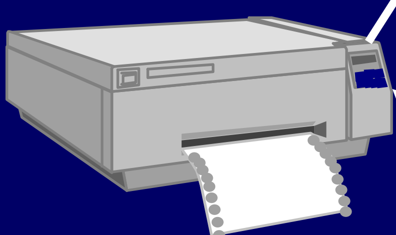
Recovery Procedure for State C



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 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).
  - i.e., run step 3 before step 2.

9.3 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 Websphere MQ)

- 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

State of the Art

- All app servers support some form of queuing
- A new trend is to add queuing to the SQL DBMS
  - Oracle & SQL Server have it.
  - Avoids 2PC for Txn2, allows queries, ….
- Queuing is hard to build well.
  - It's a product or major sub-system, not just a feature.
- Lots of queuing products with small market share.
- Some major ones are
  - IBM's MQSeries
  - Oracle BEA MessageQ
  - Oracle Streams AQ
  - Microsoft Message Queuing

9.4 Message-Oriented Middleware

- Publish-Subscribe
- Message Broker
- Message Bus

Publish-Subscribe

- Using queues, each message has one recipient
- Some apps need to send to multiple recipients
  - E.g., notify changes of stock price, flight schedule
- Publish-subscribe paradigm allows many recipients per message
  - Subscribers sign up for message types, by name (e.g. "Reuters") or predicate (type="Msft" and price > 33)
- Similar to queues
  - Send and receiver are decoupled
  - Send or receive within a transaction
  - Subscribers can push (dispatch) or pull (dequeue)

Publish-Subscribe (cont’d)

- Hence, often supported by queue managers
- E.g., Java Messaging Service (JMS) defines both peer-to-peer and pub-sub interfaces

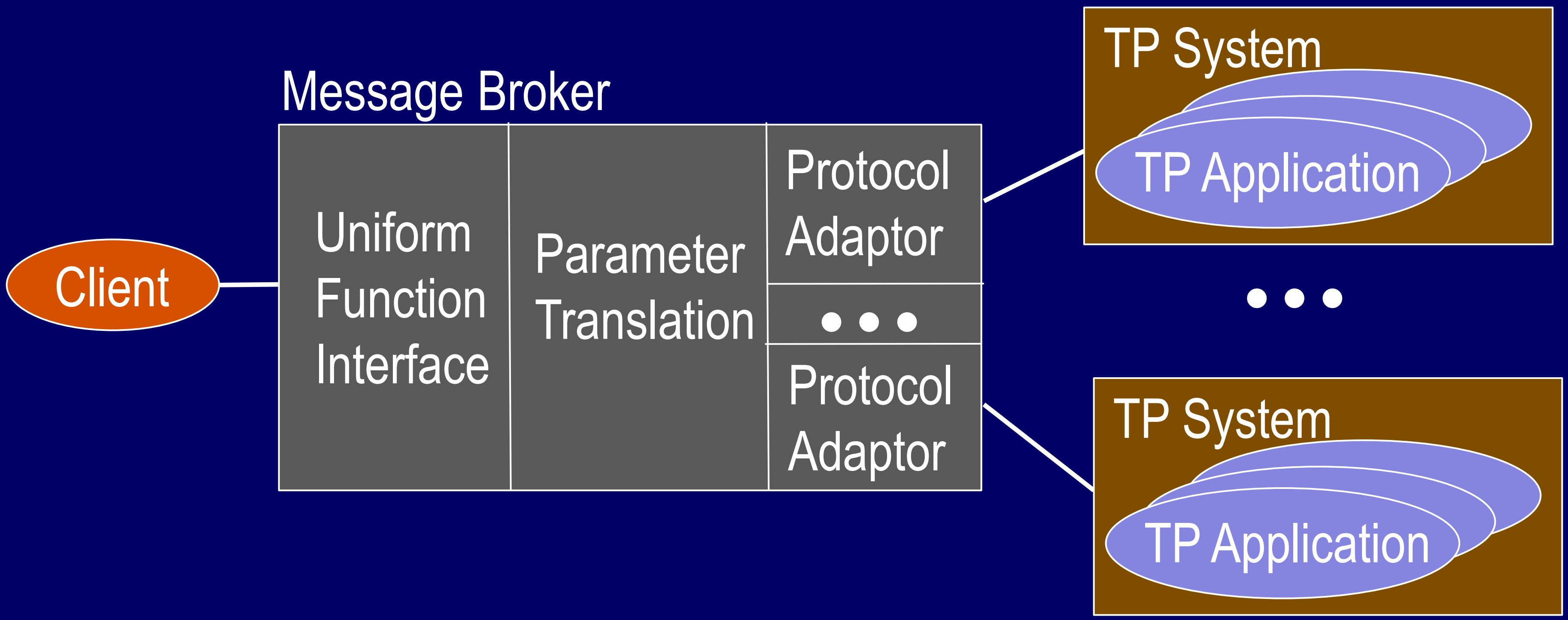
Data Streams AKA Complex Event Processing

- Treat a message stream like a table in a SQL database
- A query retrieves messages within a time window
  - Distinguish message-time from event-time
- An evolution of pub-sub and of SQL databases
- Applications – sensors, financial markets, intelligence, gaming, telecom, mobile commerce, ….
- Products are from startups & major vendors
  - Microsoft Stream Insight
  - IBM InfoSphere Streams
  - Oracle Complex Event Proc.
  - StreamBase

Broker and Bus Middleware

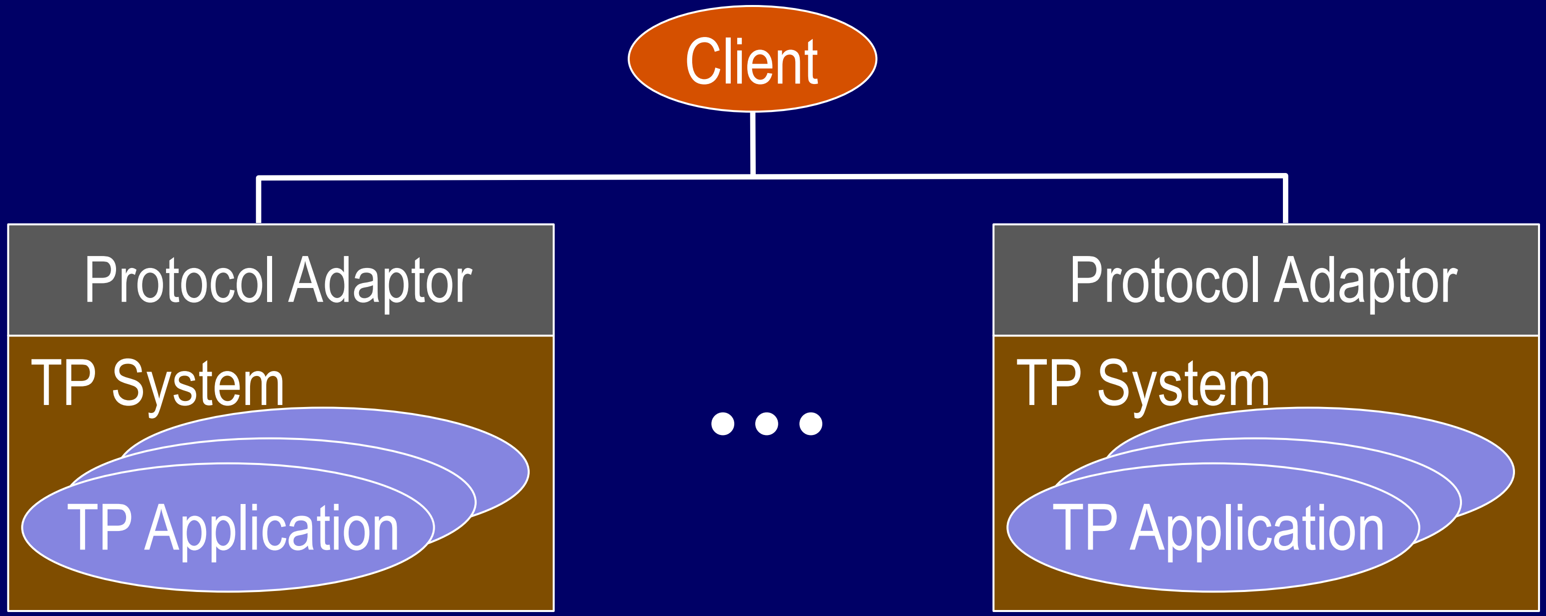
- Messaging technology is often used to integrate independent applications.
  - Broker-based: Enterprise Application Integration (EAI)
  - Bus-based: Enterprise Server Bus (ESB)
- These functions are often combined with queuing and/or pub-sub.

Broker-Based



- The Broker bridges wire protocols, invocation mechanisms, and parameter formats

Bus-Based



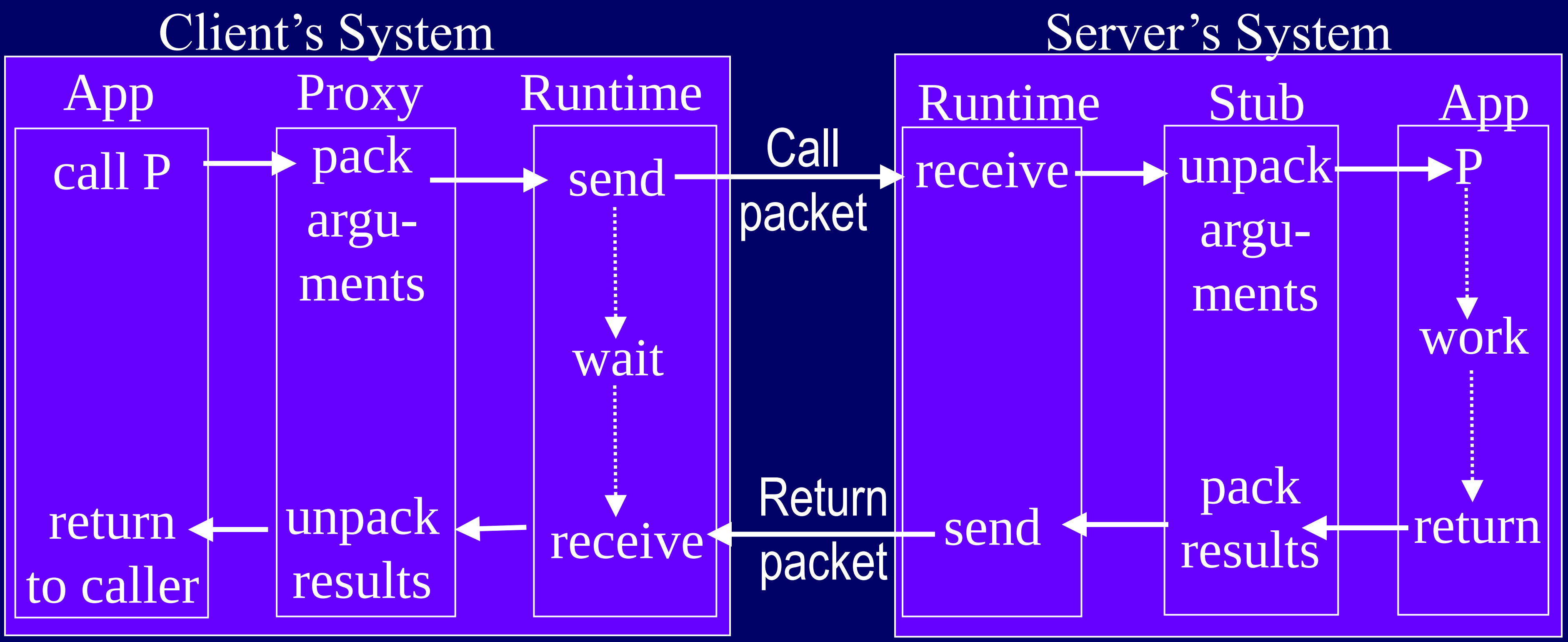
- All apps support the same protocol and invocation mechanism
- Client or broker functions still needed for translating parameter formats

Trends

- There is much variety in message-oriented middleware
- Probably this will continue, followed by some shakeout

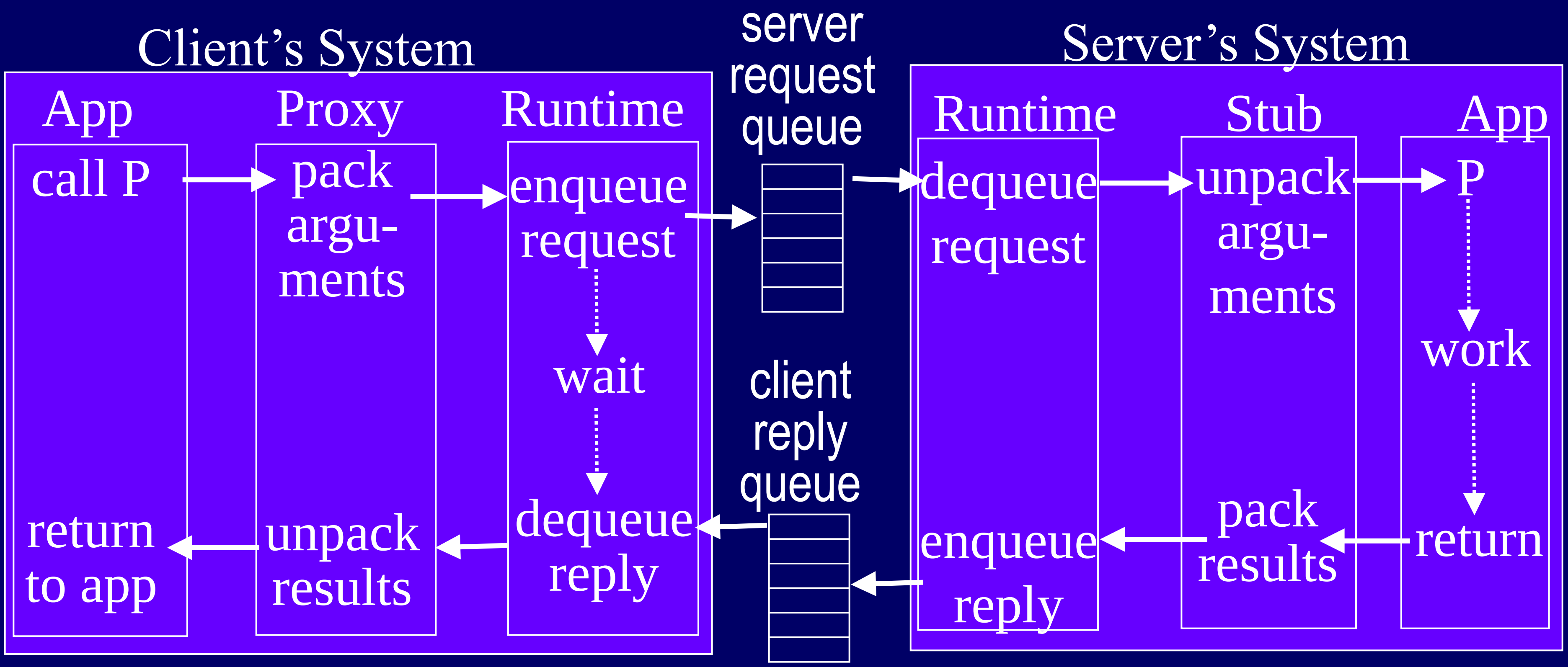
Appendix A: Marshaling

- 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

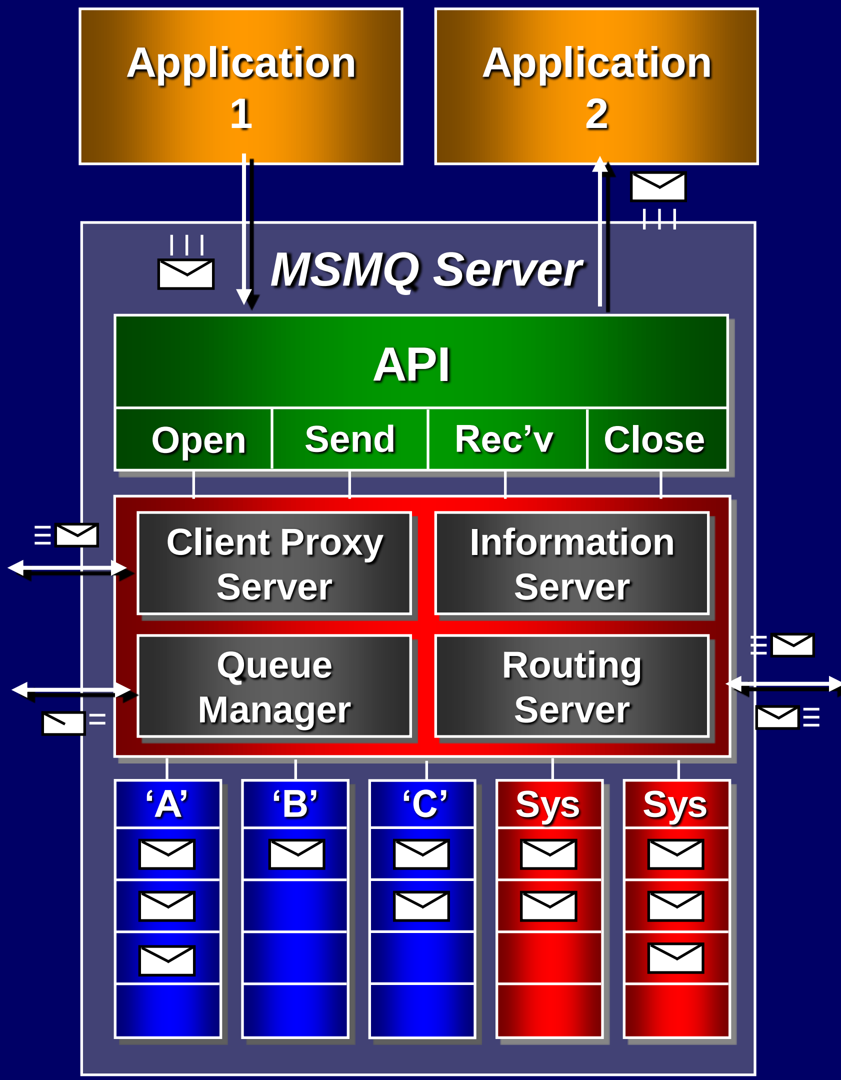


Appendix B : Microsoft Message Queuing (MSMQ) [from 2003]

- 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)

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



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