Prof. Dr.-Ing. Stefan Deßloch
AG Heterogene Informationssysteme
Geb. 36, Raum 329
Tel. 0631/205 3275
dessloch@informatik.uni-kl.de

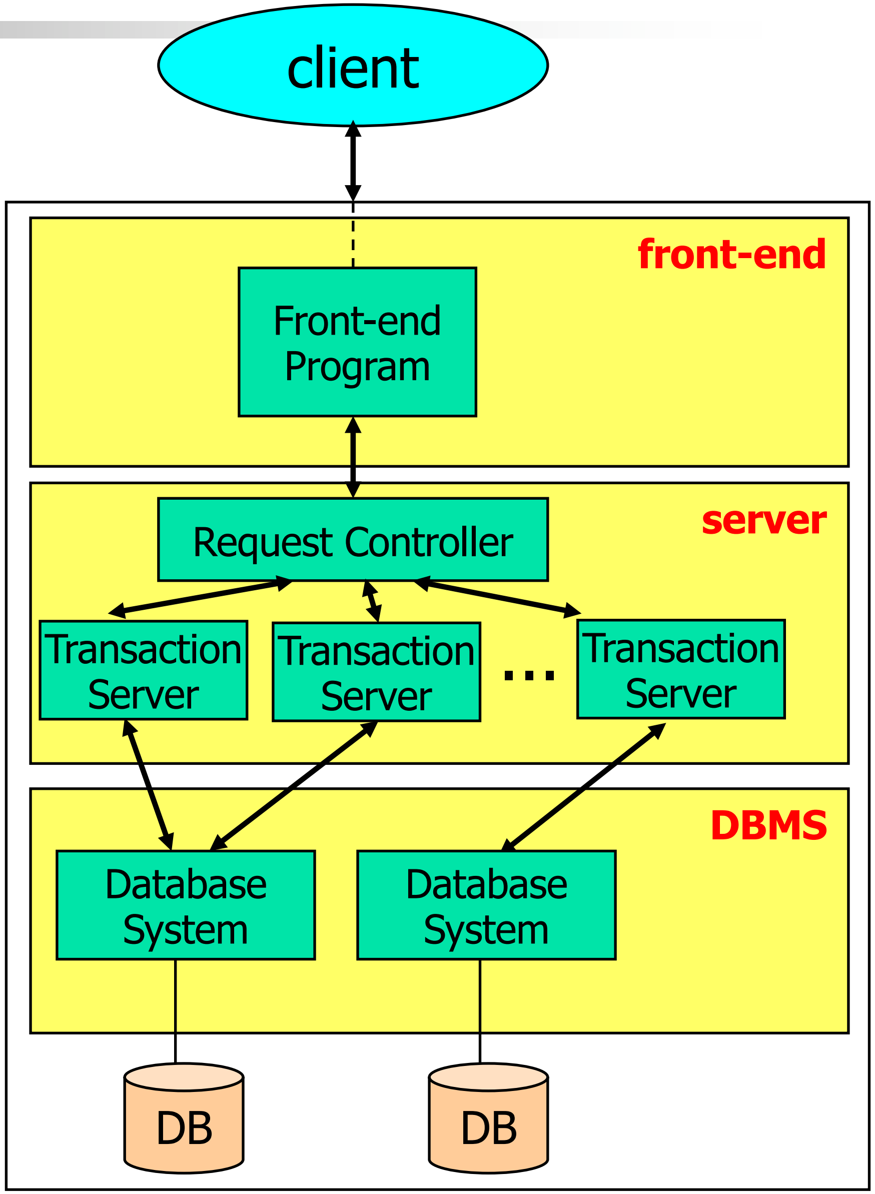
Chapter 7
Message-oriented Middleware (MOM)

Transactional Request Processing

- Direct transactions
  - requests are immediately authorized, dispatched, executed
  - reponses are returned to clients right away
  - may involve TRPC, distributed TP
- Limitations of direct TP
  - server unavailable

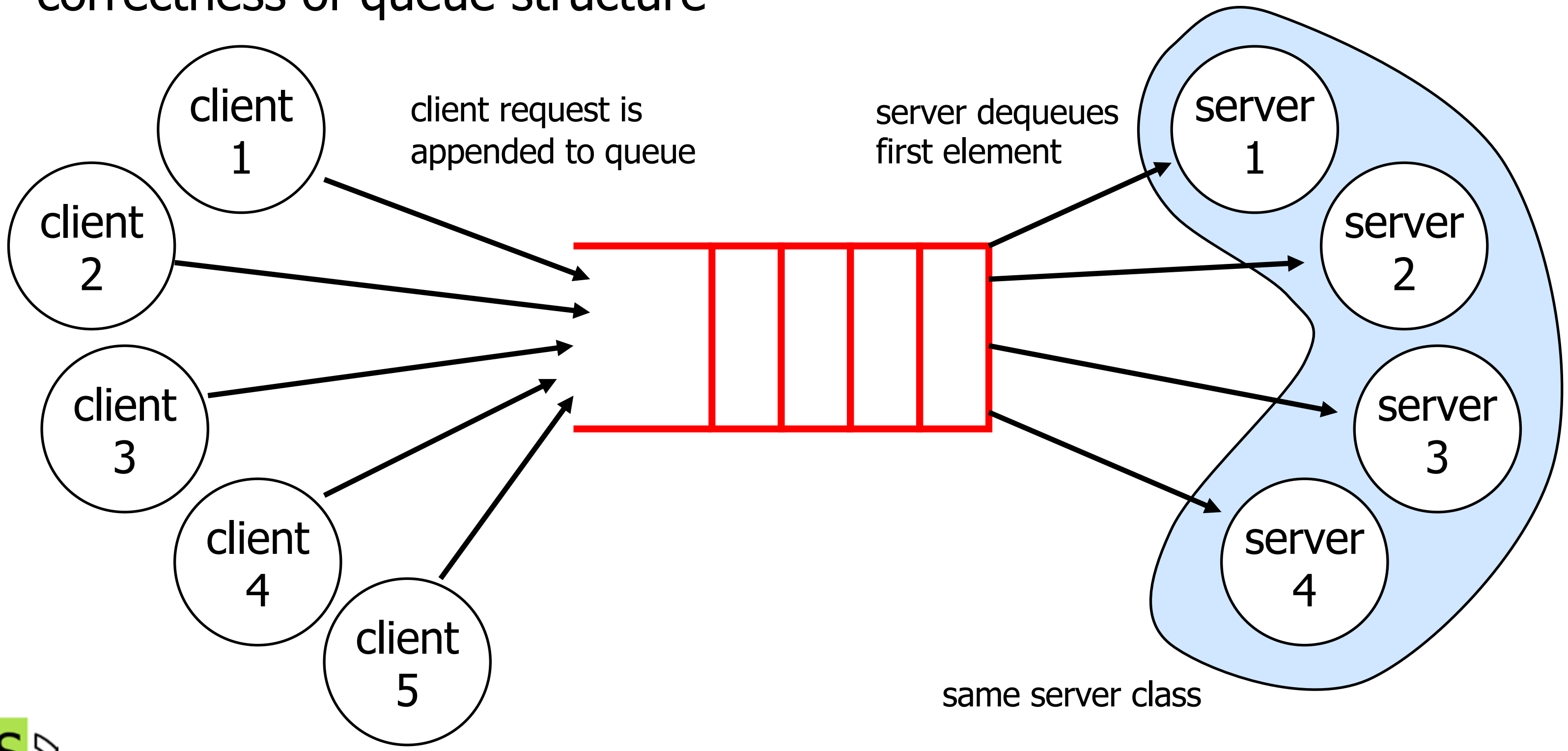
    client (front-end) cannot send request to sever, is blocked, or user may have to submit request later
  - client/communication failure

    server, after executing the transaction is unable to send reply to the client; reply may be lost; client (after restart) doesn‘t receive a response – request processed? should the client resubmit? (→ uncertain results)
  - load balancing for server pools is difficult
  - no scheduling/prioritization of requests



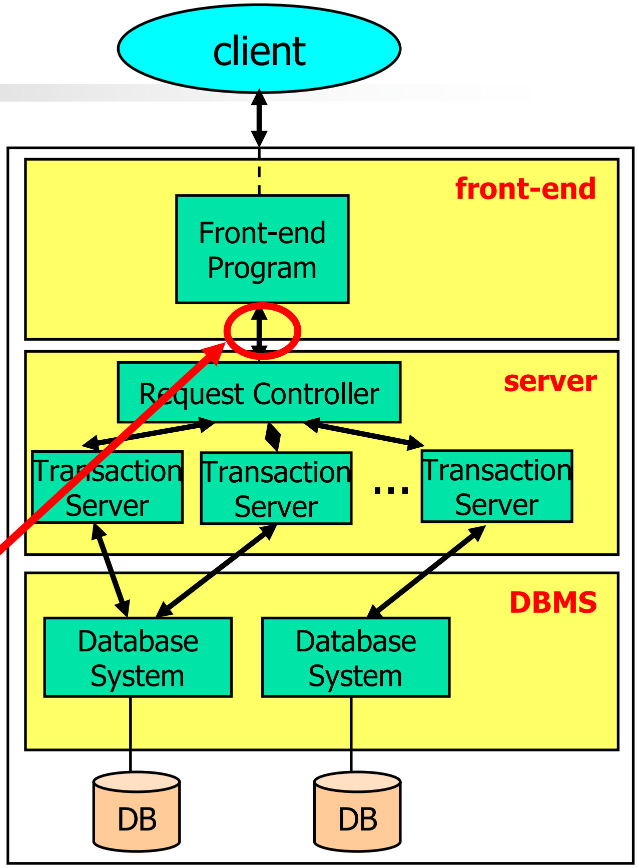
Short-term Queues for Load Control

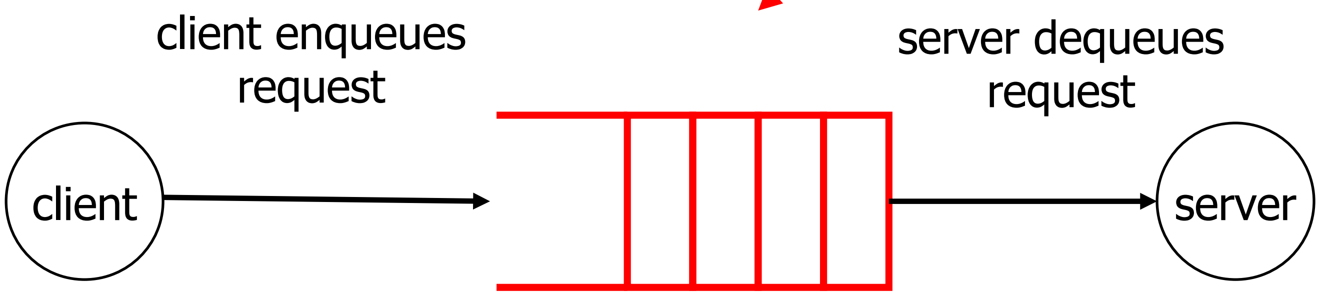
- Load control (during direct transaction processing)
  - Handle temporary load peaks
  - Store request in (temporary) internal queue to avoid creating new processes
  - Client-side model: direct, synchronous communication (queue is not visible!)
- "exactly-once" has to be guaranteed; concurrent access must preserve correctness of queue structure



Queued Transactions

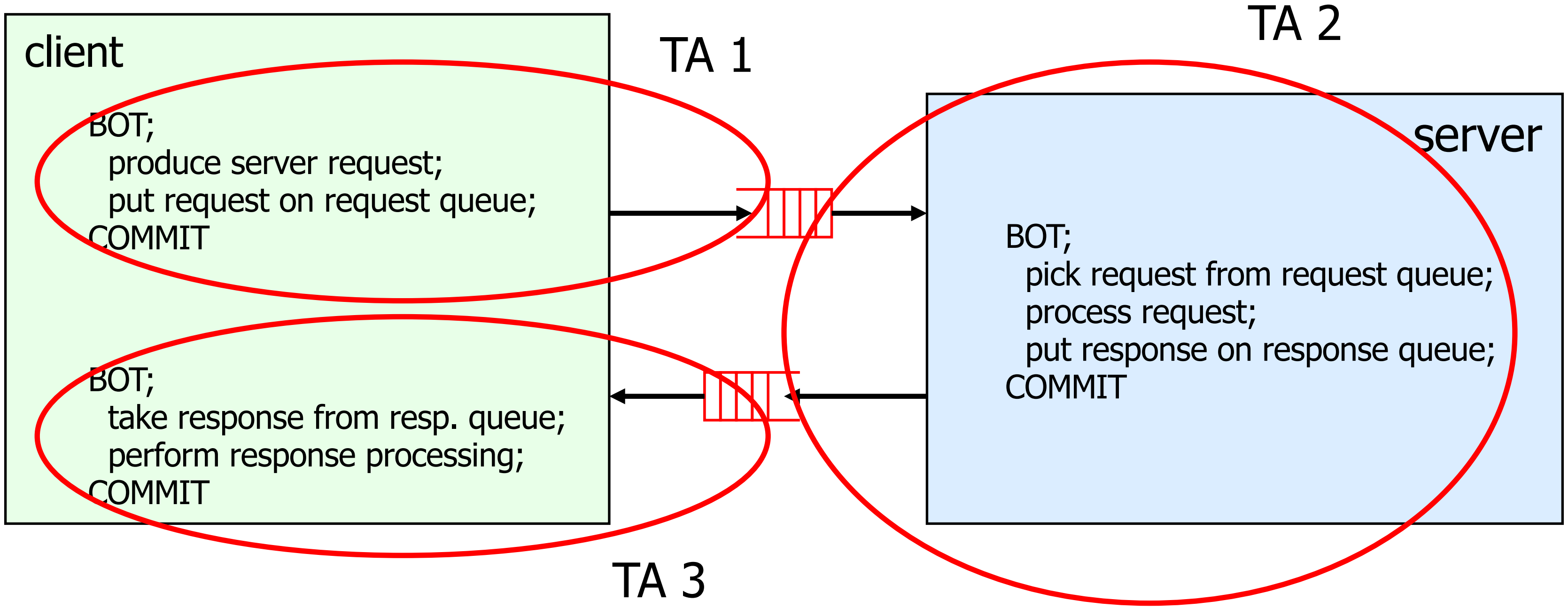
- Queues are used as buffers for requests and replies between client (e.g., front-end program) and server
  - clients send request to queues (“enqueue”)
  - servers receive requests from queues (“dequeue”)
  - replies are sent/received to/from queues as well





Asynchronous Transaction Processing

- Decoupling Request Entry, Request Processing, and Response Delivery, use separate TAs for each task
  - improved throughput
  - client/sever can submit request/reply even if the comm. partner is down
  - communication failures don’t cause uncertain results (exactly-once is guaranteed)



```
BOT;
  produce server request;
  put request on request queue;
COMMIT
```

```
BOT;
  take response from resp. queue;
  perform response processing;
COMMIT
```

```
BOT;
  pick request from request queue;
  process request;
  put response on response queue;
COMMIT
```

Queues for Asynchronous Transaction Processing

- Queues are persistent, transactional
  - distinguishable, stable objects
  - can be manipulated through ACID transactions
    - send, receive operations are part of the respective transactions
      - queuing system is yet another transactional resource manager
    - queue operations and operations on other RMs can happen within the same (distributed) transaction
    - request will become visible to other TAs only at commit of sending TA
    - if the receiving TA crashes, the request will be "put back" on the queue by the queuing system
      - server can re-process the request after recovery
- Client view
  - ACID request handling: request is executed exactly once
  - Request-reply matching: for each request there is a reply
    - request-id for relating requests and responses, provided by the client
  - At-least once response handling: client sees response at least once
    - response may have to be presented repeatedly, e.g., after client failure/restart

Client Recovery

- What if the client fails or loses connectivity, or the queue fails?
  - after recovery, the client needs to resynchronize with the queues
  - Note: we assume that the client only submits one request at a time
- Additional information required on the client to determine state and required action
  - request/reply pairs are uniquely, globally identified (ID)
  - IDs of last enqueued and last dequeued request/reply is stored persistently
    - either locally on the client
    - or by the queue manager as part of the (recoverable) session state
- Four possible states for client
  1. TA1 did not run and commit (lastEnqueuedID doesn't match last request)
     → client should resubmit the request
  2. TA1 committed, TA2 did not (lastEnqueuedID≠lastDequeuedID and no reply)
     → client waits for reply
  3. TA2 committed, TA3 did not (lastEnqueuedID≠lastDequeuedID and reply avail.)
     → client can process the reply using reply queue
  4. TA3 committed (lastEnqueuedID=lastDequeuedID)
     → client has processed the reply, can continue with new request

Handling Non-Undoable Operations

- Client processes the reply in a separate transaction (TA3)
  - what should the client do, if TA3 fails?
- Three cases to consider for the operation "process reply"
  - operation is undoable (e.g., a local DBS operation)
    → no problem, TA3 can be reexecuted
  - operation is idempotent, i.e., its repeated execution produces the same state
    - example: display reply, print receipt, send success email including unique ID

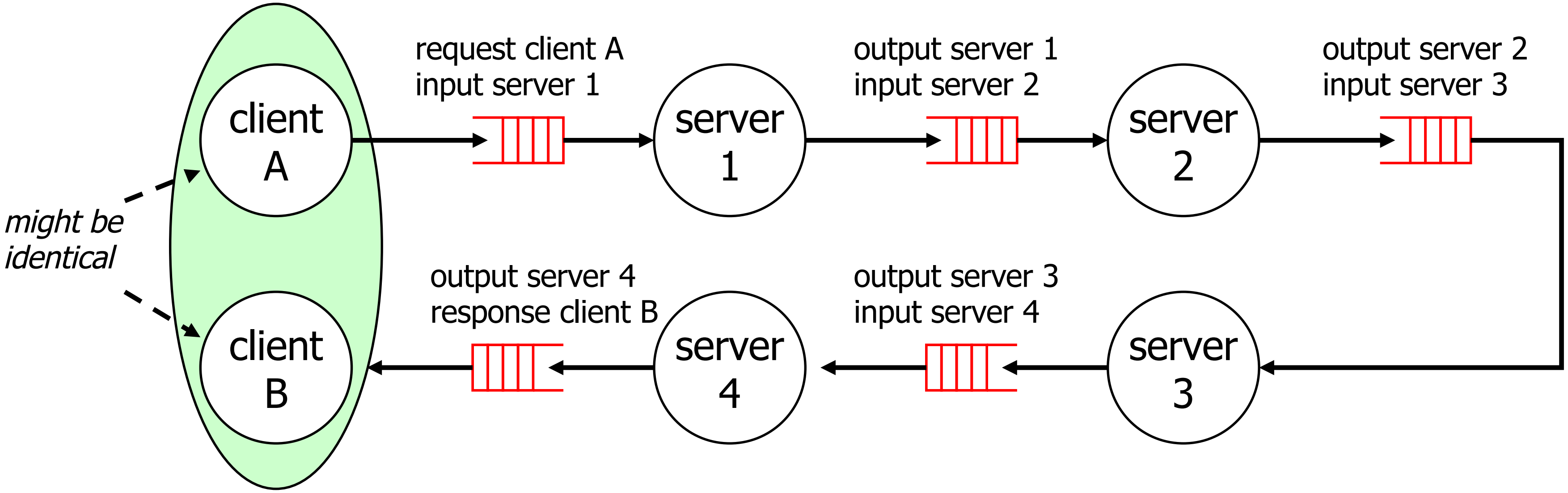
    → no problem, TA3 can be reexecuted
  - operation is neither undoable nor idempotent
    - typically involves a physical device doing some work (e.g., ticket printer, cash machine)
    - if performed before commit, the operation cannot be undone and reexecuting the TA is problematic
    - if delayed after commit, operation may be lost do to a failure after TA3 commits
    - can be solved, if the device has testable state, i.e., the device changes its state when operation is performed (example: ticket ID of last ticket printed)
    - processing the reply then should involve logging current device state on persistent store along with the reply ID (independent of the outcome of TA3) and comparing the logged state with the current device state, when reprocessing a reply

Persistent Queues in TP-Monitors

- End-user control
  - Delivering output (e.g., display information, print ticket, hand out money) is a critical step in asynchronous processing
  - Redelivery may be required until user explicitly acknowledges receipt
- Recoverable data entry
  - Some applications are driven by data entry at a high rate, without feedback to the data source
  - Optimize for high throughput (instead of short response times)
  - Input data are taken from queue by running application
  - Input data must not be lost, even during a crash
- Multi-transactional requests
  - Single request is processed in multiple transactions

Multi-transactional Requests

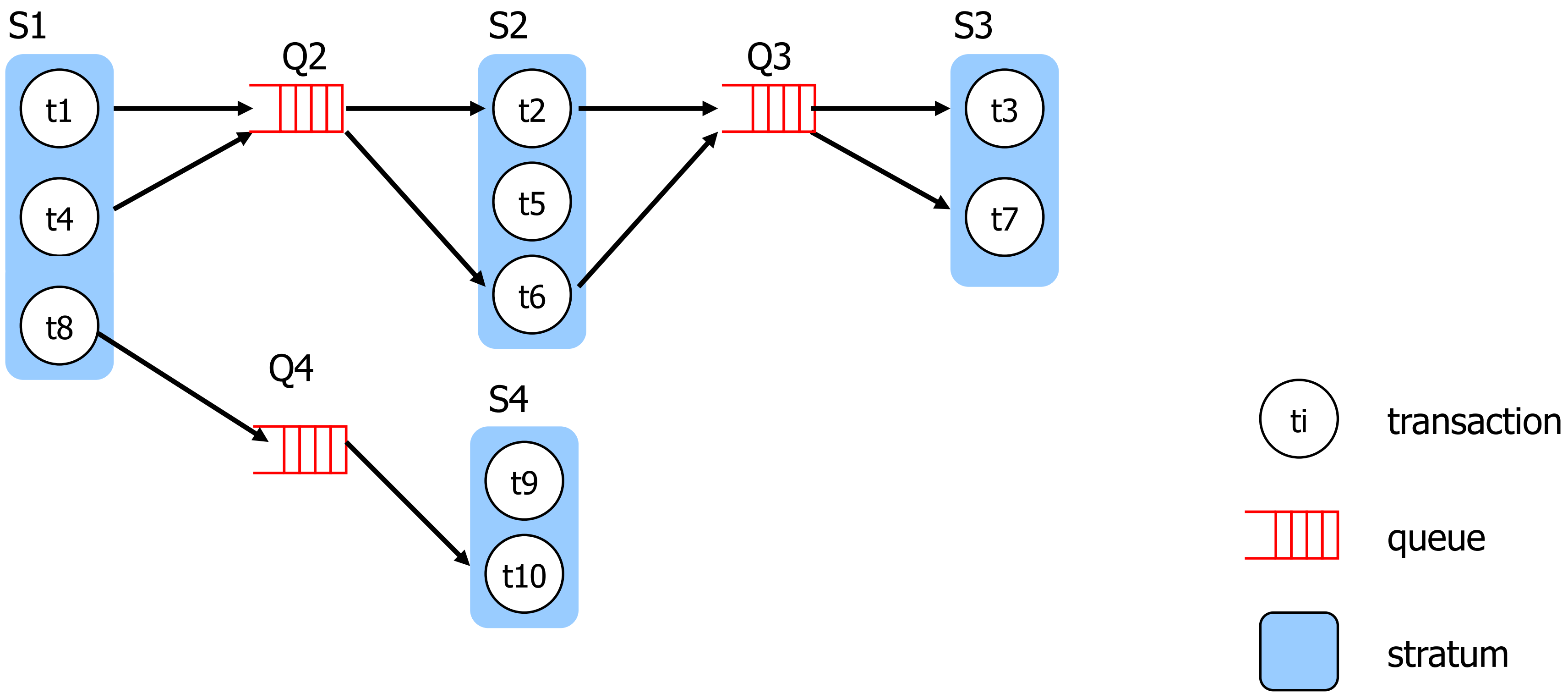
- Single request processed in a sequence of multiple transactions
  - can be scheduled asynchronously for high throughput, as long as no intermediate user interactions are required
  - avoid resource contention of single-transaction (TRPC) approach
- Based on recoverable input data (persistent queues)



- Assumption: each transaction in the sequence will finally commit
- Complete transaction sequence is no longer serializable

Stratified Transactions

- Generalization of multi-transactional requests
  - Stratum: set of transactions to be coordinated under 2PC
    - connected through message queues
  - Connected strata form a tree structure



Stratified Transactions (2)

- Structure
  - some t_i should commit at the same time
  - disjoint, complete partitioning of T into sets of transactions S_1, … S_m
    $S_i \subseteq T$ with $S_i \neq \varnothing$ and $S_i \cap S_j = \varnothing$ for $i \neq j$ and $\cup S_i = T$
  - transactions in S_i are synchronized by 2PC
  - set of transactions S_i is called stratum
  - each S_i receives requests in a request queue Q_i
  - a queue Q_i does NOT associate more than 2 S_i
- Behavior
  - requests for stratum is only visible in input queue, if parent stratum commits
    - queues are transactional
  - all strata eventually commit if their respective parent stratum commits
    - stratified TA commits if root stratum commits
  - if stratum fails repeatedly, then this is an exception that requires manual intervention, compensation

Stratified Transactions (3)

- Advantages compared to single, global TA for T:
  - early commit of individual strata; implies less resource contention, higher throughput
  - reduced observed end user response time (commit of root stratum)
  - if all transactions in a stratum execute on the same node:
    - no network traffic for executing 2PC
    - TA-Managers coordinating global TA on respective nodes don't need to support external coordinator
- Requirements
  - all resources manipulated by transactions (including messages) need to be recoverable
  - resource managers need to be able to participate in 2PC

Client Variations

- Non-transactional client
  - transaction support may not be available on the client
  - client still needs to be implemented in a fault-tolerant manner
    - make sure that the same request is not sent more than once
    - make sure that replies are delivered to the end user (at least) once
  - queuing infrastructure can help by
    - guaranteeing that message is stably stored when "enqueue message" operation returns to client
    - providing information (message-ids) about the last request submitted, last reply received when client reconnects after failure
    - allowing a client to
      - explicitly acknowledge receiving a reply
      - re-receive the unacknowledged replies
        - reply is deleted only when explicitly or implicitly acknowledged by the client
- One-way messaging
  - client requires no reply for a request
- Multiple clients submitting requests to the same queue
  - client-specific reply queue can be identified as part of the request info

Message Queuing Systems (MQS)

- Have evolved from queuing systems in TP-monitors
- Message-oriented interoperability
  - programming model: message exchange
- Loosely-coupled systems/components
  - "client" is not blocked during request processing
  - "server"
    - can flexibly chose processing time
    - can release resources/locks early
  - components don't need to be running/active at the same time
- Provide persistent message queues
  - reliable message buffer for asynchronous communication
  - "store and forward" behavior
- Transactional MQS ("reliable MQS")
  - persistent MQS
  - guaranteed "exactly-once" semantics
  - transactional enqueue/dequeue operations

Interacting with MQS

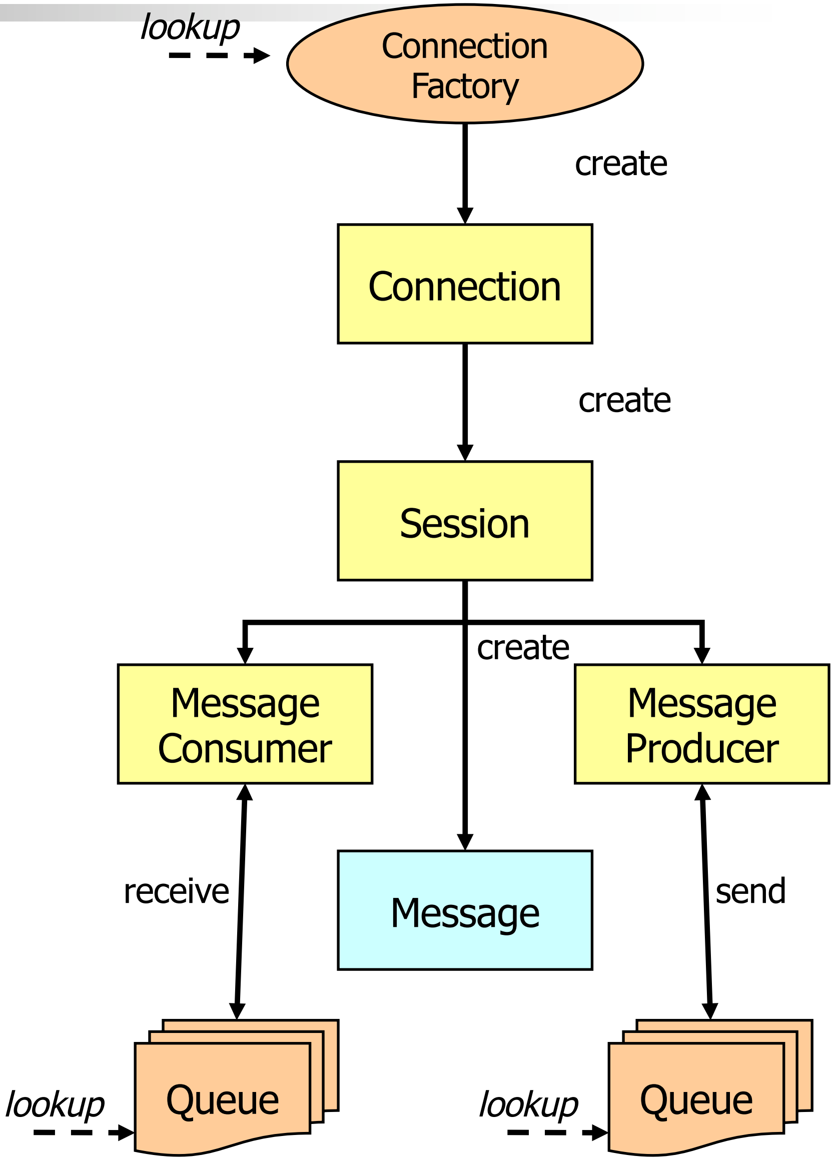
- Point-to-point messaging
  - Application explicitly interacts with message queues
  - Request/reply model needs to be built "on top"
- Basic operations:
  - Connect/Disconnect to/from MQS
  - Send or Enqueue: appends a message to a MQ
    - usually multiple producers can send/enqueue in the same queue associated with receiver
  - Receive or Dequeue: reads and removes message from a (its) MQ
- Variations
  - Shared Queues
    - support for multiple consumers per queue
      - example: load balancing by using multiple "server" components
      - but a particular message only has a single consumer
  - Additional properties for messages
    - priority, time-out, …
  - Enhanced flexibility for "receive"
    - beyond FIFO

Additional MQS Capabilities

- Filter criteria
  - dequeue messages based on their content
  - example: "importance" = "high"
  - based on message structure (header fields, properties, message body)
- Non-transactional queuing
  - execute some queue operation outside the scope of a transaction or in an independent transaction
  - example: report requests with incorrect password, even if TA fails
- Journaling
  - save a description of all operations on messages in a journal
  - keep history for auditing, etc.
- Queue management
  - start and stop a queue
  - disable enqueue or dequeue operations independently

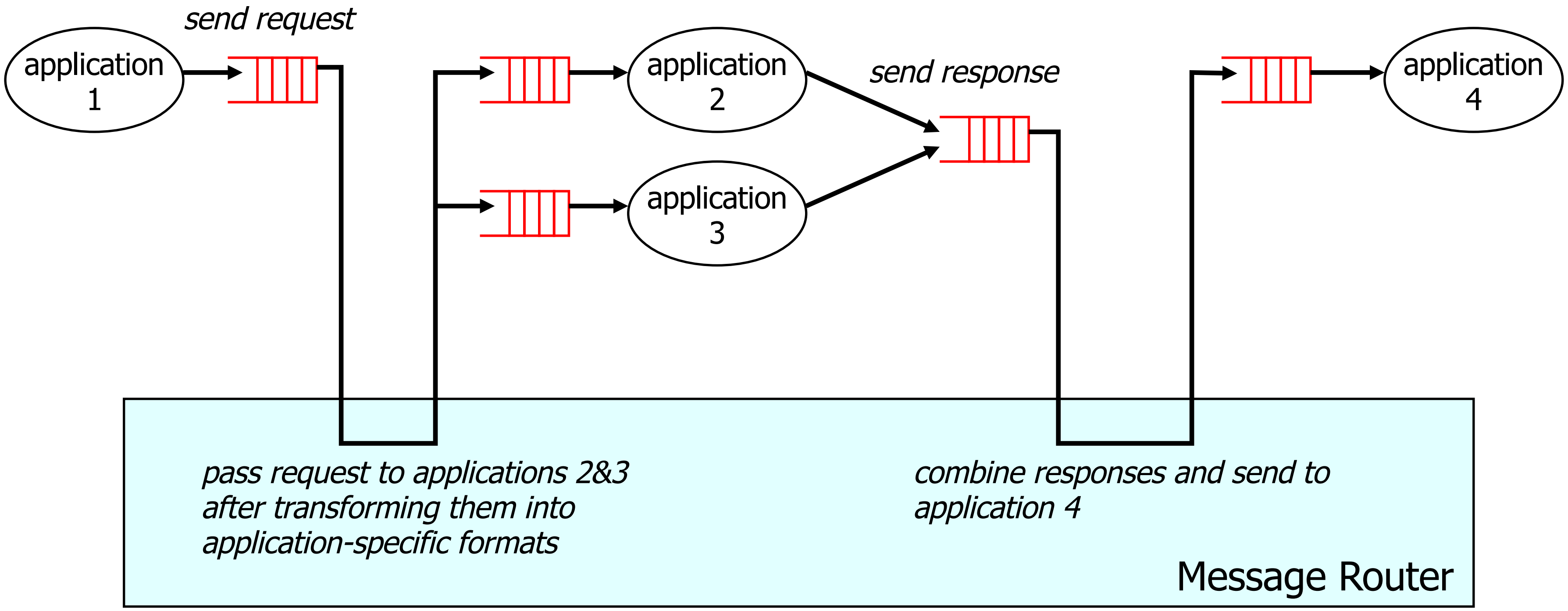
JMS – Standardized Interaction with MQS

- Administered objects
  - connection factories (contain provider details)
  - (static) destinations/message queues
  - registered/bound through JNDI
- Connection
  - represents connection to the JMS provider
  - start/stop messaging service
- Session
  - execution context for sending and receiving messages by creating messages, producers, consumers
  - may encompass a sequence of transactions
- Message
- Message producer
  - sends messages to queue
- Message consumer
  - receives messages from queue
    - synchronous receive( )
    - asynchronous using onMessage( ) method of Message Listener



Message Routing

- Idea: separate the routing and transformation logic from the applications
  - script defines sequences of application invocations and message transformation steps
    - transformations are program components invoked by the message router


send request
application 1
application 2
application 3
application 4
send response
pass request to applications 2&3 after transforming them into application-specific formats
combine responses and send to application 4
Message Router

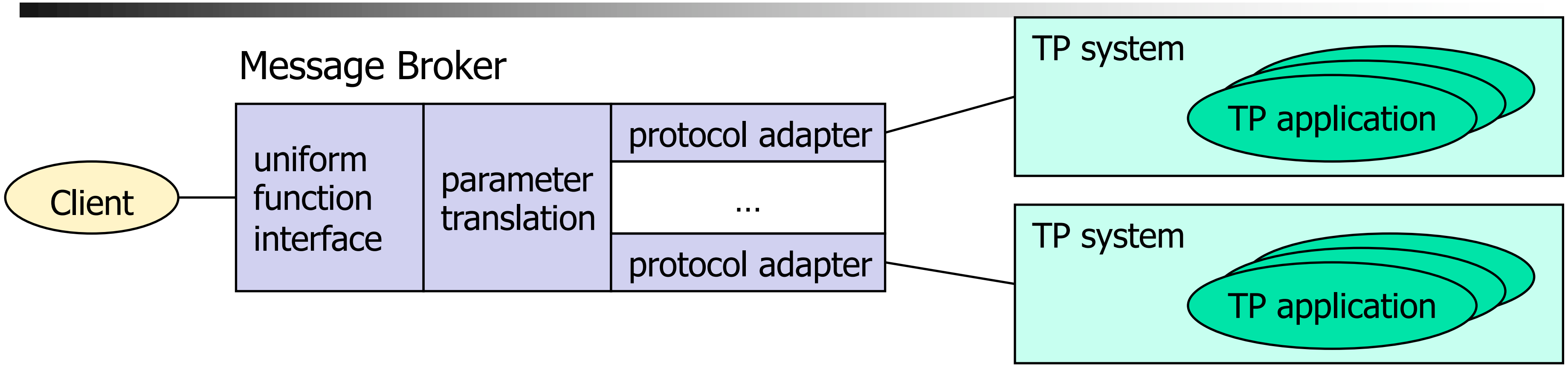
Publish/Subscribe Paradigm

- Publish and Subscribe
  - further generalizes message routing aspects
  - applications may simply publish a message by submitting it to the MOM (also called *message broker* here)
  - interested applications subscribe to messages of a given type/topic
  - message broker delivers copies of messages to all interested subscribers
- Subscription
  - can be static (fixed at deployment/configuration time) or dynamic (by application at run-time)
  - type-based subscription
    - based on defined message types
      - type namespace may be flat or hierarchical (e.g., SupplyChain.newPurchaseOrder)
    - also identified by the publisher
  - parameter-based subscription
    - boolean subscription condition identifying the messages a subscriber is interested in
      - example: type = "new PO" AND customer = "ACME" AND quantity > 1000
    - condition refers to message fields
  - non-durable subscription: published messages are not delivered if the subscriber is not active
  - durable subscription: messages are delivered until subscription expires
- JMS supports Publish/Subscribe
  - Publishers send messages to topics instead of queues
  - Subscribers create a special kind of receiver (topic subscriber) for a topic

EJB Message-Driven Beans (MDB)

- Entity and session beans can use JMS to send asynchronous messages
  - receiving messages would be difficult, requires explicit client invocation to invoke a bean method "listening" on a queue
    - may block the thread until message becomes available
- Message-driven beans should be used to receive and process messages
  - implement a message listener interface ("onMessage(…)")
  - stateless: no conversational state, can be pooled like stateless session beans
  - not invocable through RMI: don't have component interfaces (home, remote)
  - concurrent processing of messages
    - container can execute multiple instances, handles multi-threading
- Deployment
  - CMT for MDBs: only REQUIRED and NOT_SUPPORTED is permitted
  - descriptor includes additional attributes mapping to JMS processing properties
    - acknowledge-mode
    - message-selector
  - the queue from which a MDB should receive messages is fixed at deployment time

Message Broker Architecture



- Main focus is on Enterprise Application Integration (EAI)
  - bridge between heterogeneous applications
  - client submits a message to the broker, instead of calling an application directly
- Typical functions of a message broker
  - communication protocol support (required to communicate with applications)
  - unifies functional interface of applications based on canonical message format
    - broker stores a mapping to translate function invocations to the required form
    - may be implemented as a set of protocol adapters
  - tools for translating between different parameter and message formats
    - calculations, translation tables, DB table lookup, document translation (XML, XSLT)
  - routing, logging, auditing, performance monitoring, …

Databases and Messaging Systems

- Roles of DBMS in a messaging world
  - persistence manager for messaging systems
    - store/retrieve messaging data and state information
    - reliable, transactional
  - provide advanced DBMS capabilities to achieve a DBMS/MQS synergy
    - querying messaging data

S. Doraiswamy, M. Altinel, L. Shrinivas, S.L. Palmer, F.N. Parr, B. Reinwald, C. Mohan: Reweaving the Tapestry: Integrating Database and Messaging Systems in the Wake of New Middleware Technologies, in T. Härder, W. Lehner (Eds.): Data Management in a Connected World, LNCS 3551, Springer 2005: 91-110

Database as a Message Store

- Database serves as a backing store
- Messaging systems can exploit integral database features, such as
  - storage definition, management, and underlying media/fabric exploitation
    - single DB table for storing similar messages of a single/few queues
    - administrator can configure the tables appropriately
  - buffer, cache, spill management
    - DB cache allows for quick access during timely message consumption
  - index creation, management, reorganization
    - on (unique) message ids, sequence numbers, subscription topics, …
  - latching and lock management
    - avoid consumer/producers blocking on each other
    - row-level locking
    - lower isolation semantics (skip over locked messages, etc.)
  - transaction management and coordination
    - synchronous or asynchronous message store/commit in local TAs, based on QoS requirements
    - global TA support
  - high-speed and scalable logging services

Improved Database and Messaging Synergy

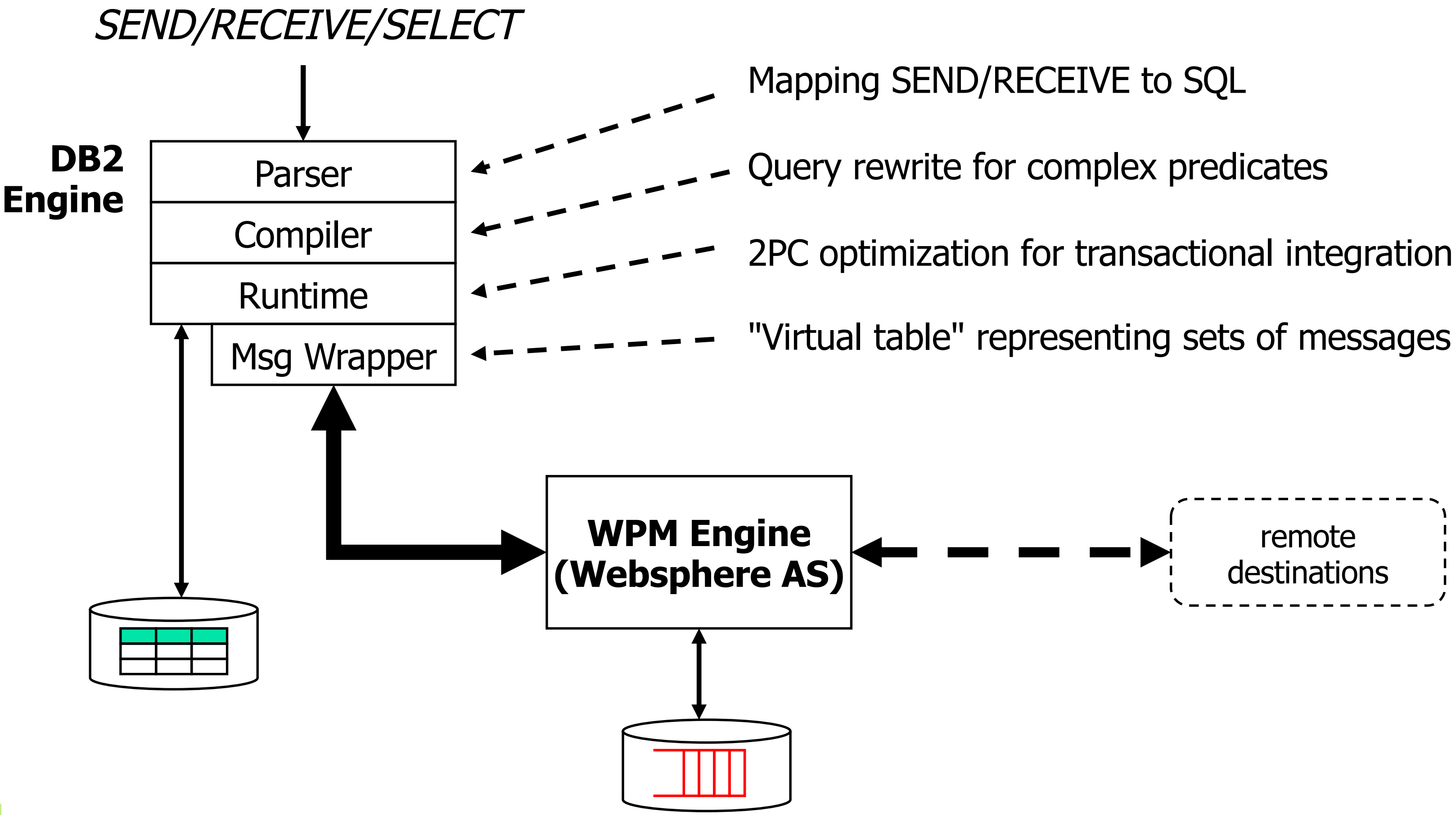
- DBMS helps accessing messaging data and destinations, possibly in combination with operational data
  - requires closer cooperation in terms of message schema and typing information
- Potential DBMS features
  - mapping message payloads structure to table structure
    - exploit object-relational and XML data capabilities of DBMS
  - message warehousing and replay functionality
    - tracking and analysis of message data
  - enabling the database for asynchronous operations
    - messaging triggers
  - use of SQL, SQL/XML, XQuery with MQS
  - publishing to message destinations as reaction to updates
    - triggers, messaging functions
    - replication
  - storing durable subscriptions
  - consume-with-wait support
    - instead of continued polling

Integration Strategies

- Database System using/integrating messaging capabilities
    - database-specific messaging and queuing
        - queuing support added to the DBMS engine
    - interfacing with message engines
        - "light integration"
        - messaging data lives in DBMS, new built-in or user-defined routines to interface with a (co-located) messaging system
- Messaging system using/integrating DBMS
    - message-system-specific persistence, transactions, logging
        - messaging engine implements all of the above by itself
    - database as a persistent message store
- Integrated Database Messaging
    - leverage the strengths of both DBMS and MQS, without reimplementation
    - potentially utilize additional middleware to achieve the integration
        - example: leverage (information integration) wrapper technology

Integrated Database Messaging – Example

- IBM research prototype based on DB2 Universal Database, WebSphere Platform Messaging (WPM)



SEND/RECEIVE/SELECT
DB2 Engine
Parser
Compiler
Runtime
Msg Wrapper
Mapping SEND/RECEIVE to SQL
Query rewrite for complex predicates
2PC optimization for transactional integration
"Virtual table" representing sets of messages
WPM Engine (Websphere AS)
remote destinations

Summary

- Queued transaction processing offers some advantages over direct TP
  - client/server can submit request/reply even if server/client is down
  - communication failure doesn't result in lost updates, uncertain results
  - easy load balancing across multiple servers
  - priority-based scheduling of requests
- Message Queues
  - persistent and transactional message queues
  - asynchronous transaction processing, multi-step TAs, stratified TAs
  - message routing, pub/sub
  - supported by TP monitors, application servers, message queuing systems
- Databases and Messaging Systems
  - database as a message store
  - DBMS/MQS synergy
  - different integration strategies
    - DBMS-extension, MQS-extension, integration