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

Chapter 5
Application Server Middleware

Outline

- Transaction processing application structure & architecture
- Application server middleware
  - tasks
  - types of middleware
- Transaction support in application servers
- Shared state
- Managing processes and threads
- Scalability
- Summary

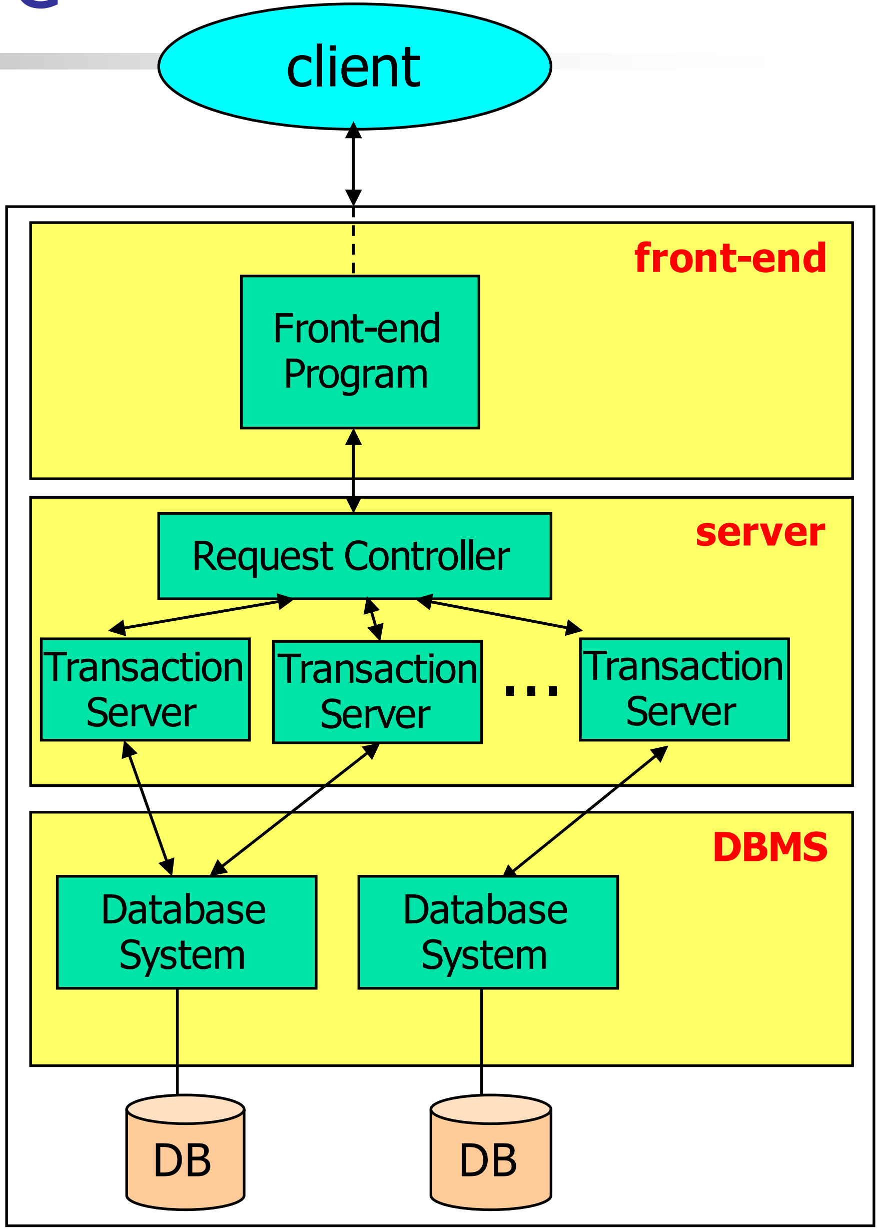
Transaction Processing (TP) Application

- End user‘s point of view
  - serial processor of requests
  - server that repeatedly
    - receives a request, does the requested work, optionally sends a reply
    - all within an ACID transaction
- Actual control flow within a TP application
  - the front-end program
    - captures the user’s request
    - translates the input into a request message
    - sends the message to a server
  - the server
    - examines the request message
    - determines what type of (business) transaction is requested and the program to process it
    - starts a transaction and invokes the program
    - the program executes the request, which typically involves invoking a DBS
    - the server commits the transaction (or aborts it in case the program was successful)
    - the server sends some output back to the source of the request

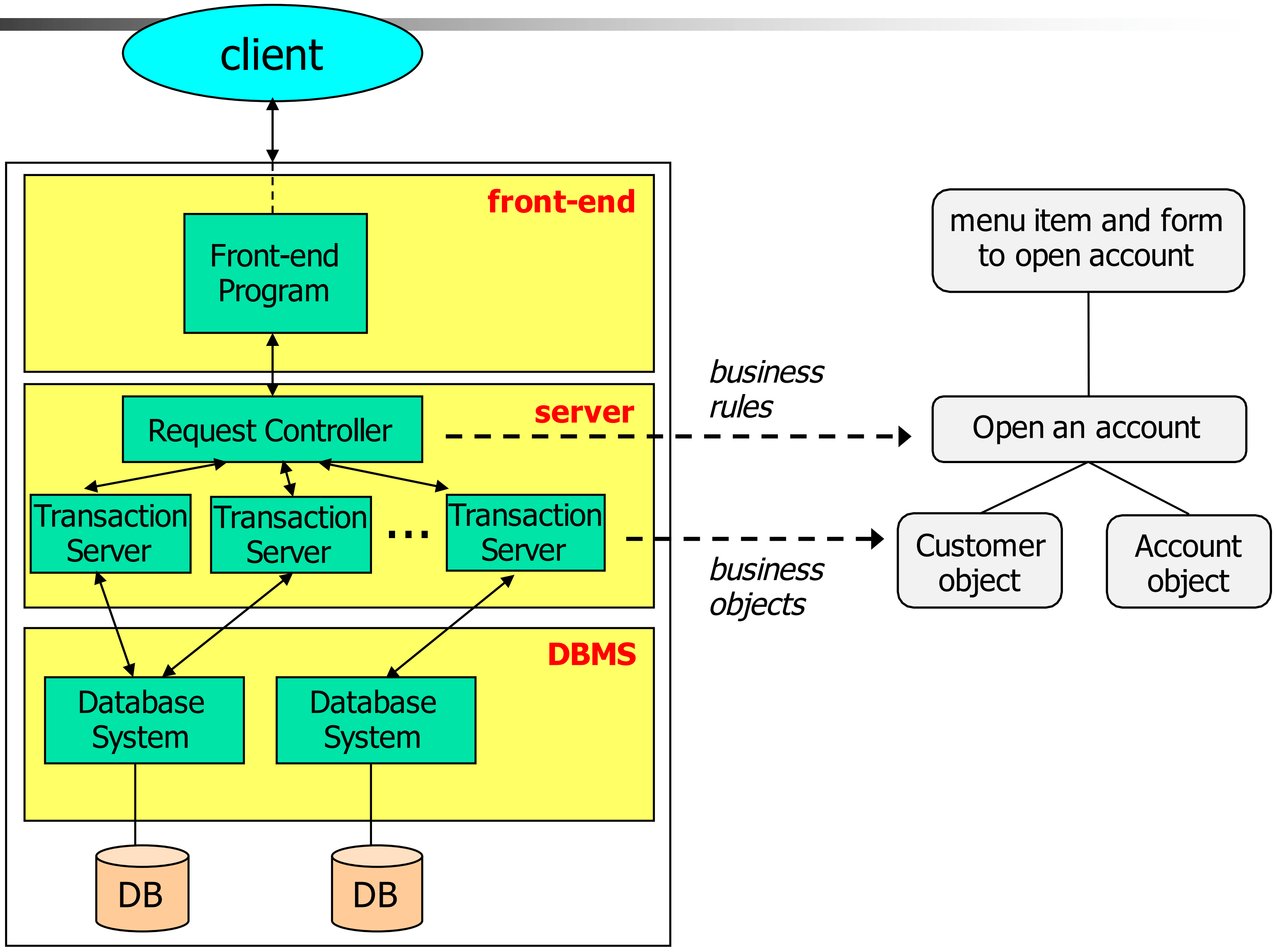
TP Application Architecture

- Front-end program
  - interacts with (possibly wide range of) display devices
  - gathers and validates input, displays output
  - constructs and forward request (e.g., as a RPC or asynchronous message)

  ➔ provides **device-independence** for server
- Request controller
  - guides the request execution
  - determines required steps, then executes them by invoking transaction servers
  - usually runs as part of an ACID transaction
- Transaction server
  - process that runs application programs doing the actual work of the request
  - almost always runs within the scope of an ACID transaction
  - typically interacts with a DBMS
  - simple applications can be composed into more complex ones (using local proc. call, TRPC, asynch. messaging, …)
    - makes difference to req. controller fuzzy



Object-Oriented Application Architecture



Application Server Middleware Tasks

- Distributed computing infrastructure (RPC, RMI)
- Transactional capabilities
  - transactional RPC/RMI
  - programming abstractions (demarcation)
  - distributed transaction management
- Scalable and efficient application processing
  - large number of client applications or end users
- Unified access to heterogeneous information sources and application systems
- Security services
  - authentication, authorization, secure transmission, …
- Reliability, high availability

Programming model abstractions that allow the developer to focus on application logic (i.e., ignore infrastructure as much as possible)

Types of Application Server Middleware

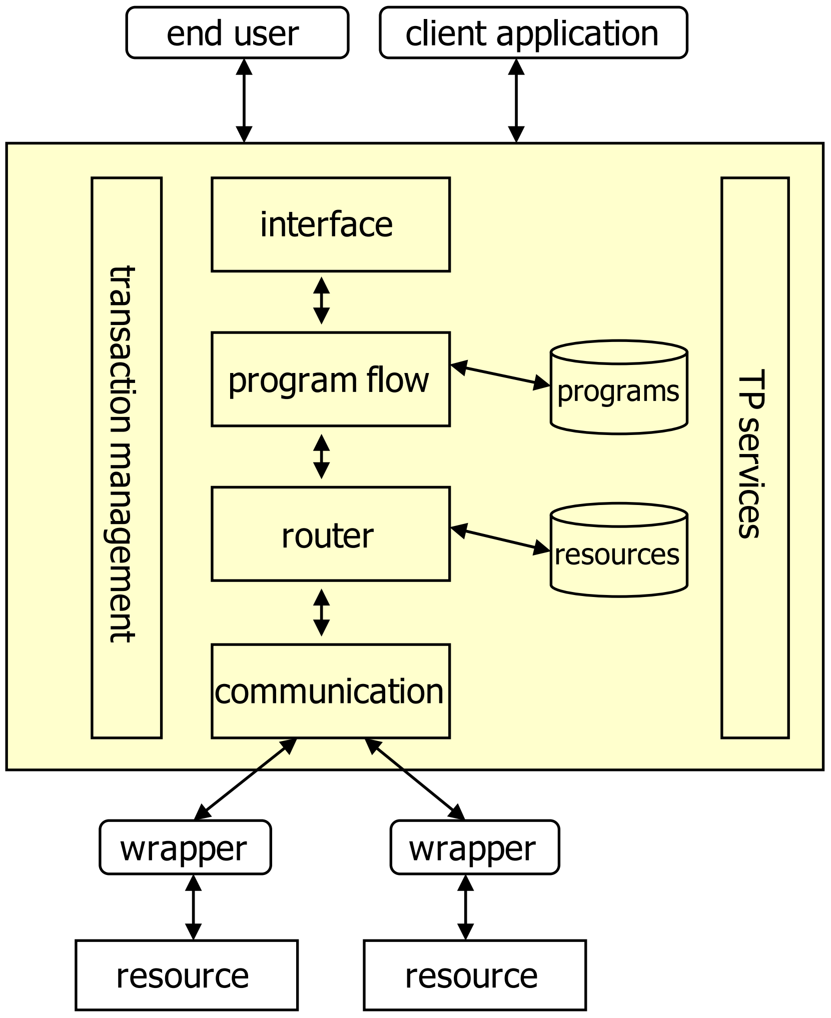
- TP monitor
  - transaction management, TRPC
  - process management
  - broad set of capabilities
- Object broker (e.g., CORBA)
  - distributed object computing, RMI
  - additional services
- Object transaction monitor
  - … = TP monitor + object broker
  - most often: TP monitor extended with object-oriented (object broker) interfaces
- Component Transaction Monitor
  - … = TP monitor + distributed objects + server-side component model

TP Monitor

- Provides functionality to develop, run, manage, and maintain transactional distributed IS
  - transaction management
  - (T)RPC
- Additional capabilities (beyond TRPC)
  - process management
  - high number of connected clients/terminals (10^2 - 10^4)
  - concurrent execution of functions
  - access shared data
    - most current, consistent, secure
  - high availability
    - short response times
    - fault tolerance
  - flexible load balancing
  - administrative functions
    - installation, management, performance monitoring and tuning
- One of the oldest form of middleware
  - proven, mature technology

Basic Components of a TP Monitor

- Interface
  - programs and terminals
- Program flow
  - store, load, execute procedures
- Router
  - maps logical resource operations to physical resources (e.g., DBMS)
- Communication manager
  - infrastructure for communicating with resources
- Transaction manager
- Wrappers
  - hide heterogeneity of resources
- Services
  - security, performance management, high availability, robustness to failures, …

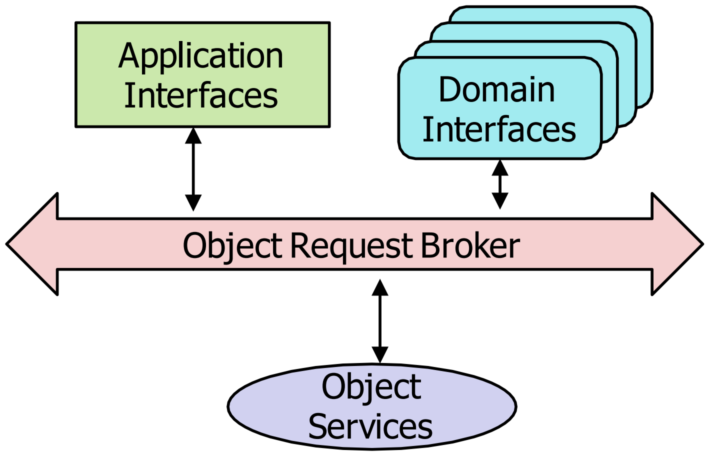


Transactional Services

- Need to strictly distinguish TP monitor and TA manager functionality
  - many users/applications don't need a TP monitor: batch applications, ad-hoc query processing
  - special application systems (e.g., CAD) have their own (terminal) environment
  - but all need transactional support
- Separation of components for
  - transactional control (TA manager)
  - transaction-oriented scheduling and management of resources (TP monitor)

Standard Object Broker - CORBA

- CORBA: **Common Object Request Broker Architecture**
- Object-oriented, universal middleware platform
  - object bus architecture based on RMI concept
    - CORBA IDL for interface definitions
  - language-independent and platform-independent
- Object Management Architecture (OMA)
  - Interfaces in different categories
    - Application Interfaces
    - Object Services (horizontal)
    - Domain Interfaces (vertical)
      - Telecommunication, Finance, E-Commerce, Medicine, ...
- ORB provides network communication
  - manages stubs (client-side)
  - maps RMI to object adapter (server side)
- Object adapter: Portable Object Adapter (POA)
  - generates object references
  - maps RMI to server objects
  - activates/deactivates/registers server objects
- ORB + object adapter = request broker



CORBA – Communication and Object Services

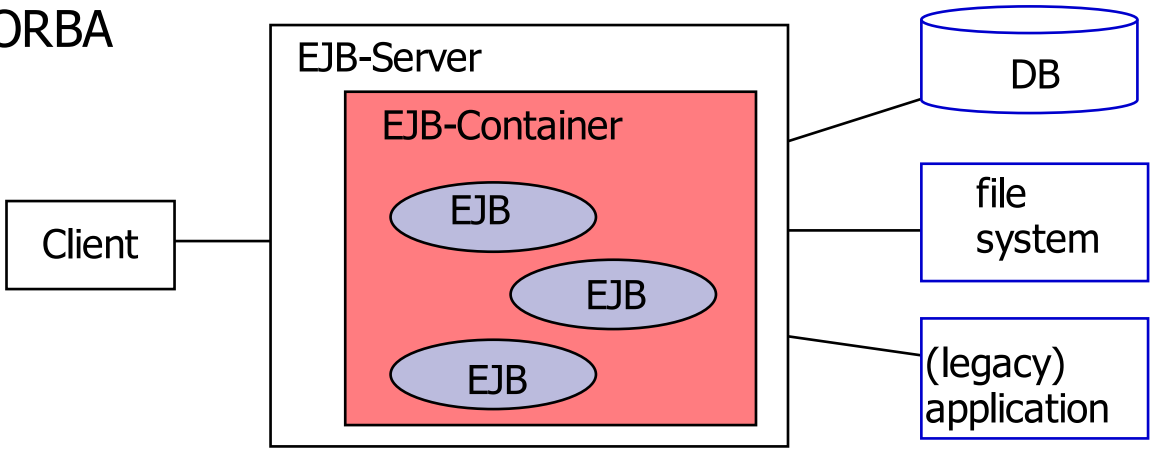
- Standardized data format and communication protocol(s)
  - *Common Data Representation* (CDR)
  - CORBA 2.0: *General Inter-ORB Protocol* (GIOP)
  - *Internet-Inter-ORB-Protocol* (IIOP)
    - maps GIOP to TCP/IP
    - internet as "Backbone-ORB"
  - optional: Environment-Specific Inter-ORB Protocols (ESIOP)
    - example: DCE Common Inter-ORB Protocol (DCE-CIOP)
- Object services extend basic ORB capabilities to provide system services
  - Naming, Life Cycle, Events, Persistence, Concurrency Control, Transaction, Relationship, Externalization, Query, Licensing, Properties, Time, Security, Trading, Collections
- Service usage
  - functionality defined using CORBA-IDL
  - CORBA object invokes method of service object (Example: NameService)
  - CORBA object *implements* interface provided as part of a service, may not need to provide any code (Example: TransactionalObject)

Server-side Component Models

- Problems with CORBA (up to 2.0)
  - complex, non-standard programming of server objects
    - service usage (transactions, security, ...)
      - behavior fixed at development time
    - resource management, load balancing
      - proprietary programming model and interfaces, is supported by object adapter
- Standardized Server-side component model
  - defines a set of "contracts" between component and component server for developing and packaging the component
  - developer focuses on application logic
    - service use can be defined at deployment time by configuring the application component
      - code generation as part of deployment step
    - resource management, load balancing realized by application server
      - component only has to fulfill certain implementation restrictions
  - server components are portable

Enterprise JavaBeans (EJBs)

- Standard server-side components in Java
  - encapsulates application logic
    - *business object components*
    - can be combined with presentation logic component models
      - servlets, JSPs
  - EJB container
    - run-time environment for EJB
      - provides services and execution context
    - *Bean-container-contract*
      - EJB implements call-back methods
- Interoperability with CORBA
  - invocation: RMI/IIOP
  - services



Deployment

- EJB is portable, server-independent
- Component properties
  - mapping of bean attributes to DB structures
  - configuration regarding transactional behavior
  - configuration of security aspects
- Specified using
  - source code annotations (specified at development time)
  - an XML deployment descriptor (customization at deployment time)
- What happens during deployment
  - generation of glue-code based on component properties
  - make classes and interfaces known
  - setting environment/context variables

Transactional Capabilities in Application Servers

- Support for transactions affects
  - the programming model
  - the API available to the programmer
  - system software components to support them
- Transaction demarcation (aka transaction bracketing)
  - Start, Commit, Abort
  - commands used by the programmer to define which operations (including RPCs) execute within the scope of the transaction
- Chained transactions
  - in this model, an application is assumed to be always executing within a TA
  - no need for an explicit start of a new transaction
  - application specifies the „boundary“ between transactions (aka syncpoint)

Transaction Composability Problem

- Two procedures (DebitChecking, Credit Checking) designed a independent transactions

```
DebitChecking(acct, amt) {
  start;
  update DB:
   acct:=acct-amt;
  if success commit;
  else abort;
}
```

```
CreditChecking(acct, amt) {
  start;
  update DB:
   acct:=acct+amt;
  if success commit;
  else abort;
}
```

- We want to call/reuse them in a new procedure (Transfer)
  - needs to execute in a single ACID TA, too!
  - but DebitChecking, Credit Checking start their own, separate TAs and commit independently

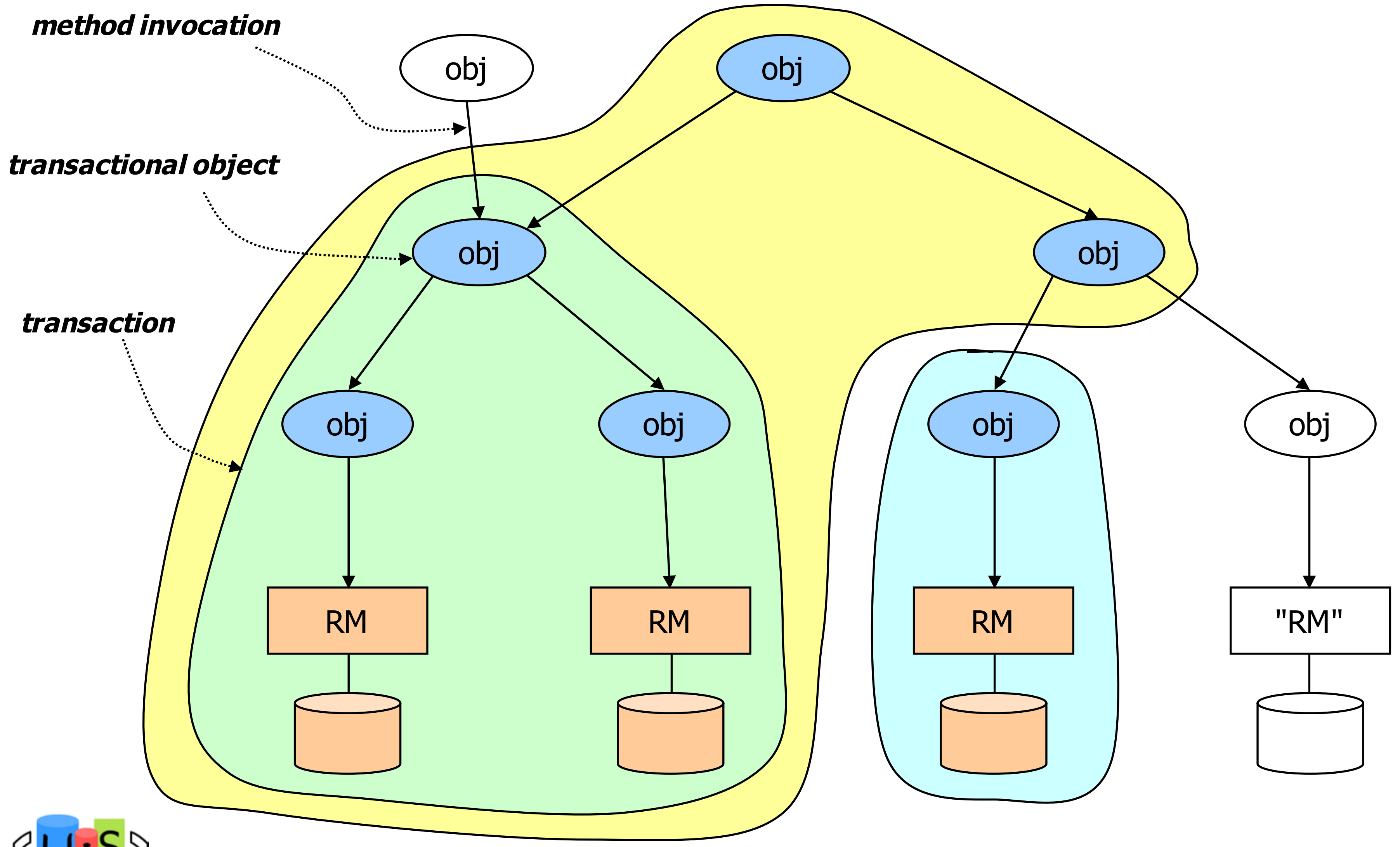
```
Transfer (acct1, acct2, amt) {
  start;
  DebitChecking(acct1, amt);
  if no success {abort; return;}
  CreditChecking(acct2, amt);
  if no success {abort; return;}
  commit;
}
```

We cannot compose DebitChecking and Credit Checking into a larger transaction!

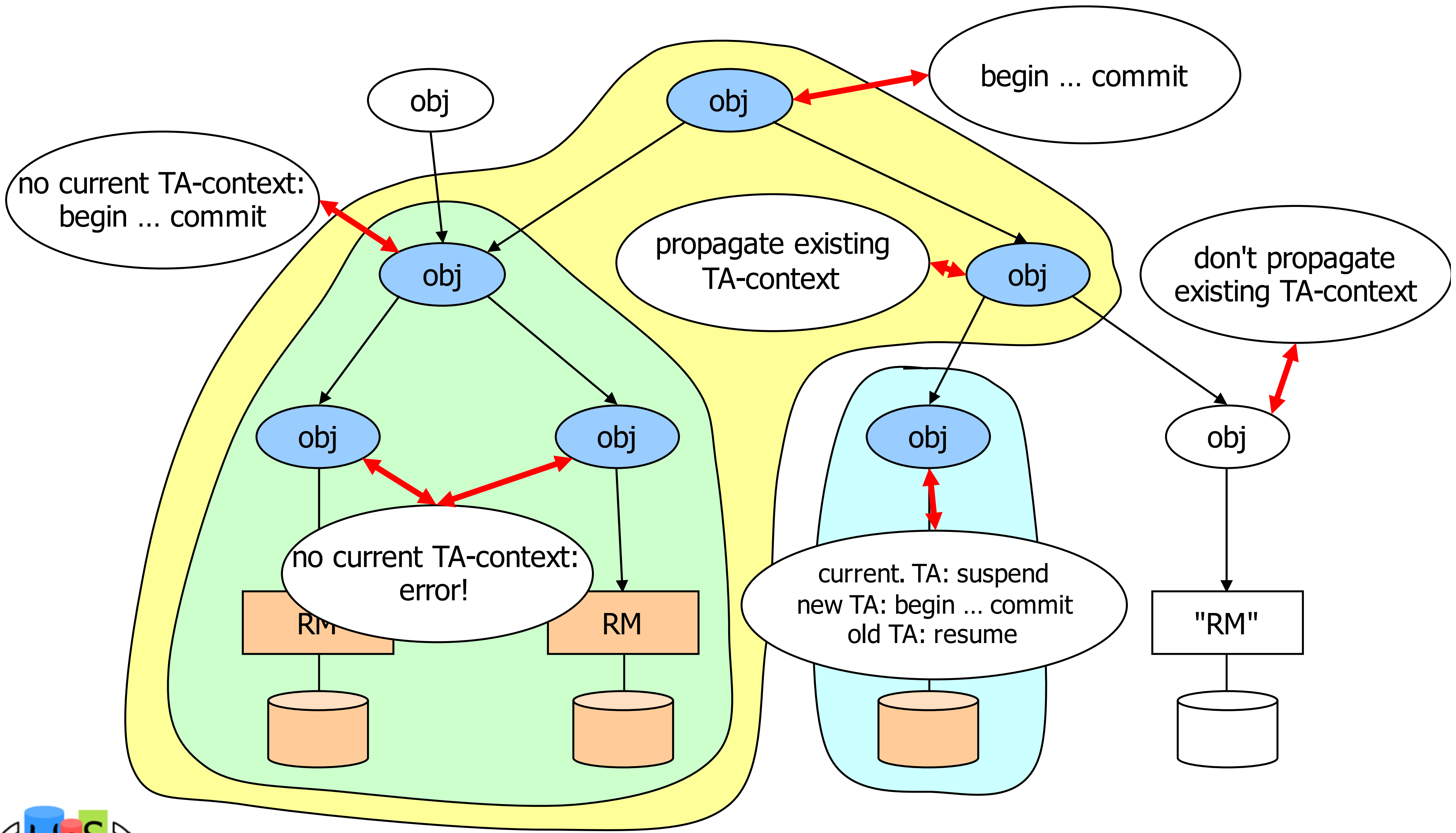
Possible Solutions

1. System ignores *start*, if the program is already running within a TA
   - System needs to keep a counter so that the corresponding *commit* is also ignored (a *start* increments the counter, a *commit* decrements it)
     - otherwise, *Transfer* starts the TA, and *DebitChecking* commits it too early
   - How are *abort* calls handled in programs for which *start* was ignored?
     - Example: *DebitChecking* aborts because of insufficient funds
     - Option 1: permit abort - complete transaction needs to abort
     - Option 2: disallow abort - treat as program error
     - Option 3: ignore abort – the calling program needs to handle the problem
2. Separate request processing code from transaction demarcation code
   - discipline:
     - request controller program always does the demarcation
     - transaction server program never does demarcation on its own
   - alternative: introduce *wrapper procedures* for (e.g., *CallDebitChecking*)
3. Support extended explicit demarcation API
   - additional calls for determining the transactional status, etc.
4. Implicit demarcation based on transaction attributes
5. Nested transaction programming model

Demarcation of Transactions



Transactional Behavior of Objects


begin … commit
obj
obj
no current TA-context: begin … commit
obj
propagate existing TA-context
obj
don't propagate existing TA-context
obj
obj
obj
obj
no current TA-context: error!
RM
RM
current. TA: suspend
new TA: begin … commit
old TA: resume
"RM"

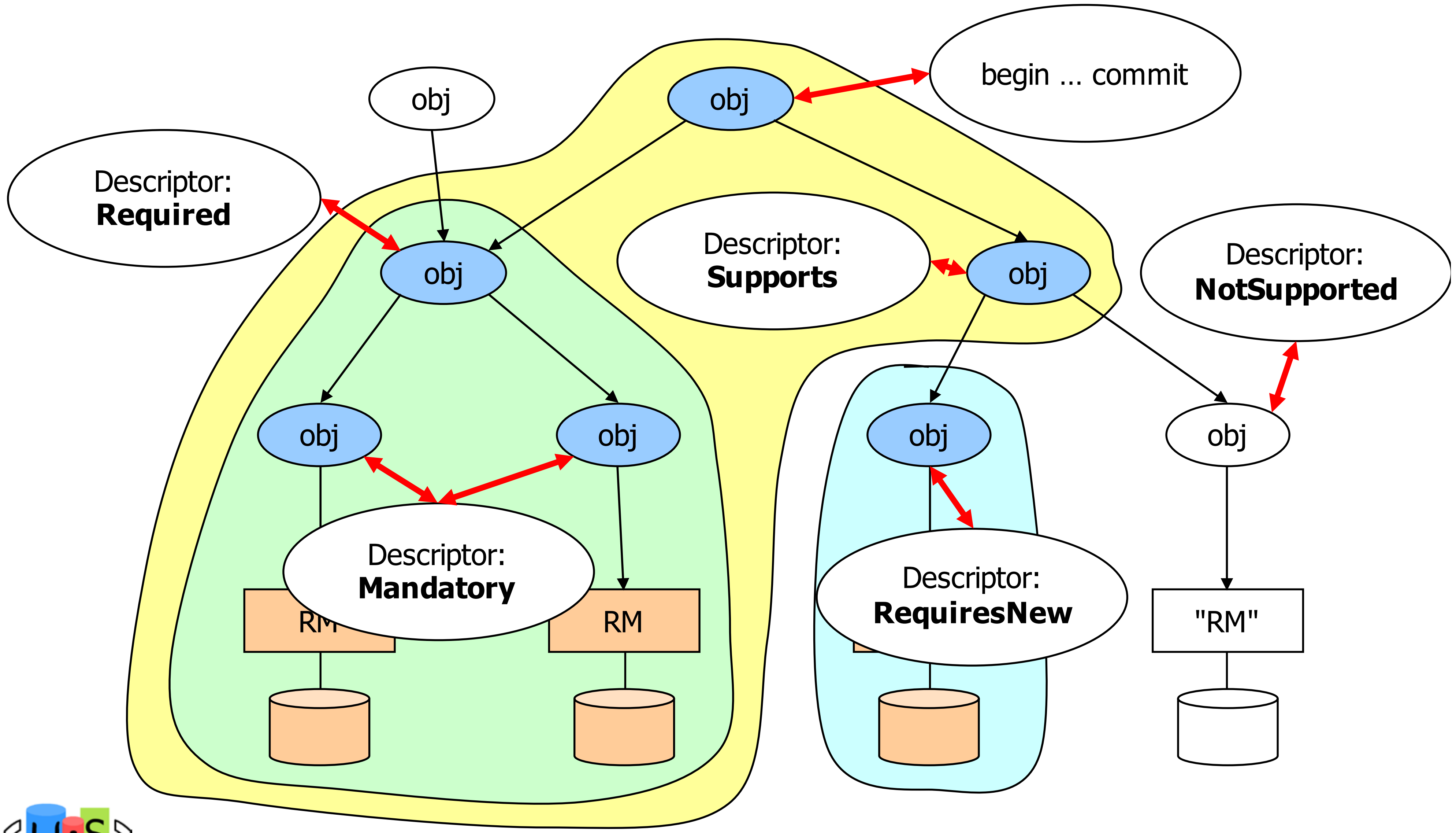
Transaction Demarcation Approaches

- Explicit (programmatic) demarcation
  - method interacts with TA manager using demarcation API
    - begin, commit, rollback
    - suspend, resume
  - management of transaction context (transaction ID)
    - direct: passed along as explicit method parameter
    - indirect (preferred!): a "current" TA context is propagated automatically
- Implicit (declarative) demarcation
  - separate specification of transactional properties for methods
    - can be realized/modified independent of application logic
    - may be deferred to deployment phase
  - application server (container) automatically performs TA demarcation before/after method is invoked
- Combination of both approaches in distributed IS

Explicit Demarcation (e.g., in Java)

- Java Transaction API (JTA) can be used by EJB Session Beans and EJB client, web components
  - in EJB this is called bean-managed transaction: in descriptor *transaction-type = Bean*
- Demarcation uses JTA UserTransaction
  - *begin()* – creates new TA, associated with *current thread*
  - *commit()* – ends TA, current thread no longer associated with a TA
  - *rollback()* – aborts TA
  - *setRollbackOnly()* – marks TA for later rollback
    - beans with implict TA-mgmnt can use method on *EJBContext*
  - *setTransactionTimeout(int seconds)* – sets timeout limit for TA
  - *getStatus()* – returns TA status information
    - active, marked rollback, no transaction, ...

Implicit (Declarative) Demarcation in EJB



Transactional Properties

- Transaction attributes for methods:

TA-Attribute	**Client-TA**	**TA in method**
Not Supported	none T1	none none
Supports	none T1	none T1
Required	none T1	T2 T1
RequiresNew	none T1	T2 T2
Mandatory	none T1	error! T1
Never	none T1	none error

- Example: EJB container-managed transactions
  - attributes are specified in deployment descriptor

Implicit Demarcation – Commit/Abort

- Method invocation may result in starting a (new) transaction
  - attribute = RequiresNew, or Required and caller is not executing in a transaction
  - we call this a top-level method invocation
- Top-level method may call other methods (called submethod)
  - execute in the same transaction, if attribute = Required, Mandatory, Supported
- Transaction
  - commits, if top-level method and all submethods terminate without an error
  - aborts, if top-level method or one of the submethods throws an exception
    - alternatively, submethod may explicitly indicate unsuccessful outcome (e.g., by calling the *setRollbackOnly* method)

Nested Transactions

- (Top-level) transactions can subtransactions nested inside them
  - same bracketing/demarcation operations are used to start a top-level/sub-TA
- Semantics
  1. Top-level TA: is created when program is not executing within a TA already and issues a start command
  2. Sub-TA: is created when program is already running within a (parent) TA and issues a start command
  3. Commit and Abort of top-level TAs have the usual semantics
  4. If subtransaction S aborts, then all operations of S (including subtransactions of S) are undone. <u>Note</u>: the parent TA does not need to fail, it is only notified!

     ➔ subtransactions are atomic
  5. While subtransaction S is executing, it is isolated from other transactions and subtransactions
  6. When subtransaction S commits, the effects of S become visible to other subtransaction of its parent TA
     - ➔ subtransactions are not durable
- Natural solution for transaction composition, but rarely supported in products!

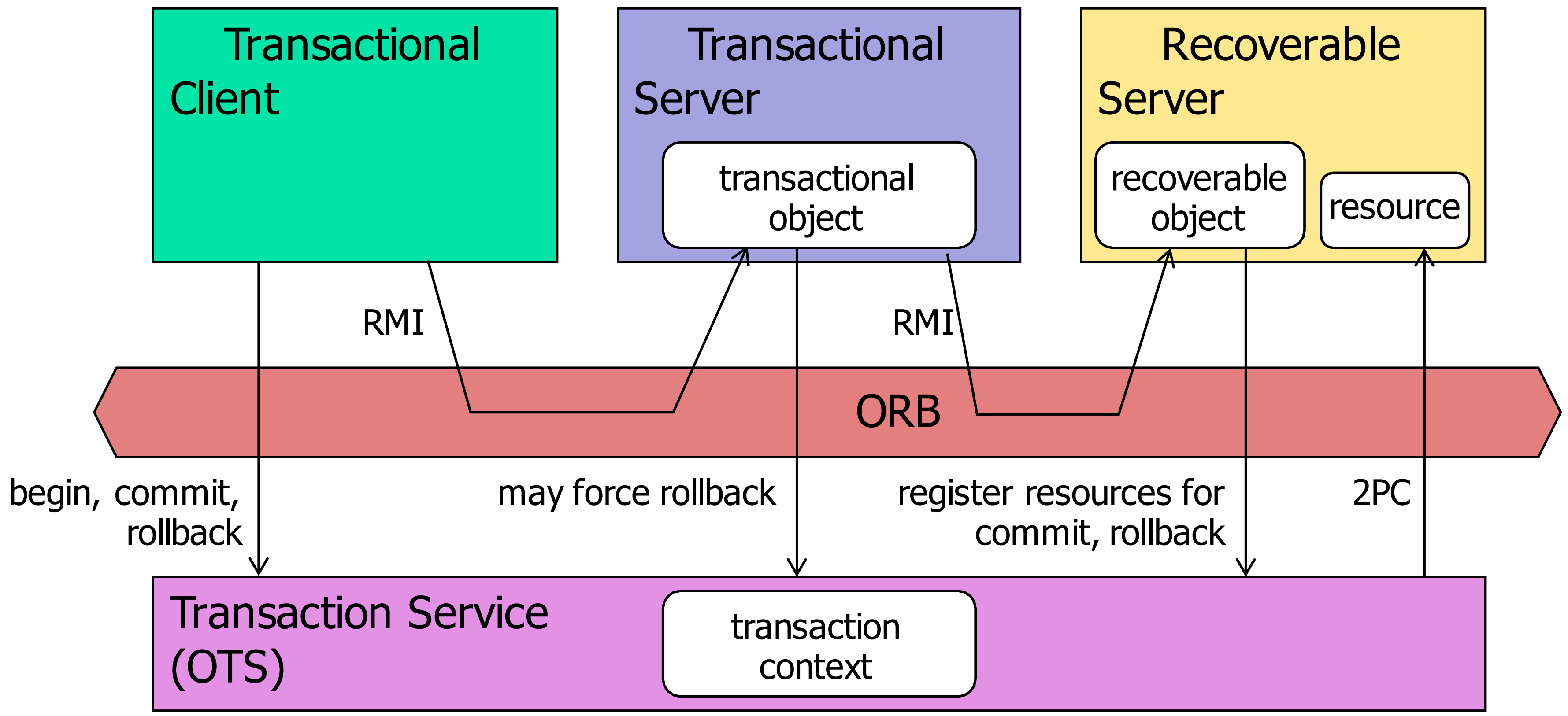
Transaction Processing Standards

- Goal: Interoperability and Integration in Distributed Transaction Processing
  - multiple transactional middleware products can exchange data and control information while executing within the same transaction
  - components running in different middleware platforms can perform different functions that work in combination
- Prominent standards
  - X/Open DTP – covered extensively in the previous chapter
  - CORBA Object Transaction service (OTS)
    - incorporates X/Open XA protocol
  - Java Transaction API (JTA) and Java Transaction Service (JTS)
    - incorporates both XA and OTS

CORBA – Object Transaction Service

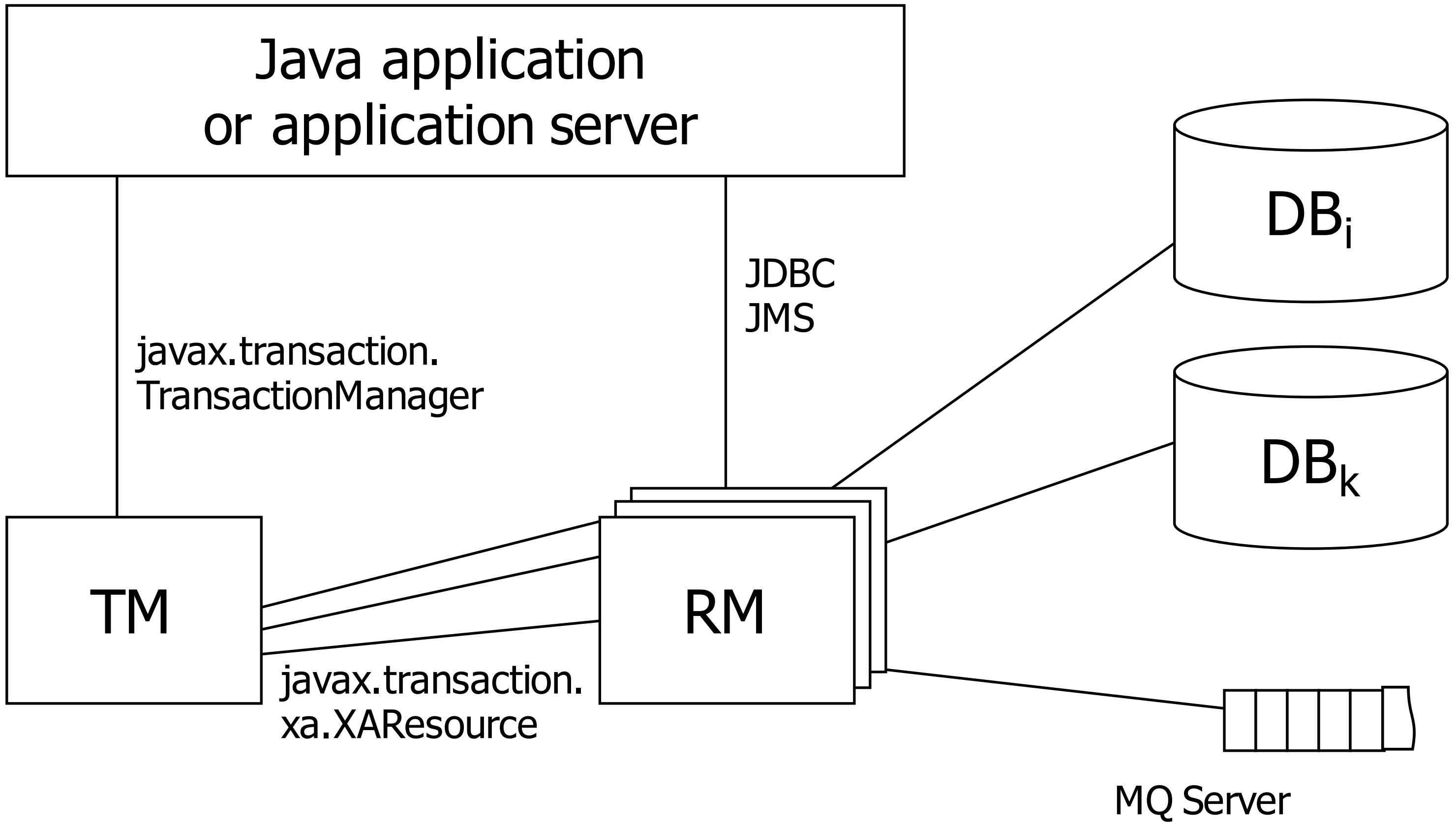
- Based on X/OPEN DTP model and capabilities
  - (flat) ACID transactions
    - optional: nested transactions
  - TAs may span across ORBs
  - X/OPEN DTP
    - interoperability with "procedural" TA-Managers
- Roles and interfaces
  - transactional client
    - demarcation (begin, commit, rollback)
    - uses OTS Interface **Current**
  - transactional server
    - participates in TA, does not manage any recoverable resources
    - "implements" OTS Interface **TransactionalObject**
      - only serves as a "flag" to have the ORB propagate the transaction context
    - optionally uses OTS Interface **Current**
  - recoverable server
    - participates in TA, manages recoverable resources
    - implements OTS Interface **TransactionalObject** and **Resource**, uses **Current** and **Coordinator**
      - participates in 2PC

OTS – Elements and Interaction

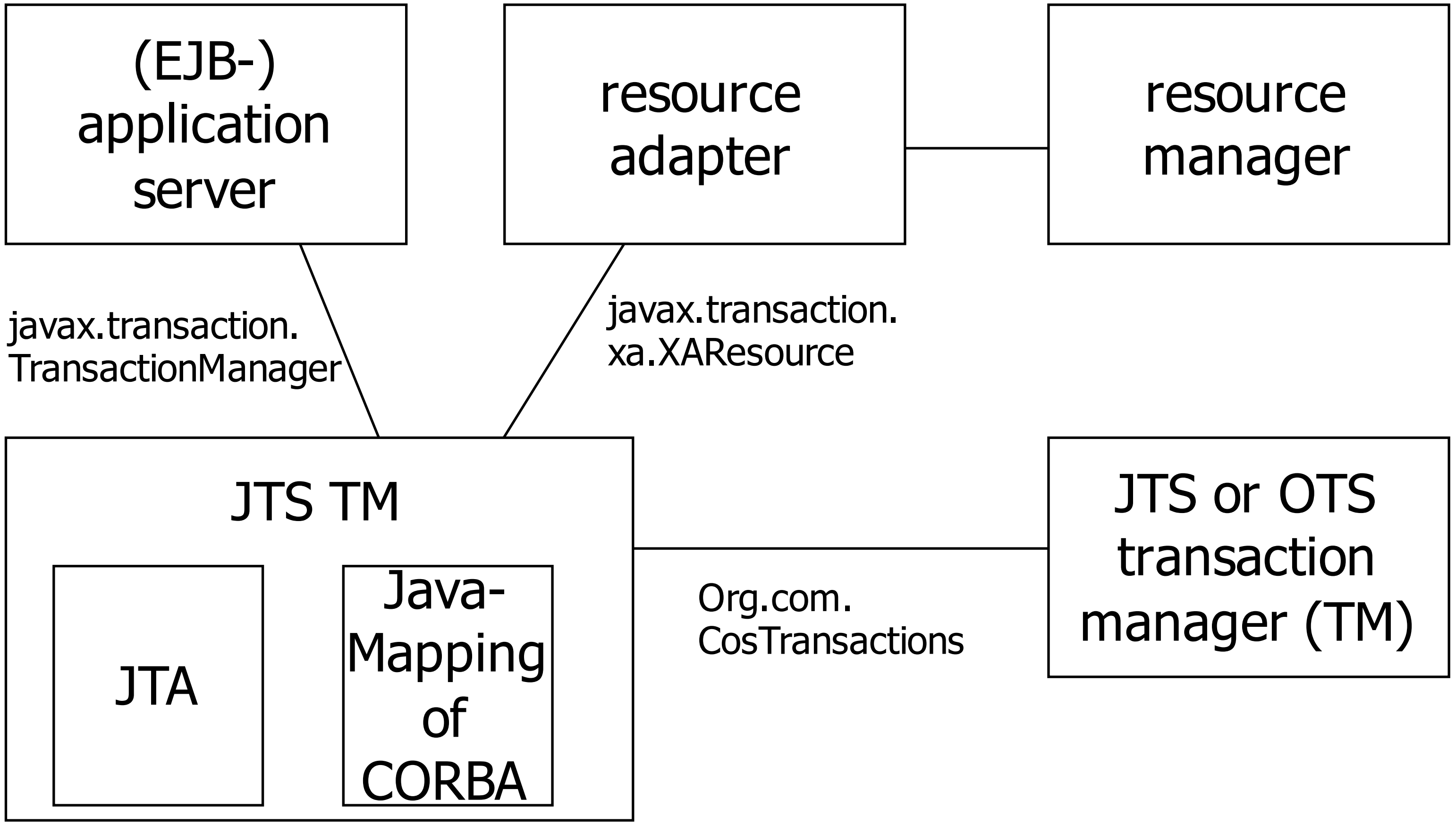


Transactions in Java EE

- Application component may use Java Transaction APIs (JTA)
- UserTransaction object provided via JNDI (or EJB-context)



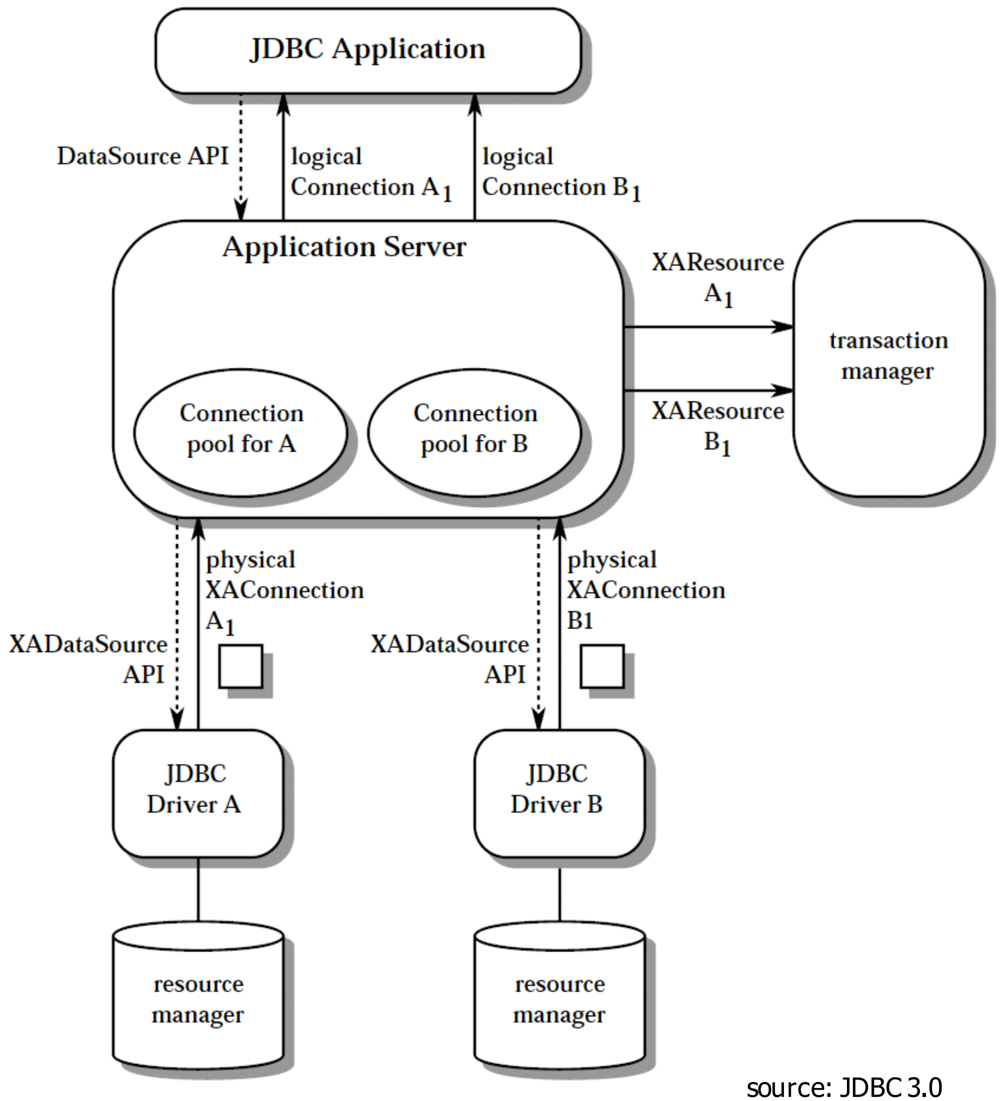
JTS Architecture


(EJB-) application server
resource adapter
resource manager
javax.transaction. TransactionManager
javax.transaction. xa.XAResource
JTS TM
JTA
Java-Mapping of CORBA
Org.com. CosTransactions
JTS or OTS transaction manager (TM)

JDBC - Distributed Transaction Support

- Requires interaction with a transaction manager
  - X/Open DTP, Java Transaction Service (JTS)
- Demarcation of transaction boundaries
  - Java Transaction API (JTA)
    - UserTransaction Object
  - NOT using methods of Connection interface
- JDBC defines additional interfaces to be supported by a driver implementation to interact with transaction manager
  - XADataSource, XAConnection, ...
- DataSource interface helps to make distributed transaction processing transparent to the application

Distributed Transaction Processing with JDBC



source: JDBC 3.0

Shared State

- Components of a TP system may need to share state information
  - Transactions – transaction ID/context information
  - Users – authenticated identity, address of user‘s device
  - Activities – identity/content of last message, temporary information shared between client and server (e.g., shopping cart)
  - Components – identity of TA-managers that need to participate in commit proc.
- Common characteristic: state is short-lived and "transient"
  - state can be discarded after a while
  - often information that describes an activity of limited duration
  - shared mostly for convenience/performance
  - can be reconstructed if it is lost (e.g., due to a failure)
- Differs significantly from long-lived, permanent state
  - e.g., databases containing essential business data

Sessions

- Communication session
  - lasting connection between two system components (e.g., processes) to share state
  - avoids resending/reprocessing information in every message
  - examples
    - network address of both components
    - access control information
    - cryptographic keys
    - transaction-ID (during TRPC)
  - may be created/closed
    - explicitly (e.g., using three-way handshake: request-accept-confirm)
    - as a side-effect (e.g., with the first RPC request from a client to a server)
  - shared session state
    - is communicated when session is initiated
    - requires allocation of storage/memory by each partner
      - space may be significant if a large number of clients connect to a single server
- Common use in TP: database session, session involving client/server process

Stateless Servers

- Potential problems with sessions that involve client and server processes
  - session ties the client to a particular server process (restricts load balancing with multiple processes running the same application)
  - session is lost, if server process fails
  - retaining shared state for a large number of clients costs a lot of memory for the server
- Stateless servers can be used to avoid these problems
  - there is no session with shared state between client and server processes
  - server maintains no application state after a request has been process, starts in a “clean“ state when the next request is processed
- Option for communication between (middle-tier) servers and front-end programs
  - does not apply for communication between server and resource managers

Stateful Applications

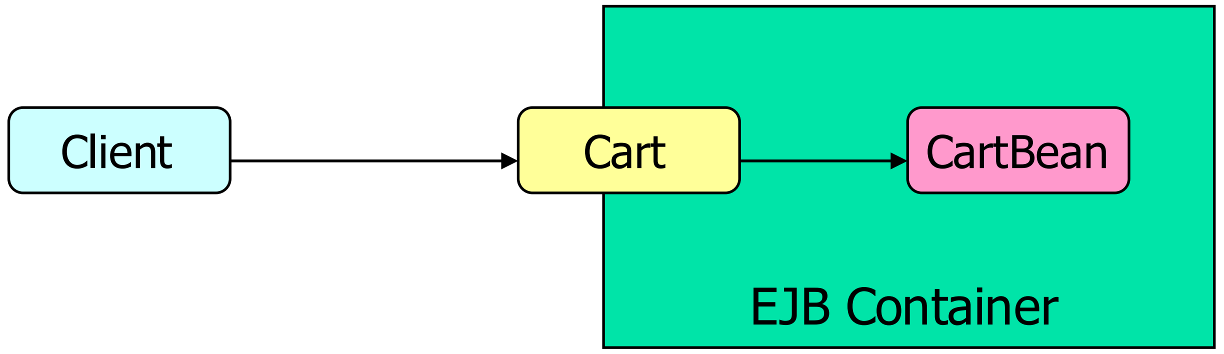
- ... but keeping state in server applications is often required! Scenarios:
  1. user request involves processing multiple transactions (e.g., business process)
  2. server wants to use previous user interaction for customizing the current one
  3. front-end has established a secure connection with user authentication
  4. user accumulates data in multiple steps (e.g., shopping cart) to be used later in the session
- How to manage state?
  - state has to be labelled with identity of client and/or server
  - options for storing the state
    - in persistent shared storage (DBMS) as part of the transaction
    - in persistent shared storage, but not within a transaction
    - in volatile/persistent storage that is local to the server (makes the server stateful)
    - don't store, but return to the caller, who has to resend the state with future requests
  - which options are suitable for the above scenarios?

EJB – Types Of Objects

- Session Object
  - realizes business activity or process
  - often remotely accessible, “course-grained”
  - relatively short-lived (transient)
- Entity Object *(see next chapter)*
  - represent persistent, transactional business object
  - usually locally accessible, “fine-grained”
  - can be long-lived
- Message-driven Object
  - asynchronous, message-oriented invocation (see subsequent chapter)
  - facilitates integration with existing applications

EJB - Concepts

- Enterprise Bean (EB) consists of (ejb-jar file):
  - class implementing business logic (*Bean, e.g., CartBean*)
  - bean business interface, defining methods (*e.g., Cart*)
    - remote and/or local access
  - deployment descriptor/meta-data
- Client interacts with bean using *business interface object*
  - generated at deployment time
  - contains infrastructure code (transaction & security support, ...)
  - client obtains reference to interface object using JNDI (or dependency injection)



Session Beans

- Realization of session-oriented activities and processes
  - isolates client from entity details
  - reduces communication between client and server components
- Session beans are transient
  - bean instance exists (logically) only for duration of a "session”
- *stateless* session bean
  - state available only for single method invocation
- *stateful* session bean
  - (conversational) state is preserved across method invocation
    - session context, stored in the bean’s field variables
  - association of bean instance with client necessary
- *singleton* session bean
  - a single bean instance is shared across applications with concurrent access support
- not persistent, but can manipulate persistent data
  - example: use JDBC, SQLJ to access RDBMS

Example – Client of Stateful Session Bean

- look up Cart interface

```
@Resource SessionContext ctx;      //use dependency injection to obtain JNDI context
Cart cart = (Cart) ctx.lookup(“cart”); //perform lookup, autom. creates EB object
```

- call method to initialize bean

```
cart.startShopping(“John”, “7506”);
```

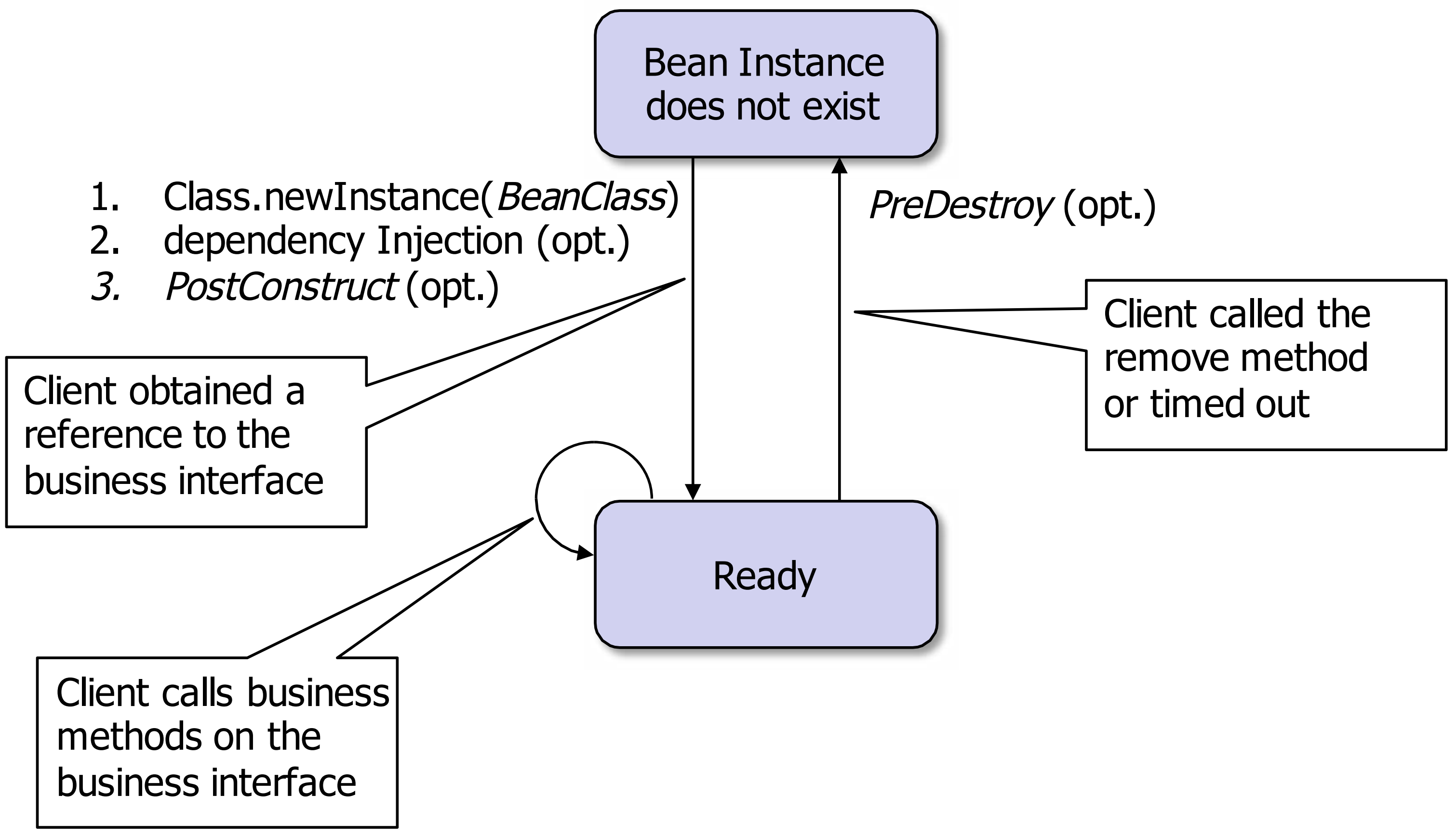
- invoke bean methods

```
cart.addItem(66);
cart.addItem(22);
...
```

- remove session bean

```
cart.close() // the “close” method was annotated/declared as a “RemoveMethod”
```

(Partial) Life Cycle Of Stateful Session Bean



Bean Instance does not exist
1. Class.newInstance(*BeanClass*)
2. dependency Injection (opt.)
3. PostConstruct (opt.)
PreDestroy (opt.)
Client called the remove method or timed out
Client obtained a reference to the business interface
Ready
Client calls business methods on the business interface

Processes and Threads

- Process = virtual processor running a program
  - processor state (processor context) consists of
    - control thread (content of processor registers, processor stack)
    - address space
  - processes provide
    - ability to execute programs in parallel
    - protection entity
    - capability to structure computations into independent execution streams
    - fault containment, if program fails
- Traditional time-sharing systems
  - each display device has its own process
  - each process has exactly one thread executing
  - all programs running on behalf of a display device execute in the same process

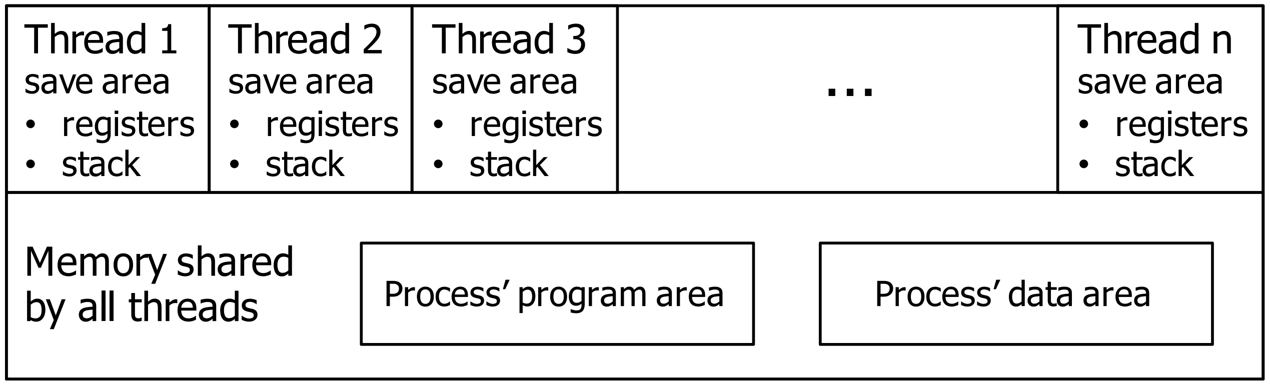
Processes and Threads (2)

- Use time-sharing model for TP-processing?
  - combine front-end, request control, transaction server into one program
  - each terminal (end-user) is connected to its own process running the program
  - main disadvantage: too many processes!
    - operating systems functionality doesn’t scale well enough
    - lots of (expensive) context switching
    - difficult to control the load on such a system
    - data sharing between processes is expensive
- Use a single process (with one thread) for the TP system?
  - would result in other, serious problems
    - no concurrent/parallel processing
      - a page fault will stop the TP processing environment
    - no exploitation of multi-processor systems

Transactional middleware included support for multithreaded processes very early!

Multithreaded Processes

- Multithreading
  - many control threads in a single process/address space
  - each thread is an independent path of execution
  - all threads execute the same program and use the same process memory
  - each of them has a save area for register values and private data
- Advantages
  - save memory (shared process memory)
  - context switching is cheap
  - helps reduce the number of processes



Multithreaded Processes (2)

- Implementing threads
  - by middleware (i.e., OS doesn’t know about the threads)
    - scheduling of threads and processes may interfere
    - middleware has to trap all synchronous I/O operations to avoid OS putting the process to sleep
  - by the operating system (has become ubiquitous in TP products)
    - avoids unnecessary context switching
    - can exploit thread parallelism in multiprocessor/multicore systems
    - performance overhead (context switching involves system calls)
- Disadvantages of multi-threading for TP
  - little/no memory protection between threads (no fault containment, potential security leaks)
    - partially addressed in OS threads by protected memory areas for special subsystems
    - also reduced by use of strongly-typed programming languages
  - imposes requirements of multithreading support for resource managers’ client support

Process Structure of Request Controller and Transaction Server

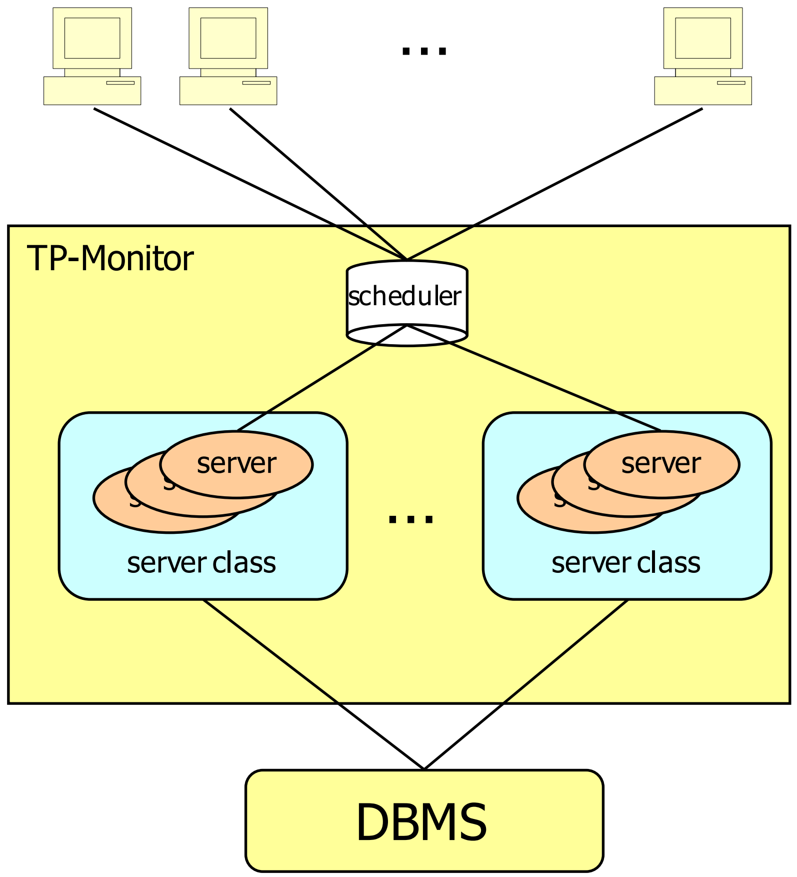
- Request controller typically runs in a multithreaded process
  - dedicated to request controller functions
  - combined with front-end program functions (e.g., in a web server)
  - combined with transaction server functions (using local procedure call/method invocation)
- Transaction server uses multithreading to scale to high request rates
  - needs to have enough threads to handle the maximum required load
  - each active thread/transaction in a transaction server needs a database session
    - implications on performance, security (to be discussed later)
- Application server provides for multithreading support
  - thread management, dynamic or static allocation of threads to devices, sessions
  - system management functions for load balancing, performance tuning and monitoring, etc.
- TP program developer does not need to implement multithreading support, but may have to adhere to certain restrictions/guidelines

Server Classes

- A server class is a set of single-threaded processes, all running the same program
  - "emulate" a pool of threads
  - good alternative, if multithreaded operating system processes are not available
    - normal blocking behavior is fine, not need to trap synchronous I/O calls
    - no potential scheduling conflicts, no possible memory corruption problems
    - failure of individual process doesn't bring down the complete server class
    - each process can use single-threaded services
  - disadvantages
    - one process per thread (expensive) → need for pooling of servers
    - additional scheduling for load balancing required (across servers of the same class)
- Popular approach in "classic" TP-monitors

Application Processing – Server Classes

- Managing large workloads
  - one process per client is not feasible
  - TP monitor manages server pools
    - groups of processes or threads, pre-started, waiting for work
  - client requests are dynamically directed to servers
  - extends to pooling of resource connections
- Load balancing
  - distribute work evenly among members of pool
  - TP monitor can dynamically extend/shrink size of server pools based on actual workload
  - management of priorities for incoming requests



Scalability

- Building TP system to handle high load involves
  - scaling up a server system
    - choose appropriate hardware configuration
    - caching
    - resource pooling
  - scaling out by adding more machines
    - distribute the workload over multiple machines
    - partitioning
    - replication

Caching

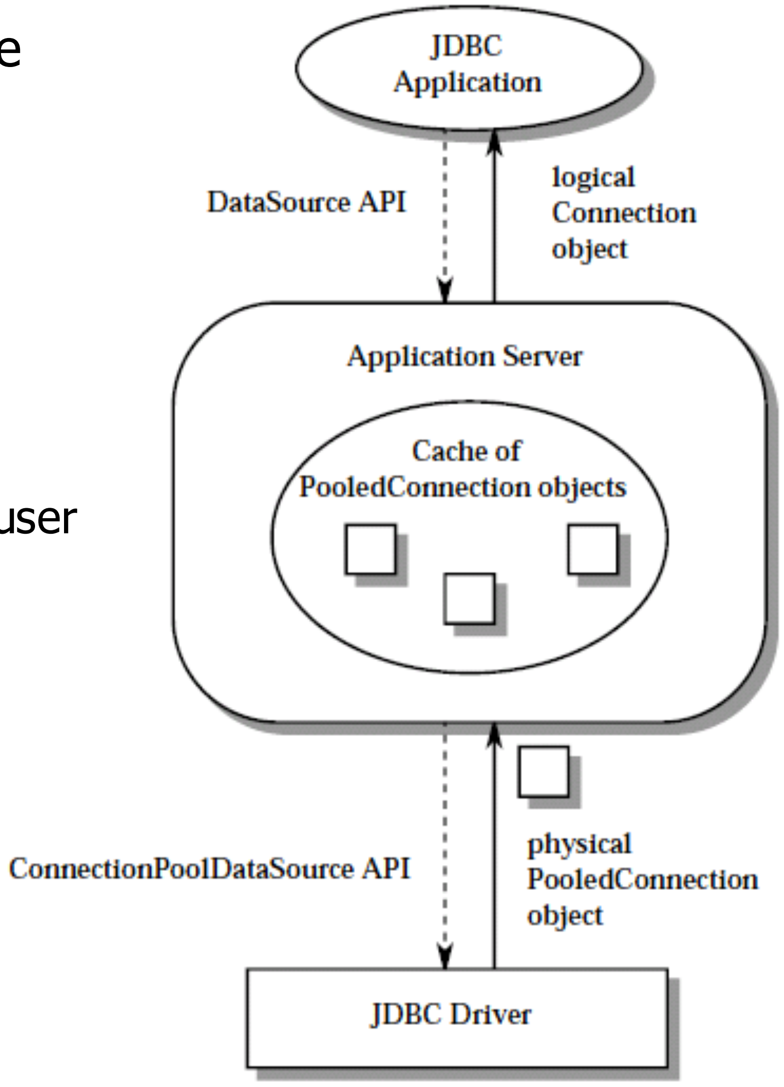
- Cache
  - area of memory that contains data whose “permanent home” is somewhere else
  - ideally, cached data is frequently accessed by applications using the cache
    - avoids (expensive) frequent access of data at the home location
- TP use of caching
  - web browser, web server, proxy server, 3rd party websites
  - application server
  - DB system (DB-cache, main-memory databases)
- Potential issues
  - fast lookup, effective replacement strategies
  - cached data may be out of date
    - happens when data is updated in its permanent home
    - cache is most useful for infrequently updated data
    - requires cache invalidation strategies
  - cache coherence – all caches should hold the same (recent) data values
  - updatable caches
    - propagation of updates to permanent home (write-through, write-back)

Resource Pooling - Connection Pooling

- "Caching" of resources that are costly to create but relatively inexpensive to access
  - database sessions/connections
  - process threads
  - processes in server classes

Connection Pooling

- Server-side application components
  - DB access often in the context of few (shared) user ids
  - connection is often held only for short duration (i.e., short processing step)
- Reuse of physical DB connection desirable
  - open -> "get connection from pool"
  - close -> "return connection to pool"
- Connection pooling can be "hidden" by DataSource, Connection interfaces
  - transparent to the application

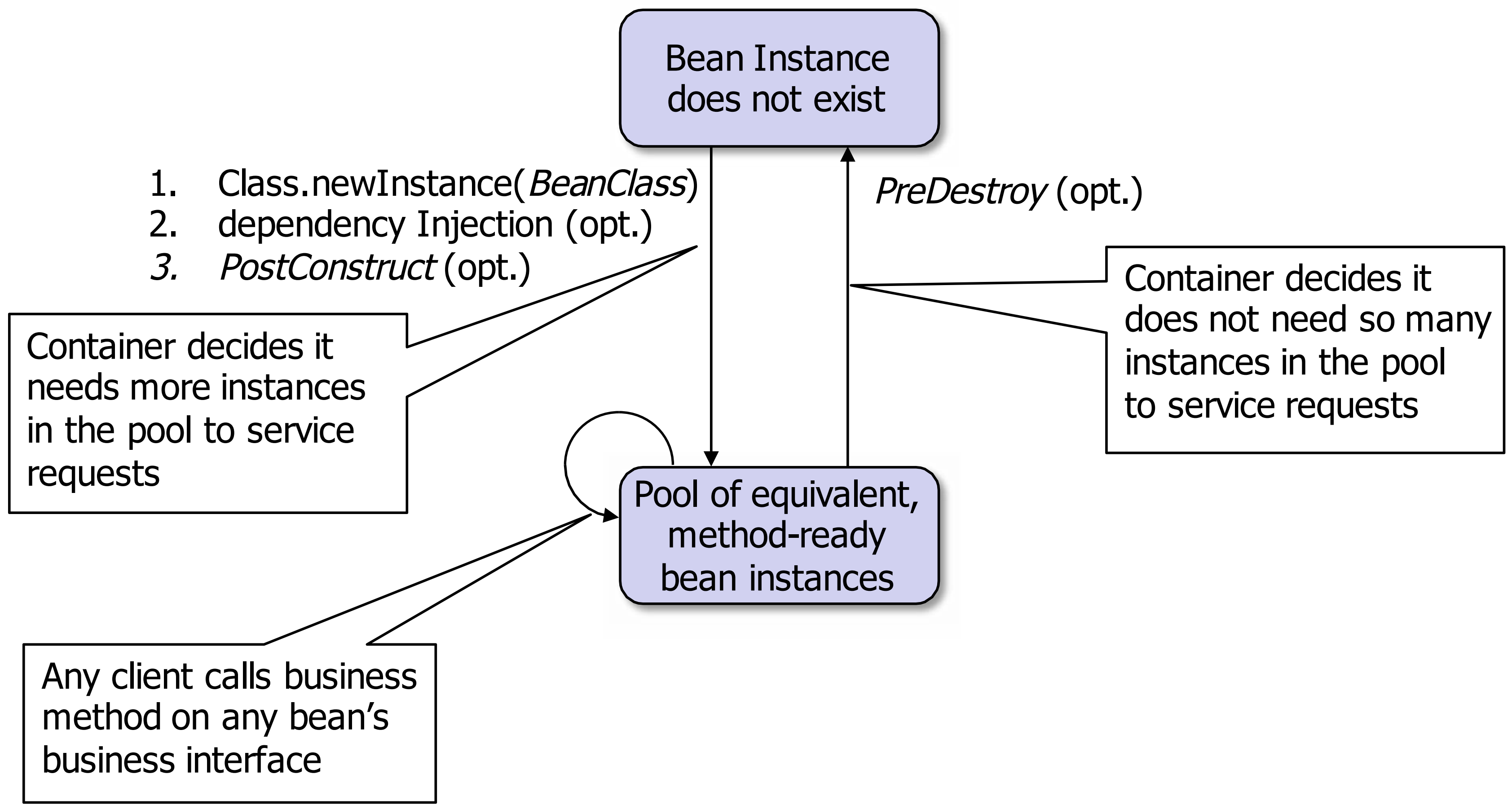

JDBC Application
DataSource API
logical Connection object
Application Server
Cache of PooledConnection objects
ConnectionPoolDataSource API
physical PooledConnection object
JDBC Driver

source: JDBC 3.0

EJB Resource Management

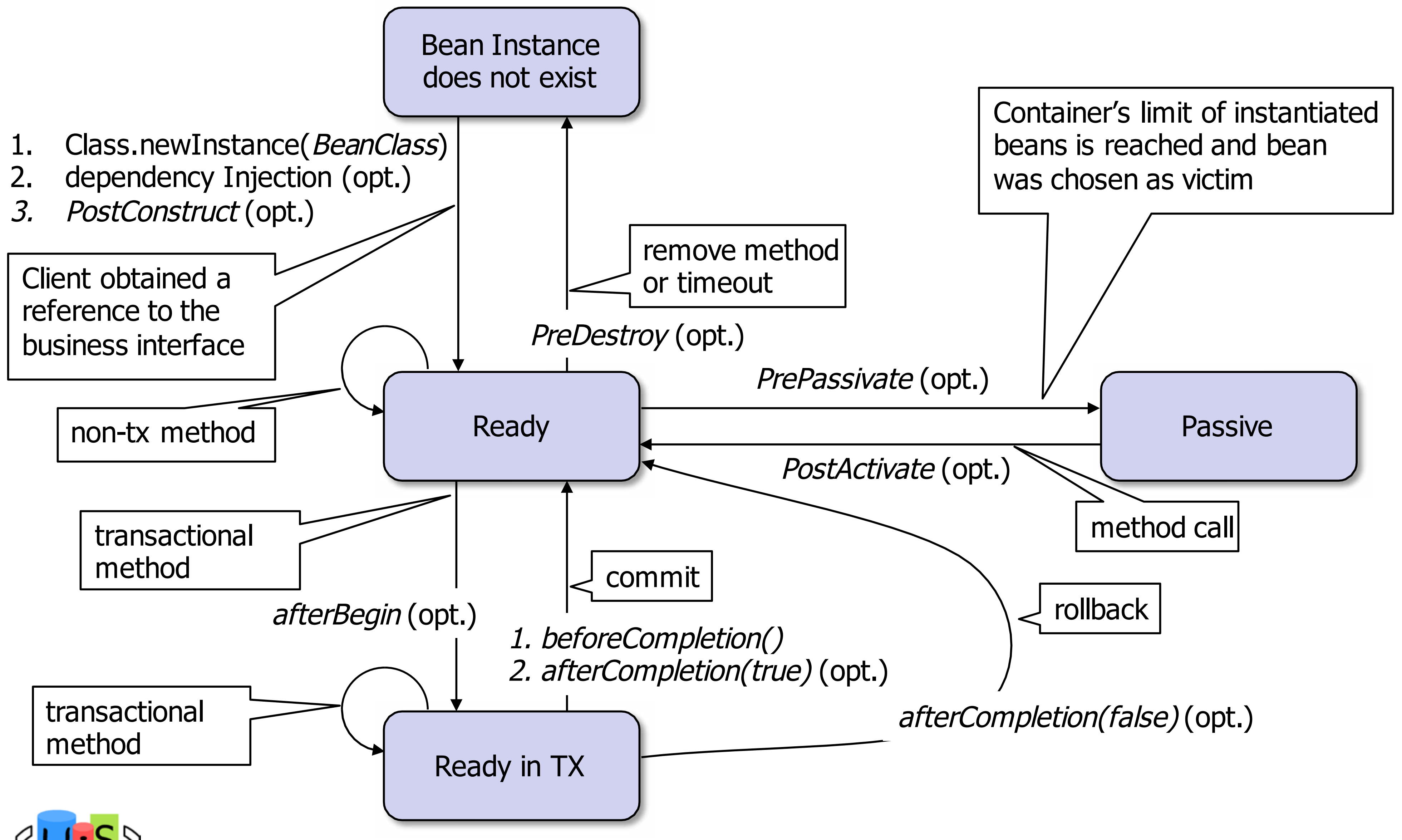
- Traditional task of a (component) TP monitor
  - pooling of resources, load management and balancing
- EJB specification
  - *Instance Pooling* and *Instance Swapping*
    - EJB server manages (small) number of Enterprise Beans
      - reuse, dynamic selection for processing incoming requests
    - made possible by 'indirect' bean access via EJB business interface
    - usually only applicable for **stateless session beans**
  - *Passivation* and *Activation*
    - bean state can be stored separately from bean (*passivation*)
      - allows freeing up resources (storage), if bean is not used for a while (e.g., end user think time)
    - if needed, bean can be reactivated (*activation*)
    - uses Java Serialization
    - can be used for **stateful session beans**

Life Cycle Of Stateless Session Bean



Bean Instance does not exist
1. Class.newInstance(*BeanClass*)
2. dependency Injection (opt.)
3. PostConstruct (opt.)
PreDestroy (opt.)
Container decides it needs more instances in the pool to service requests
Container decides it does not need so many instances in the pool to service requests
Pool of equivalent, method-ready bean instances
Any client calls business method on any bean's business interface

Complete Life Cycle Of Stateful Session Bean


Bean Instance does not exist
1. Class.newInstance(*BeanClass*)
2. dependency Injection (opt.)
3. *PostConstruct* (opt.)
Client obtained a reference to the business interface
non-tx method
remove method or timeout
PreDestroy (opt.)
Container's limit of instantiated beans is reached and bean was chosen as victim
PrePassivate (opt.)
Ready
Passive
PostActivate (opt.)
method call
transactional method
commit
rollback
afterBegin (opt.)
1. beforeCompletion()
2. afterCompletion(true) (opt.)
transactional method
afterCompletion(false) (opt.)
Ready in TX

Partitioning

- Partition the application and its data
  - separate according to type of work (e.g., credit cards, loan processing, accounts)
  - assign each type to a different machine
  - effective, but limited
- Range partitioning
  - different copies of the server handle different ranges of an input parameter
    - involves parameter-based routing of requests
    - example: same debit-credit application running on multiple servers for different account ranges
  - directly supported by many high-function DBMS
- Hash partitioning
  - use a hash function to map an input parameter to a server
  - likely to achieve better load balancing
- Partitioning sessions
  - insert a routing layer between clients and servers, partitioning the set of clients
  - reduces the overall number of sessions, if each client needs to access each server

Replication

- Distribute workload by replicating server processes
  - assign replicas to different systems (can all process the same requests)
  - works well with stateless servers
  - server system may implement load balancing (spraying requests) across the servers
  - load balancing can be further improved by using shared request queues (*see chapter on message-oriented middleware*)
- Replication interacts with caching
  - sequence of "similar" requests
  - cache affinity
- *More discussion on data replication in a later chapter*

Summary

- Architecture of transaction processing applications
  - front-end programs, request controller, transaction server
- Application servers are crucial for supporting development and execution of TP application programs to build scalable TP systems
- Different types of application server middleware
  - TP monitors, object brokers, object transaction monitors, component transaction monitors
- Transactional capabilities in application servers
  - address the transaction composability problem
    - transaction demarcation/bracketing approaches (explicit vs. implicit demarcation)
    - nested transactions
  - transaction processing standards and interoperability
- Support for shared state
  - sessions, stateless vs. stateful applications/servers
- Mapping application components to processes and threads
  - multithreading, server classes, process structure of request controller, TA server
- Scalability (caching, pooling, partitioning and replication)