Beyond EJB

Client Transactions with EJB

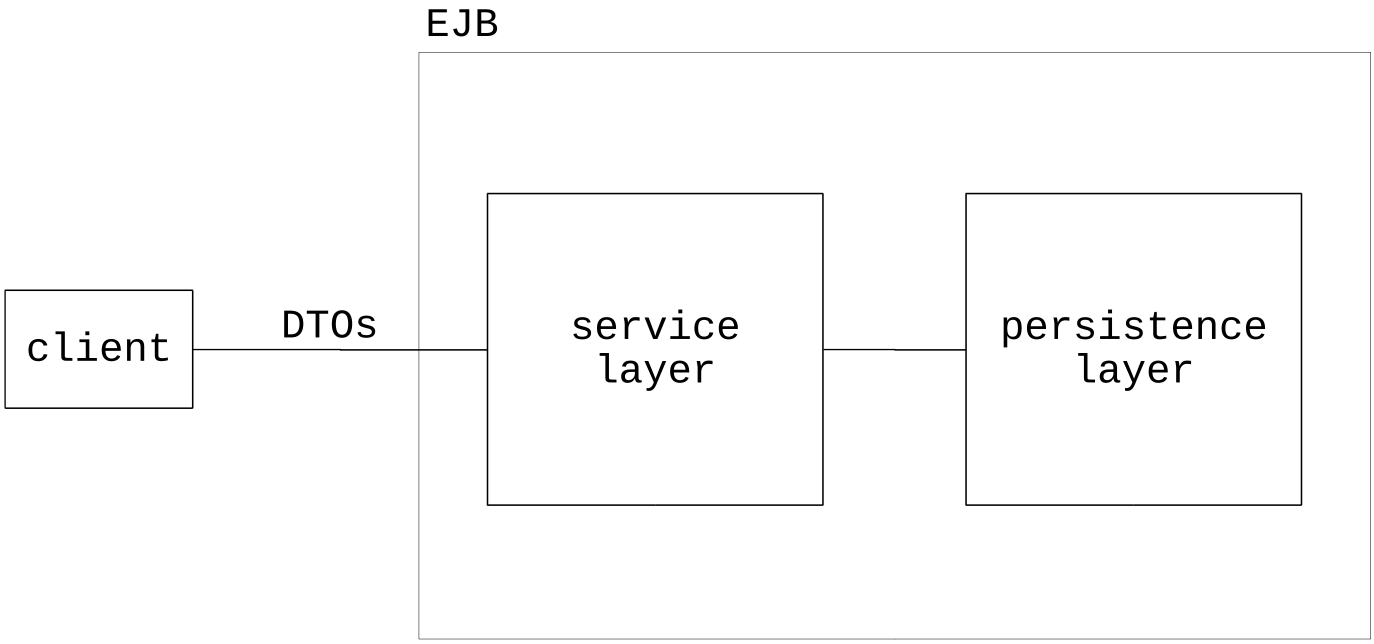
Angelika Langer
Trainer/Consultant
http://www.AngelikaLanger.com

Klaus Kreft
Senior Consultant
klaus.kreft@siemens.com

objective

- EJB:
  - Java's model for component-based enterprise applications
- main benefits:
  - support for transactions
  - support for persistence
- objective of this tutorial:
  - build transactions and persistence with and on top of EJB
  - show benefits and limitations of EJB framework
  - explain common techniques

J2EE blueprint architecture



J2EE blueprint architecture

- client
  - Java application with Swing
  - browser with servlets/JSP
- DTOs = Data Transfer Objects
  - also known as Value Objects
  - generic hashtable of key-value pairs
  - domain-specific business object representations
- service layer
  - session beans
  - message driven beans
- persistence layer
  - entity beans

persistence layer

- alternatives for persistence layer
  - entity beans
  - JDBC: service layer directly uses JDBC
  - object-relational mapping tools, e.g. JDO (Java Data Objects)
- use entity beans in all examples
  - alternatives do not affect principles of solution

transaction properties – ACID

- Atomicity
  - operations in a transaction (TX) appear as one unit of work
  - all-or-nothing; commit or rollback
- Consistency
  - always maintain data in a consistent state
  - each TX transforms data from one consistent state into another
- Isolation
  - concurrent TXs are isolated
  - operations must be synchronized via locks
- Durability
  - data updates are permanent
  - Txs manipulate a persistent data store

transactional models

- EJB supports TXs in various ways (CMT / BMT)
  - CMT: TXs strictly tied to beans methods
  - BMT: more latitude, still mostly fine grained TXs
- of actual interest are TXs tied to end-user interactions

terminology

- system TX
  - EJB transaction or JTA/JTS transaction
  - basically everything that is performed by EJB container or EJB TX manager
  - includes underlying database TXs
- logic TX
  - TX on application level
  - "unit of work" in the sense of ACID

objective

- discuss several approaches for implementing logic TX
- plain system TX
  - simply use system TX
    - atomic services
    - client-initiated TX
- user-implemented TX
  - complex use of system TX functionality
    - optimistic locking
    - pessimistic locking

agenda

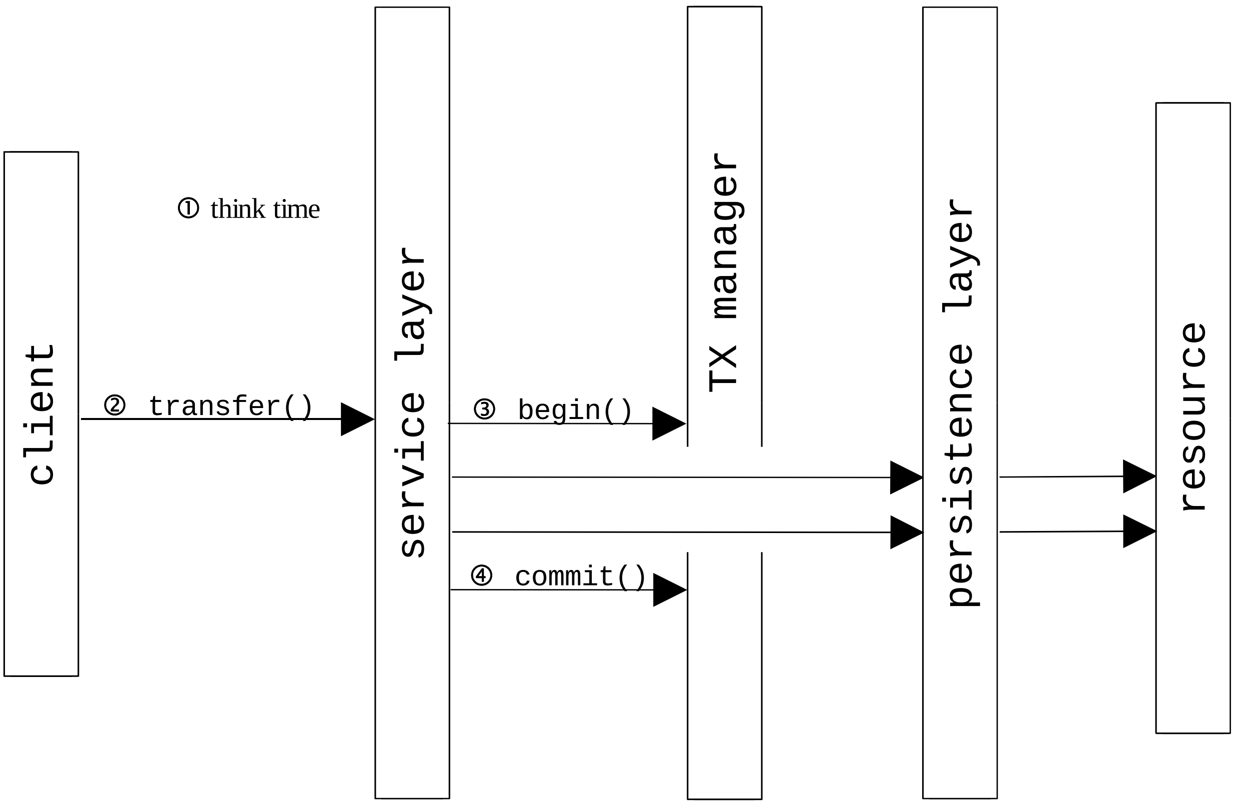
- transactions
  - atomic services
  - client-initiated TX
  - optimistic locks
  - pessimistic locks
- case study

atomic services

- all services in an application are atomic
  - no logic TX spans several operations
  - logic TX = system TX
  - possible if no user dialog necessary to perform service
  - common in B2C domains

  - example:
    - money transfer from one account to another
    - user provides all necessary data on invocation
    - service performs operation in one TX
  - counter example:
    - travel arrangements
    - book my flight A only if there are seats available for members of my party on flight B, C, and D

atomic services



user dialog

- "no user dialog" not quite correct
- dialog can be performed on client side
- consequences:
  - lack of TXs
  - no clean separation of concerns:
    - service logic partly moved to client

atomic services – evaluation

- common in practice
- drawbacks:
  - problematic if concurrent access to resources required
  - example:
    - corporate bank account
    - accessed simultaneously by several departments
  - does not evolve with changing business requirements
  - example:
    - change money transfer: check balance prior to transfer
    - requires user dialog

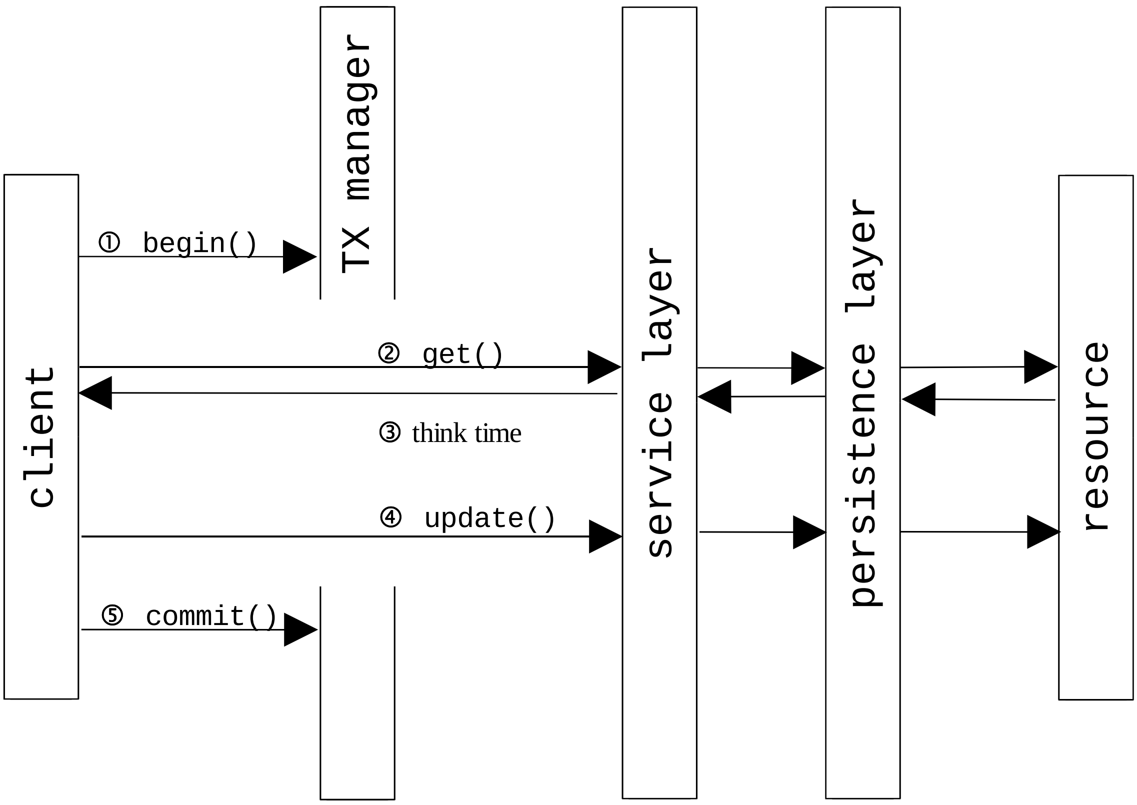
agenda

- transactions
  - atomic services
  - client-initiated TX
  - optimistic locks
  - pessimistic locks
- case study

client-initiated transactions

- idea:
  - logic TX = system TX
  - client starts and ends the transaction
  - user dialog, business logic and persistence run under protection of client TX

client initiated TX



client-initiated TX – implementation

- scenario
  - client uses JTS / JTA
    - to begin and commit / rollback the TX
  - client calls service layer methods
    - included in client TX scope
  - data exchange between client and service layer via DTO
    - DTO = data transfer object
  - service layer consists of session beans
    - must be CMT (= container managed transaction) with TX attribute `Required` (or `Mandatory`)
    - BMT starts its own TX and suspends client TX
  - service can cause failure
    - throws system exception (`RuntimeException`)
    - requests rollback (via `setRollbackOnly()`)

client-initiated TX – client code

```
Context jndiContext = getInitialContext();
Object ref = jndiContext.lookup("ServiceBeanHomeRemote");
ServiceBeanHomeRemote home = (ServiceBeanHomeRemote)
  PortableRemoteObject.narrow(ref,ServiceBeanHomeRemote.class);
ServiceBeanRemote sb = home.create();
int id = 4711; DTO dto = null;

UserTransaction utx = (javax.transaction.UserTransaction)
  jndiContext.lookup("java:comp/UserTransaction");
utx.begin();
try  {
  dto = sb.getDTO(id);
  if (dto == null) {  dto = new DTO(id,null,null);  }
  ... figure out new values ...
  dto.setAttribute1(... new value ...);
  dto.setAttribute2(... new value ...);
  sb.setDTO(dto);
  utx.commit();
}
catch( javax.transaction.RollbackException e )
{ /* automatic rollback was performed instead of commit */ }
catch( Exception e )
{ utx.rollback();  }
```


client-initiated TX – DTO

```
public class DTO implements java.io.Serializable {
   private int id;
   private String attribute1;
   private String attribute2;

   public DTO(int pk, String s1, String s2)
   { id = pk; attribute1 = s1; attribute2 = s2; }
   public int getId()
   { return id; }
   public String getAttribute1()
   { return attribute1; }
   public void setAttribute1(String s)
   { attribute1 = s; }
   public String getAttribute2()
   { return attribute2; }
   public void setAttribute2(String s)
   { attribute2 = s; }
}
```

client-initiated TX - service layer session bean

```
public interface ServiceBeanRemote extends javax.ejb.EJBObject {

   public DTO getDTO(int pk) throws RemoteException;
   public void setDTO(DTO data) throws RemoteException;
}
```


client-initiated TX - service layer session bean

```
public class ServiceBean implements javax.ejb.SessionBean
{
  public DTO getDTO(int pk) throws RemoteException
  {
     DataBeanHomeLocal home = null;
     try {
       Context jndiContext = new InitialContext();
       home = (DataBeanHomeLocal)
         jndiContext.lookup("java:comp/env/ejb/DataBeanHomeLocal");
     }
     catch (NamingException ne) { throw new EJBException(ne); }
     DataBeanLocal dataBean = null;
     try { dataBean = home.findByPrimaryKey(new Integer(pk)); }
     catch (FinderException fe) { return null; }
     DTO result = new DTO(pk,dataBean.getAttribute1(),
                             dataBean.getAttribute2());
     return result;
  }
}
```

client initiated TX – evaluation

- leads to long lasting transaction
  - blocks many resources for a long time
    - e.g. blocks DB connections
  - includes user think time

    = time between "get resource" and "update resource"
- poor decoupling
  - client is involved in business Txs
- rarely used in practice
  - does not scale

agenda

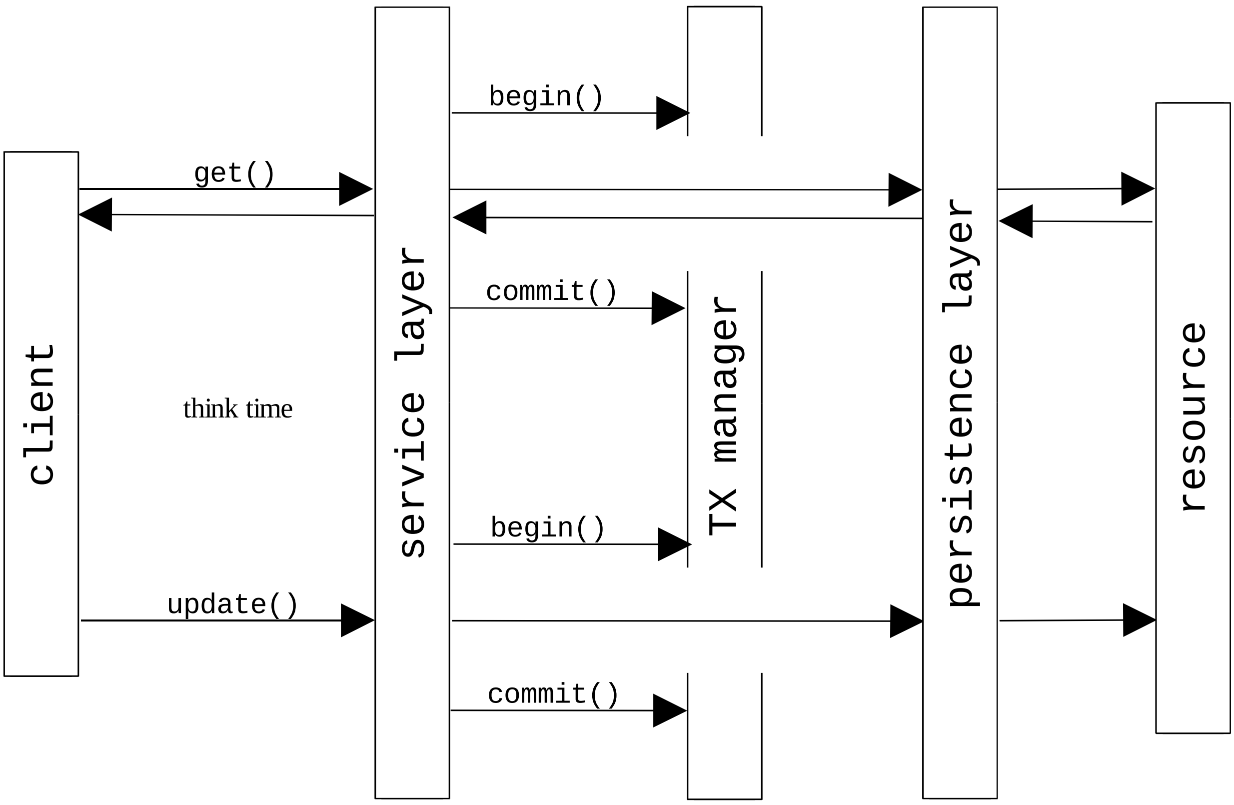
- transactions
  - atomic services
  - client-initiated TX
  - optimistic locks
  - pessimistic locks
- case study

optimistic lock – motivation

- goal: exclude user think time from TX

- logical TX as before
  - comprises "get resource", think time, and "update resource"
  - update fails in case of conflict
- logic TX = user-implemented TX
  - split logic TX into shorter system TXs
    - for "get resource" and "update resource" respectively
  - no explicit demarcation for logic TX necessary
    - if update fails no rollback is necessary because nothing has been changed yet

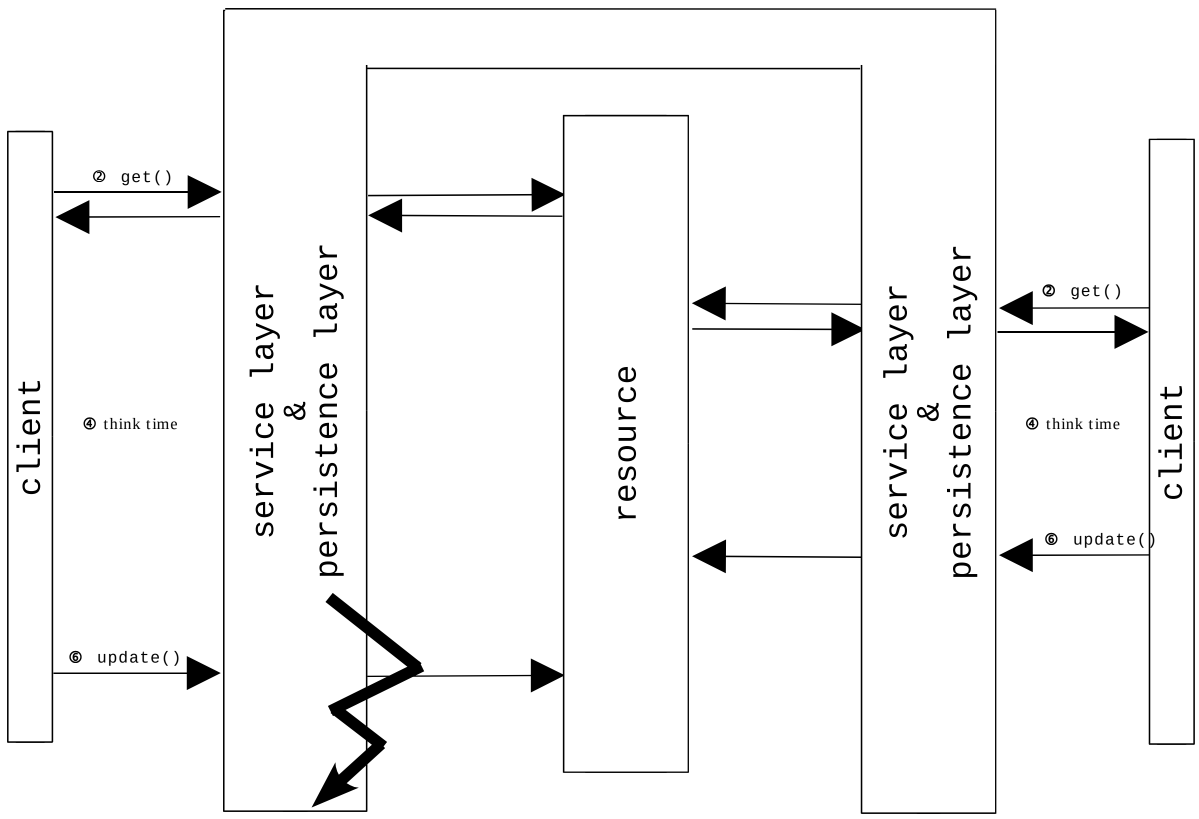
optimistic lock



stale data problem

- problem:
  - if several clients compete for the same resource updates might be based on stale data
- solution:
  - update fails in case of conflict

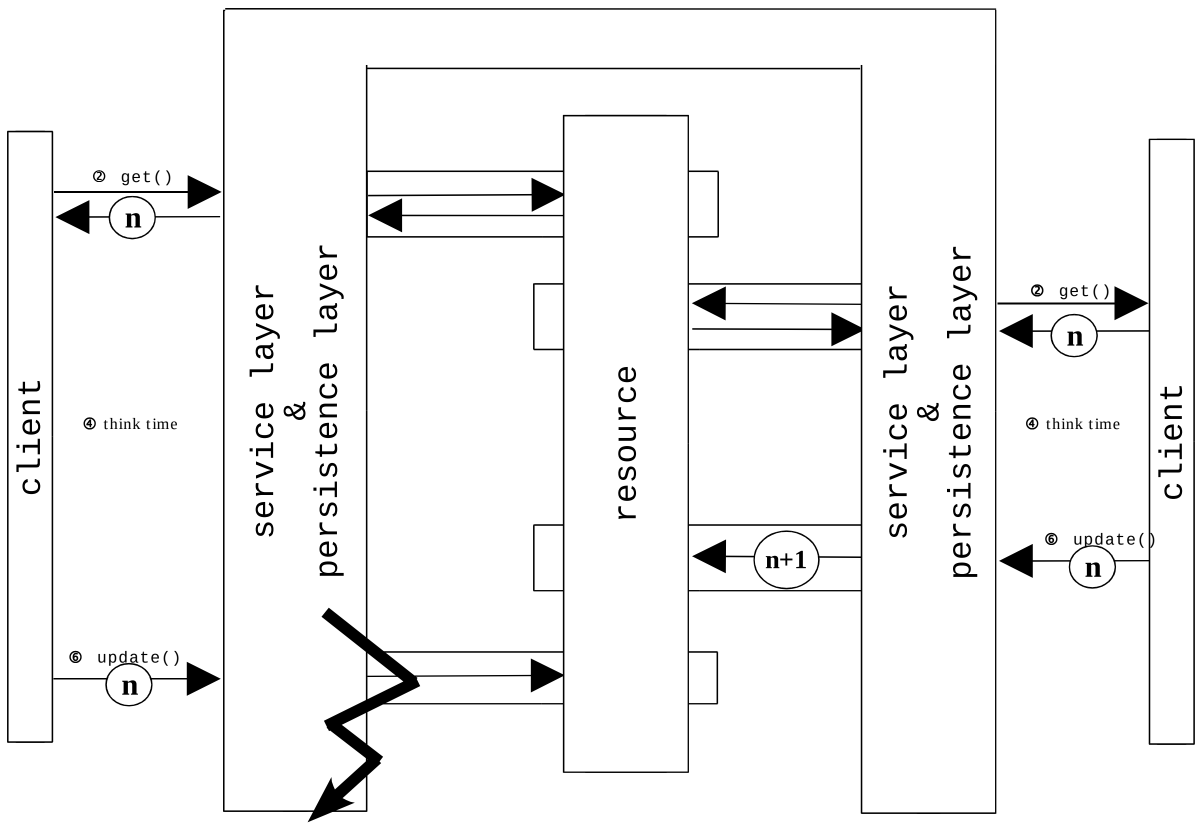
stale update



stale update – solution

- use version numbers or time stamps to perform staleness checks
  - add version number to resource
    - sometimes already provided by persistence layer if object-relational mapping tools are used
  - carry around version in all data transfers
  - on update check for matching versions
    - reject update on mismatch
    - otherwise perform update and increment version

optimistic locking using versions



optimistic lock – implementation

- persistence layer: entity beans
  - add version number to all entity beans
  - add method for staleness check
- service layer: session beans
  - add version number to all DTOs
  - hide version from client
  - invoke staleness check
  - inform client of failure

CMP entity bean

```
public abstract class DataBean implements javax.ejb.EntityBean
{
  public Integer ejbCreate(Integer id) throws CreateException
  {  this.setId(id);
     this.setVersion(new Integer(0));
     return null;
  }
  public abstract void setId(Integer id);
  public abstract Integer getId();

  public abstract void setVersion(Integer vers);
  public abstract Integer getVersion();

  public abstract void setAttribute1(String name);
  public abstract String getAttribute1();

  public abstract void setAttribute2(String name);
  public abstract String getAttribute2();

  ... continued on next slide ...

}
```

add version

CMP entity bean (cont.)

```
public abstract class DataBean implements javax.ejb.EntityBean
{
  ... continued from previous slide ...

  public void checkAndUpdateVersion(Integer transferVersion)
    throws VersionMismatchException
  {
     Integer storedVersion = getVersion();
     if ( transferVersion.intValue() == storedVersion.intValue())
        setVersion(new Integer(++storedVersion));
     else
        throw new VersionMismatchException(storedVersion,transferVersion);
  }
}
```


DTO

```
public interface DTO extends java.io.Serializable {
   public String getAttribute1();
   public void setAttribute1(String s);
   ...
}
```

client view

```
public class StampedDTO implements DTO {
   private int id;
   private int version;
   private String attribute1;
   ...
   public StampedDTO(int pk, int vers, String s1, ...)
   { id = pk; version = vers; attribute1 = s1; ... }

   public StampedDTO(int pk)
   { this(pk,0,null,...); }

   public int getId() { return id; }
   public int getVersion()  { return version; }
   public String getAttribute1() { return attribute1; }
   public void setAttribute1(String s) { attribute1 = s; }
   ...
}
```

service layer view

service session bean

```
public DTO getDTO(int pk) throws RemoteException
{
  DataBeanHomeLocal home = null;
  try {
    Context jndiContext = new InitialContext();
    home = (DataBeanHomeLocal) jndiContext.lookup
    ("java:comp/env/ejb/DataBeanHomeLocal");
  } catch (NamingException ne) { throw new EJBException(ne); }

  DataBeanLocal data = null;
  try {
    data = home.findByPrimaryKey(new Integer(pk));
  } catch (FinderException fe)
  { // DataBean does not exist; create empty DTO
    return new StampedDTO(pk);
  }
  // DataBean found
  return new StampedDTO(pk,data.getVersion().intValue(),
                          data.getAttribute1(),
                          data.getAttribute2());
}
```


service session bean (cont.)

```
public void setDTO(DTO dto)
  throws RemoteException, StaleUpdateException
{
  DataBeanHomeLocal home = ... JNDI lookup ...

  StampedDTO sdto = (StampedDTO) dto;
  Integer pk = new Integer(sdto.getId());
  Integer vs = new Integer(sdto.getVersion());
  boolean mustBeCreated = (sdto.getVersion()==0);

  DataBeanLocal data = findOrCreateDbEntry(home,pk,mustBeCreated);
  try {
    data.checkAndUpdateVersion(vs);
    data.setAttribute1(sdto.getAttribute1());
    data.setAttribute2(sdto.getAttribute2());
  }
  catch(VersionMismatchException e) {
    ejbContext.setRollbackOnly();
    throw new StaleUpdateException();
  }
}
```

TX attributes in CMT

- service session bean implicitly starts system TX
- data entity bean is included in system TX scope
  - both via CMT attribute `Required`
  - alternatives: `RequiresNew` for session bean / `Mandatory` for entity bean

```
<container-transaction>
  <method>
    <ejb-name>DataEJB</ejb-name>
    <method-name>*</method-name>
  </method>
  <method>
     <ejb-name>ServiceEJB</ejb-name>
     <method-name>*</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
```


client code

```
Context jndiContext = getInitialContext();
Object ref = jndiContext.lookup("ServiceBeanHomeRemote");
ServiceBeanHomeRemote home = (ServiceBeanHomeRemote)
  PortableRemoteObject.narrow(ref,ServiceBeanHomeRemote.class);
ServiceBeanRemote sb = home.create();

while (manipulateData(sb))
{ /* retry */ }
```

```
private static boolean manipulateData
(ServiceBeanRemote sb, int id)
{ ... figure out id ...
  DTO dto = sb.getDTO(id);
  ... figure out new values ...
  dto.setAttribute1(... new value ...);
  dto.setAttribute2(... new value ...);
  try {
    sb.setDTO(dto);
    return false;  // update sucessful
  } catch (StaleUpdateException e) {
    return true;   // update denied: retry
  }
}
```

retry

- retry updates until success
  - typically not just a programmatic loop

  - might require user dialog
    - so that user can decide what to do

  - reaction includes:
    - re-navigate to re-obtain data
    - re-perform operations and re-try update

  - can be supported by service layer
    - by providing current data present in data store
    - reduces effort of re-navigation

optimistic locks – evaluation

- very common technique
  - 90% of all applications work like this
- upside
  - easy to implement
  - avoids bottleneck of long client initiated TXs
- downside
  - client must cope with update failure
  - puts burden onto end user

lack of atomicity

- optimistic locks work for simple get/set cases
- repeated get/set does not perform as "unit of work"

- staleness checks ensure isolation, but no atomicity
  - in case of update failure "unit of work" is only partly done
  - client is responsible to ensure atomicity
  - two conceivable solutions:

    [1] retry

    [2] abort

retry

- not a real option
  - retry might never succeed or
    might be undesired in the first place

  - example:
    - booking flight tickets for a party of 2+ people
    - if flight is booked
      - we would retry on the same flight forever
      - and a retry on another flight for half of the party is undesired

abort

- two techniques:
  - write-through
    - perform update immediately on DB
    - undo in case of failure
  - cache
    - postpone operations; store data updates in cache
    - perform updates in case of success

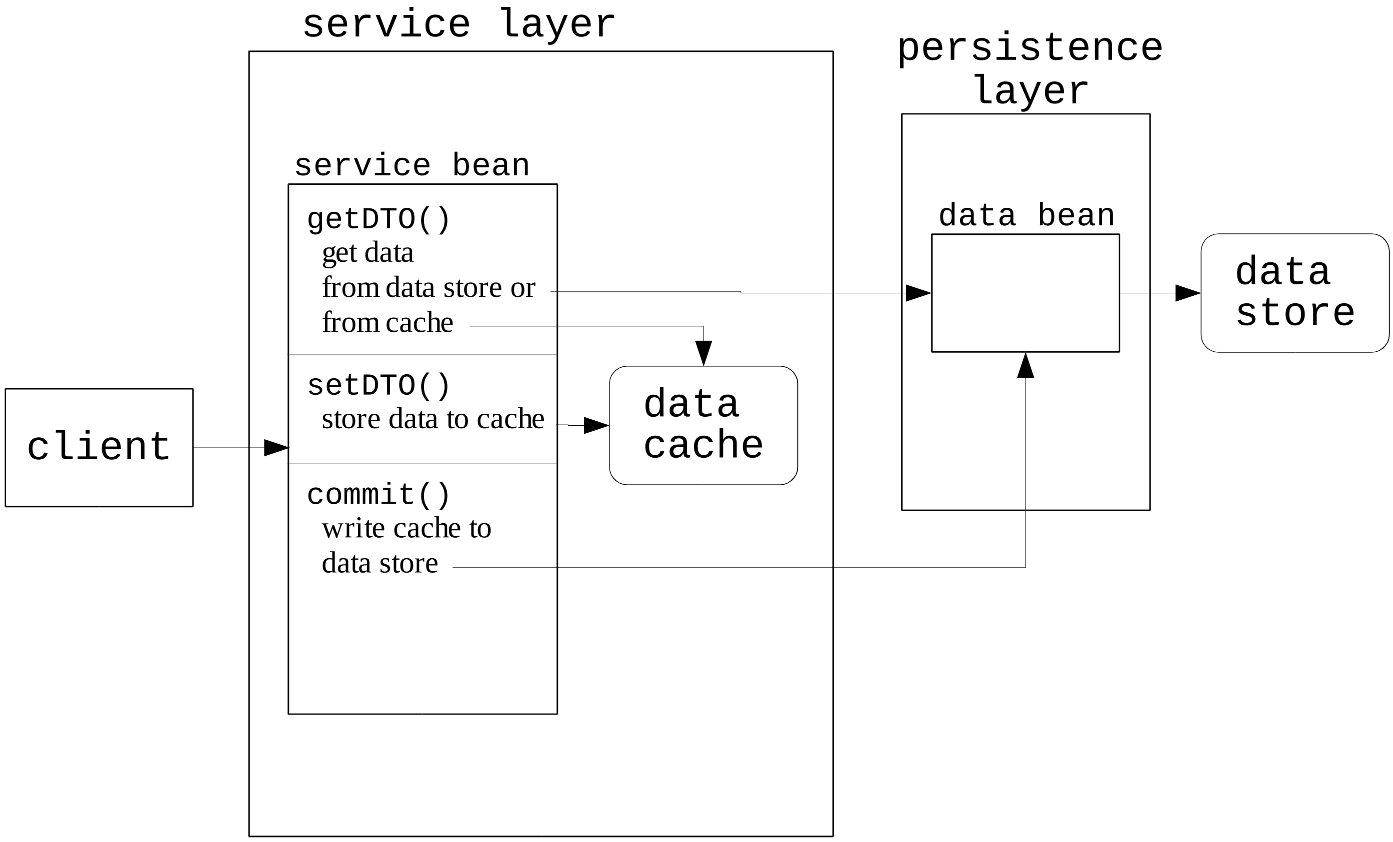
- evaluation:
  - write-through does not work
    - cannot "unset" if data was modified in the meantime

caching

- someone must make provisions for commit/rollback
  - client himself
  - servlet in its session context
  - session bean in its conversational state

- can be supported by service layer
  - aggregate all update requests in a cache without performing them
  - satisfy data requests from cache or from data store as needed
  - when client indicates "commit"
    - trigger all aggregated updates in one TX with staleness check for each update

service layer support – implementation



service layer support – details

- `begin()`
  - fetches all TX-relevant data from DB and places it in cache
    - might happen implicitly with first call to `getDTO()`
- `getDTO()`
  - passes "smaller" portions of data directly from cache to client
- `setDTO()`
  - puts "smaller" portions of data into cache
- `commit()`
  - flushes cache into DB
    - might happen implicitly
  - fails in case of version mismatch
    - EJB TX management automatically triggers rollback of already flushed data

limitations

- placing *all* TX-relevant data in cache at TX begin
  - infeasible when lots of data is (potentially) involved in TX
  - solution only reasonable
    - where "small" amount of data is TX-relevant
  - solution does not work
    - where client can navigate large portions of database with TX
  - lazy caching is not an alternative
    - i.e. filling cache in several steps as needed
  - leads to lack of isolation
    - cache could contain inconsistent data
    - because other clients might have modified data between snapshots

implement "add data"

- presented solution still rudimentary
  - caches only *update* requests
  - how about request to *add* or *remove* data?

- *add* requests already covered:
  - *add* request is cached as *update* request with data version #0
  - staleness check must fail if in the meantime another client added the data element in question
    - data in data store will have version #1 or higher
    - data to be added has version #0
    - version mismatch leads to failure of staleness check

implement "remove data"

- *remove* requests must be cached
  - must add information about type of operation (update, add, remove) to cache
  - staleness check must fail if in the meantime another client removed the data element in question
    - data to be removed does not exist in data store
    - leads to failure of stateness check

isolation level

- isolation and atomicity problem solved
  - by means of staleness check and postponed operations
- one restriction remains: *phantom reads*
- levels of transactional isolation
  - dirty reads
    - read uncommitted changes made by another TX
    - might later be rolled back by the other TX
  - nonrepeatable reads
    - subsequent read in same TX yields different result
    - can see committed changes made by another TX
  - phantom reads
    - subsequent read in same TX yields larger result set
    - because data was added by another TX

phantom read

- example:
  - service shall add a bonus to all customers
  - staleness check prevents that modification is made
    if any of the customers was concurrently modified by another TX
  - if another TX adds a customer we would not notice
    - all staleness checks would succeed
      since no customer was modified by the other TX
    - yet logically the operation failed
      since not all customers received their bonus

optimistic locking – evaluation

- optimistic locking is not as simple as it looks at first sight
  - easy for the service layer implementer
  - puts the burden onto clients and end users
    - in case of multiple get/set operations

- without service layer support (caching)
  - client remains responsible for atomicity (i.e. commit or rollback)
  - must retry each failed operation until success
- with service layer support
  - atomicity is achieved
    - either all updates are made or none of them
  - but postponing operations increases likelihood of staleness
    - whole TX will fail more often
  - additional overhead; decreased performance

agenda

- transactions
  - atomic services
  - client-initiated TX
  - optimistic locks
  - pessimistic locks
- case study

pessimistic locks – motivation

- attractivity of optimistic locking decreases
  - when probability of update failure increases
- probability of collisions increases for
  - long-lasting TXs
  - heavily used resources

client shutdown

- another problem:
  - cache is transient
- client shutdown leads to loss of cached updates
  - problematic no matter where cache is held
    - client data
    - servlet session context
    - conversational state of session bean
- means:
  - client can't suspend his work for any extended period of time

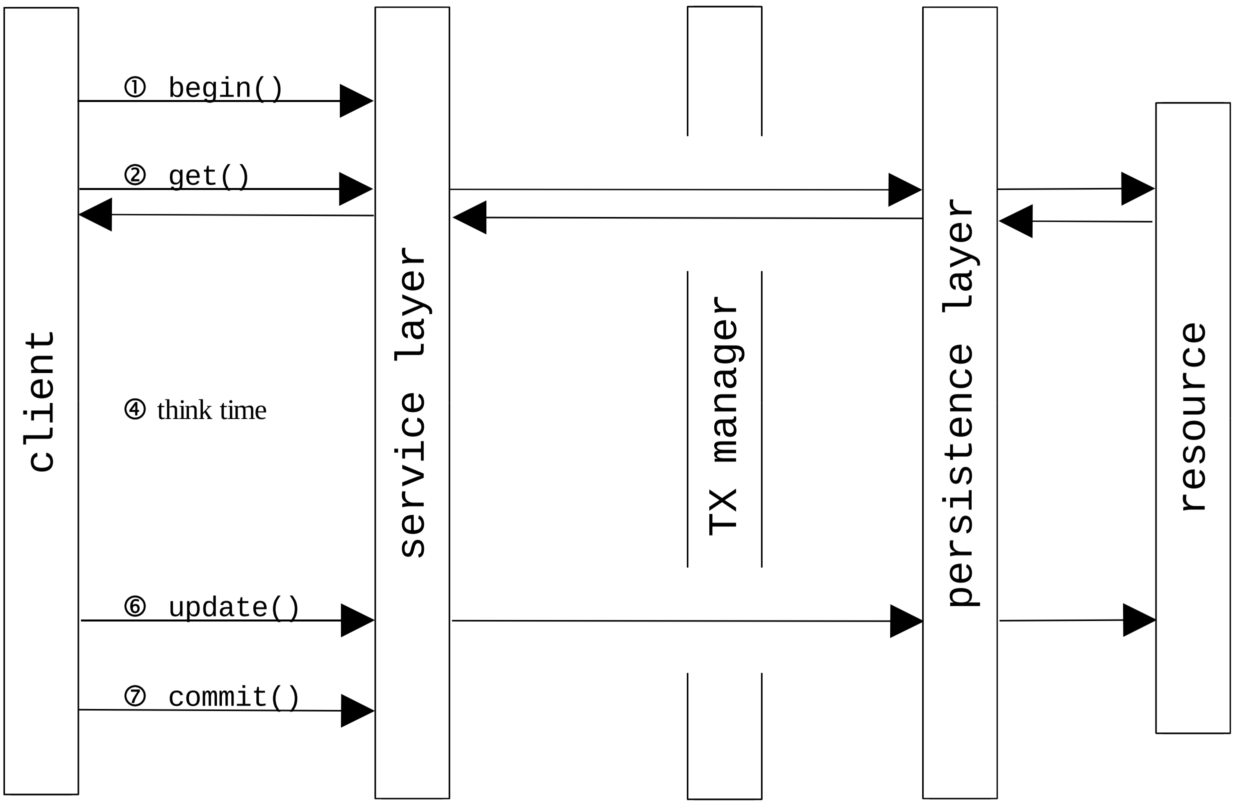
solutions

- eliminate update failures
  - lock critical resources on retrieval already (rather than risking update failures)
  - block out other client access for duration of TX
- make intermediate updates permanent
  - (rather than transient)
  - store initial state of resource
  - make updates directly to data store
  - in case of rollback restore initial state

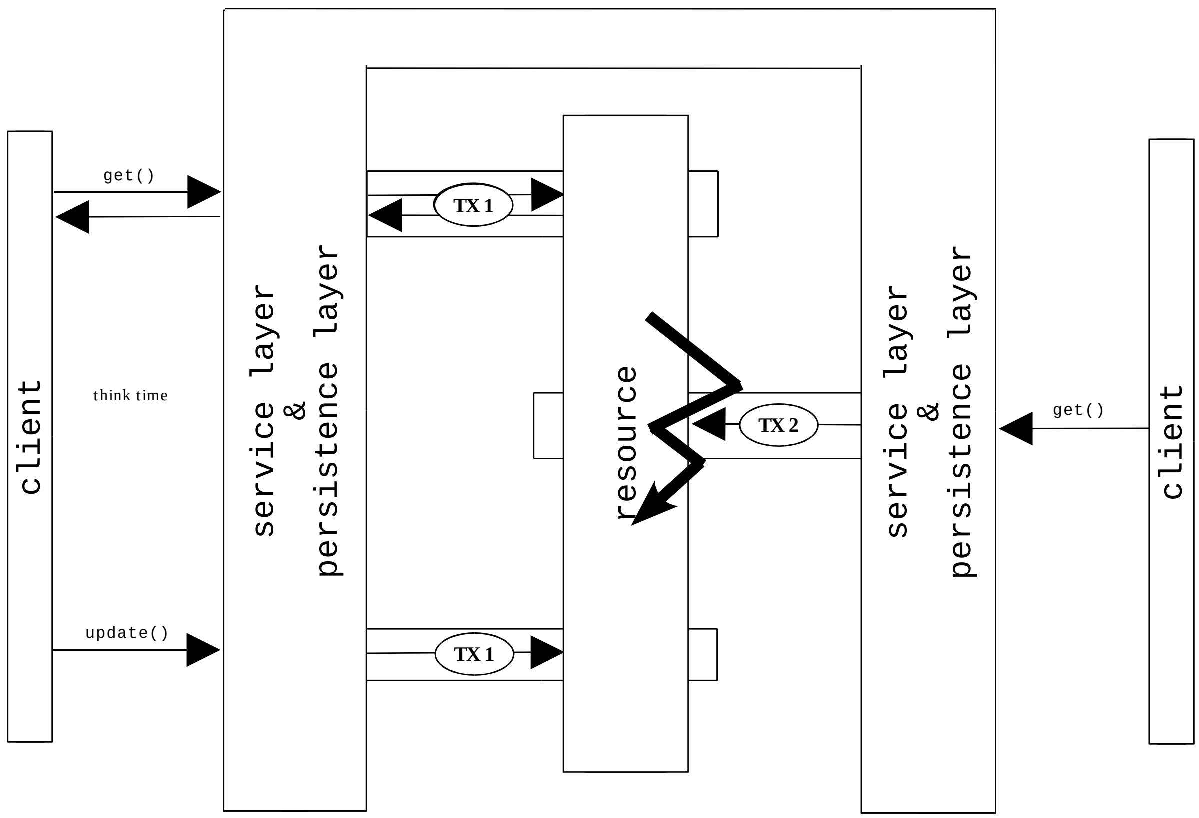
pessimistic locks – goals

- leads to implementation of application-specific TX management
  - often integrated into workflow management
- support long-lasting user-level TXs on top of EJB
  - EJB manages short TXs on bean level
  - user TXs can span several days (or even weeks)
  - user TXs permit
    - begin TX
    - suspend & resume TX
    - close TX with commit or rollback

pessimistic locking



collision

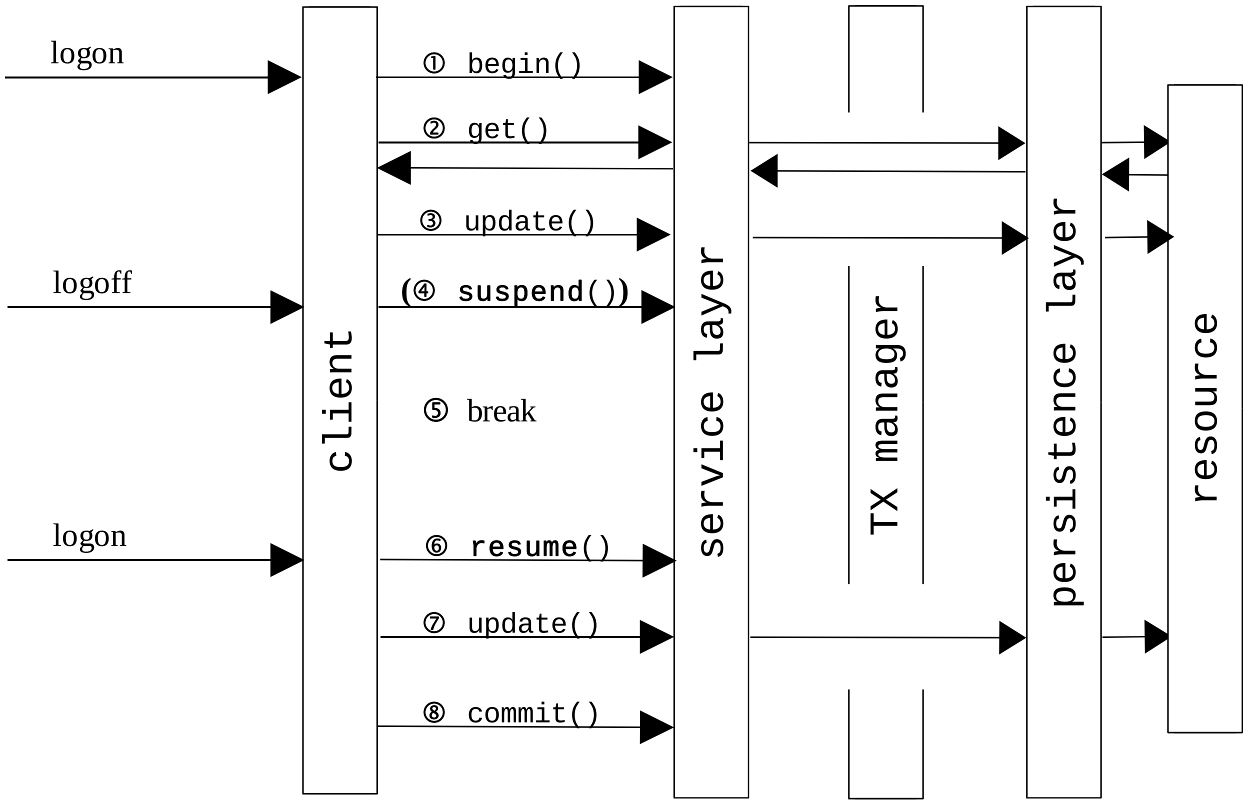


suspend / resume

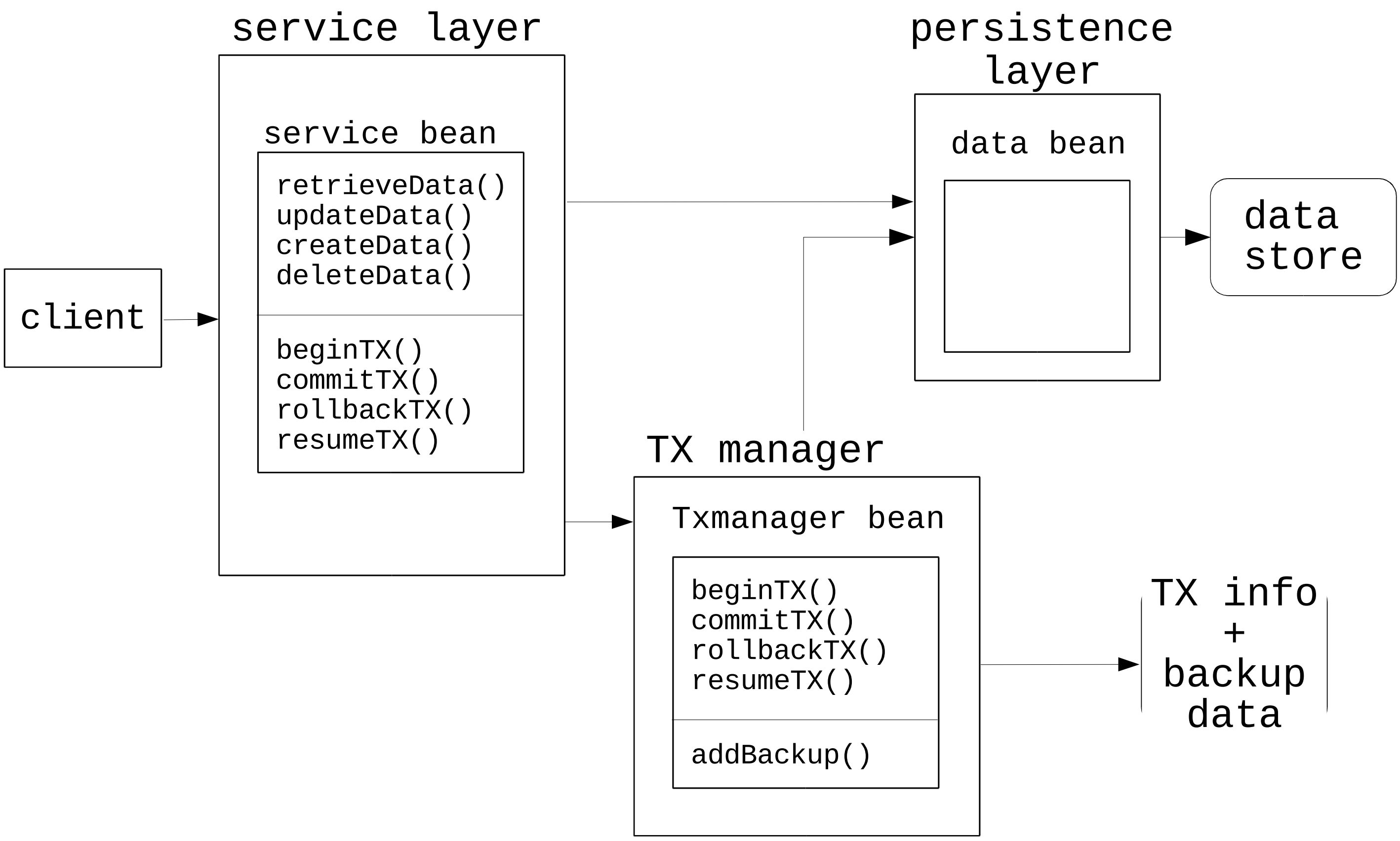
- allow suspend and resume
  - to permit client shutdown or logoff

- suspend can be implicit
  - client simply walks away

suspend / resume



pessimistic locking – implementation



use case: TX begin

- client:
  - initiates TX begin and provides his user id
- TX manager:
  - creates and returns new TX id

use case: retrieve data

- client:
  - requests data and provides his TX id
- persistence layer:
  - reads data
    - succeeds if data is locked for this TX or not locked at all
    - fails if data is locked by another TX
  - locks data for this TX
- TX manager:
  - stores current state of data for subsequent rollback

use case: update data

- client:
  - provides data for update along with his TX id
- persistence layer:
  - performs data update
    - succeeds if data is locked for this TX
    - fails if data is locked by another TX or not locked at all
      - can only happen if client did not previously retrieve the data for his TX
    - *design decision:*
      - *update is only allowed on previously retrieved data*
    - data remains locked
- TX manager:
  - no operation
    - initial state has already been stored on retrieval

use case: TX commit

- client:
  - requests commit
- persistence layer:
  - unlocks all locked data for this TX
- TX manager:
  - closes TX
    - removes all information for this TX id
    - discards initial state of data

use case: TX rollback

- client:
  - requests rollback
- persistence layer:
  - restores initial state and unlocks all data for this TX
- TX manager:
  - provides persistence layer with initial state of data
  - closes TX, i.e. removes all information for this TX id

use case: TX suspend

- client:
  - no operation
- persistence layer: no operation
  - data in data store remains locked for this TX
- TX manager: no operation
  - keeps TX open
  - still remembers all initial data states

use case: TX resume

- client:
  - requests resumption and provides his user id
- TX manager:
  - offers all open TXs for this user id
    - client picks a TX id for subsequent requests

use case: create data

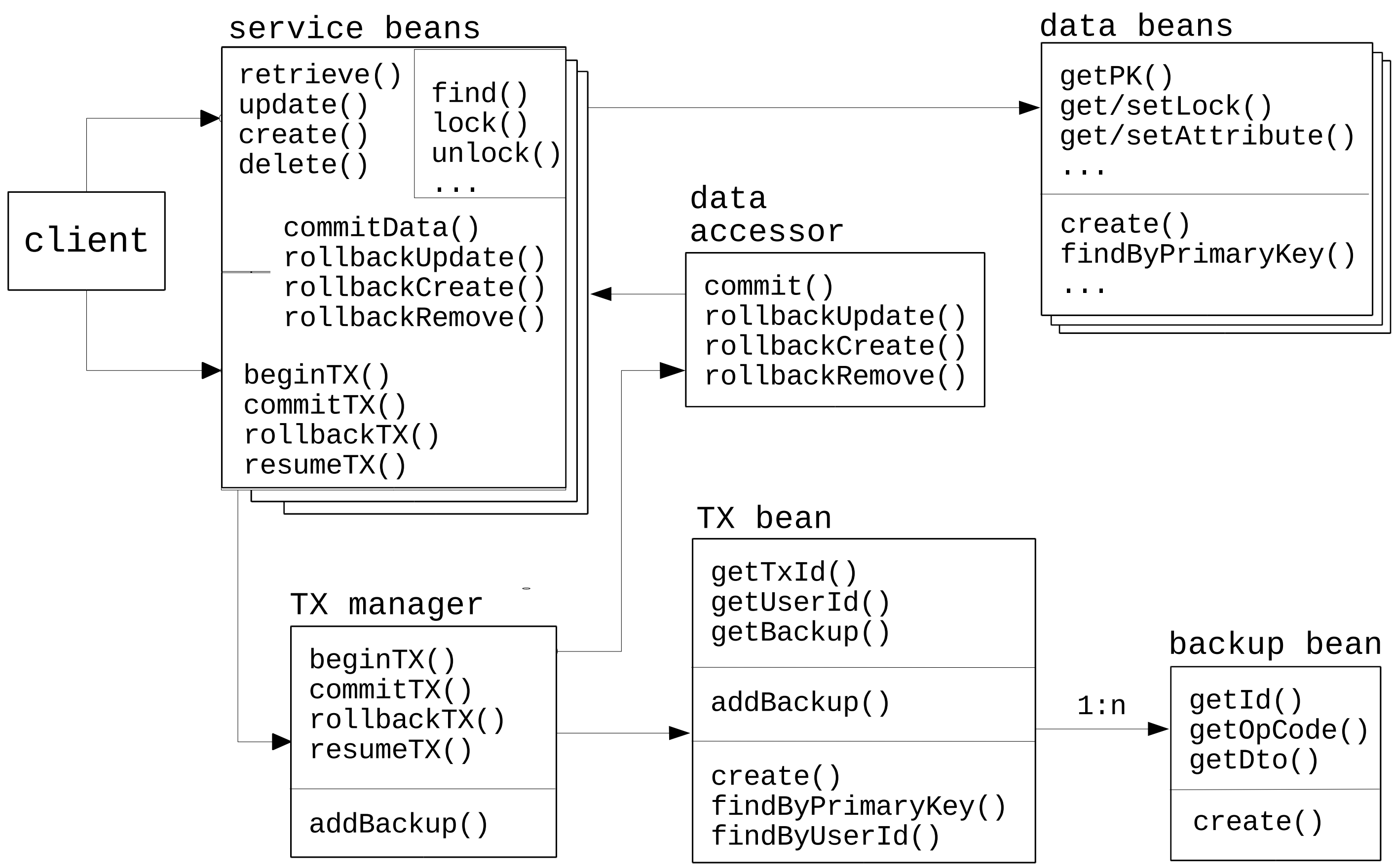
- client:
  - requests creation of new data and provides his TX id
- persistence layer:
  - creates and locks new data entry
    - succeeds if data does not yet exist
- TX manager:
  - takes a note that data must be removed on rollback

use case: delete data

- client:
  - requests deletion of data and provides his TX id
    - *design decision: delete only allowed on previously retrieved data*
- persistence layer: deletes data
  - succeeds if data is locked for this TX
  - fails if data is locked by another TX or not locked at all
    - can only happen if client did not previously retrieve the data for his TX
- TX manager: no operation
  - initial state has already been stored during retrieve

- *data deletion might be problematic*
  - *after deletion in this TX another TX can re-create the data entry*
  - *subsequent rollback of delete (i.e. insert) in this TX will fail*

pessimistic locking – implementation



service layer

- service session bean(s)
  - offers TX related (remote) services to client
  - delegates TX related tasks to TX manager

  - offers data related (remote) services to client
  - uses persistence layer for data related tasks

  - offers (local) support for commit and rollback to data accessor
  - provides additional data related (local) functionality for internal purposes

- "session bean(s)" means "many *types* of service beans"
  - naturally there are many bean *objects* anyway

persistence layer

- data entity bean(s)
  - maintain additional field for locking
    - empty if not locked
    - contains TX id if locked

PK	actual data			lock
#8374	Egon Ochse	Schulweg 43	52687 Köln	-
#2019	Erna Artig	Dorfstr. 29	48176 Castrop	#73632
#1047	Elke Unruh	Hauptstr. 6	72946 Ulm	#18374
#8265	Hugo Hurtig	Uferstr. 72	92834 Arzberg	-

- again: "data bean(s)" means "many *types* of data beans"

TX management

- TX manager
  - implements all TX related tasks
  - talks to the data accessor
    - accepts initial state of data for backup whenever data gets locked
    - provides data accessor with data backups for rollback
    - triggers data unlock on commit
  - maintains TX book keeping in a data store
- TX entity bean
  - keeps information per TX
  - 1:n relationship to backup entity bean
- backup entity bean
  - keeps information about initial state of data for rollback

TX information

TxId	UserId	Backups: PK	DTO (as BLOB)	Typ	Op
#92476	Sales	1	#4711 | Hein Doof | Schratweg 8 | 47362 Boxberg	ADDR	OLD
		2	#9283 | Bodo Blöd | Dorfstr. 25 | 82736 Sellrain	ADDR	OLD
		3	#7236	ADDR	NEW
		4	#6263 | Ilse Ulkig | Bachallee 3 | 54294 Frauenau	ADDR	OLD
#73632	Shipping	24	#8345 | Lola Lustig | Katerweg 4 | 81736 München	ADDR	OLD
		27	#382 | Hermann Hesse | Steppenwolf | pbck | 11.40	PROD	OLD
#18374	Sales	103	#8374 | Hugo Hurtig | Schulweg 43 | 52687 Köln	ADDR	OLD
		274	#2019 | Anna Artig | Dorfstr. 29 | 48176 Castrop	ADDR	OLD
		625	#1047	ADDR	NEW
		187	#8265	ADDR	NEW

backups

- challenge:
  - find a generic solution (independent of the actual data)
  - store initial state of data as BLOB
  - pass around as opaque DTO types

data transfer objects

- data passed around as opaque DTO
  - among service bean, data accessor and TX manager
- TX manager
  - never unwraps DTOs
  - stores them as BLOBs
  - iterates over backup BLOBs for commit/rollback
- data accessor
  - unwraps DTOs
  - knows how to identify them
  - dispatches them to "their" respective service bean
  - triggers unlock (on commit) and storage in data store (on rollback)

note

- TX manager and data accessor
  - can be implemented as stateless session beans
  - but equally well as plain Java classes with static methods
  - because all functionality is reentrant
    - no method needs any data beyond the arguments passed to the method

- open issue:
  - generation of user id
    - authentification and authorization via JAAS

workflow management

- pessimistic locking often embedded into workflow
  - role (in workflow) = user (in our model)
    - workflow assigns roles and associated resources
    - addresses the authentification and authorization issue
  - stages (in workflow) = long-lasting TX (in our model)
  - browsing (in workflow) = begin or resume TX (in our model)
    - displays all possible activities, including suspended work

- pessimistic locking not just an implementation detail
  - embedded into organisational/domain model of workflow

pessimistic lock – evaluation

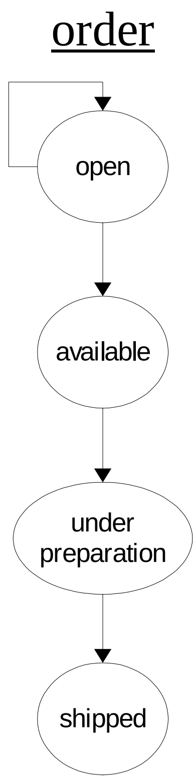
- complex technique
  - requires manual TX management
    - registration of locked resources and their owners
    - permanent storage of data backup for rollback
  - requires interception of *all* access to *all* resources
    - in business services or persistence services or elsewhere
  - difficult to implement completely generic support
    - acquisition and release are best integrated into workflow logic
  - data deletion is a problem
    - still have phantom reads

agenda

- transactions
  - atomic services
  - client-initiated TX
  - optimistic locks
  - pessimistic locks
- case study

TXs in practice

- TX techniques often used in combination
- case study: online bookstore



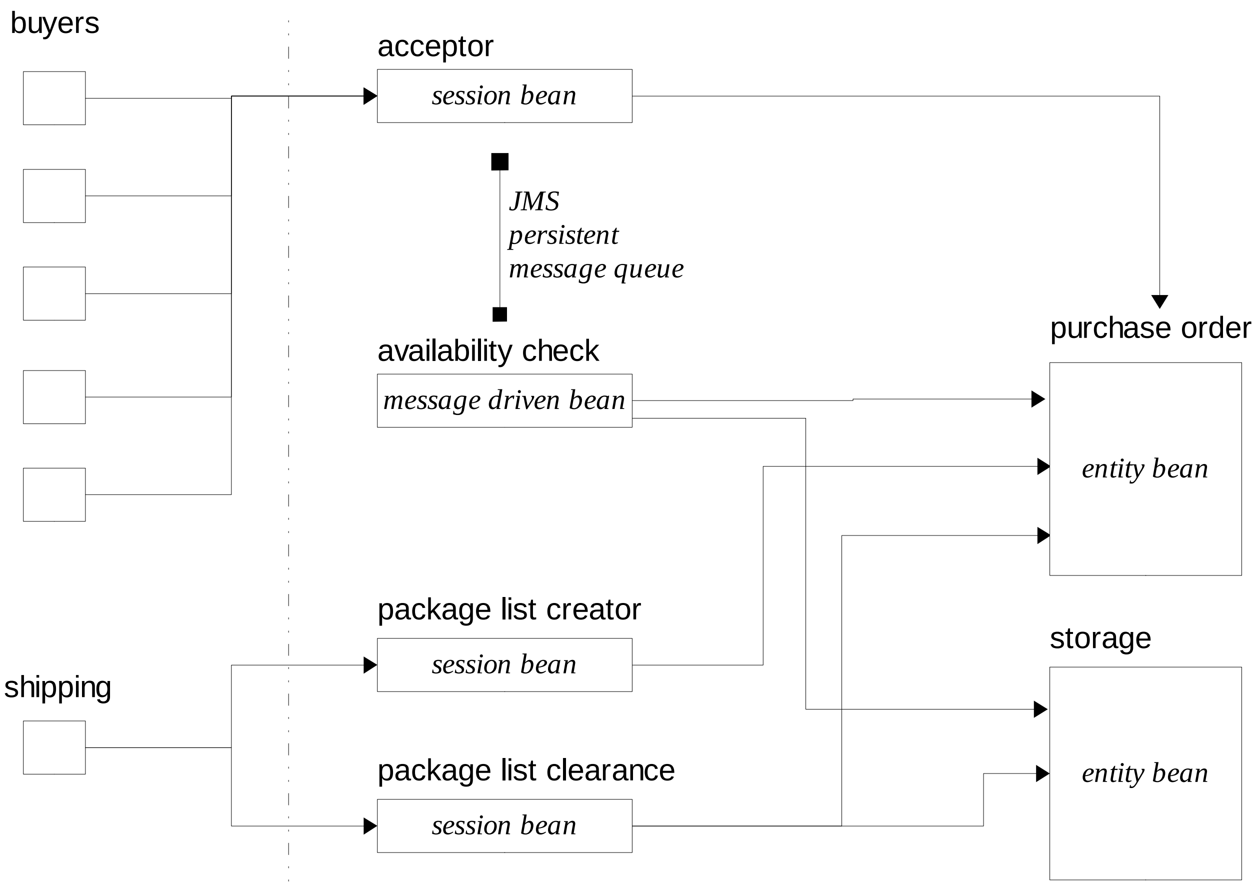
→ open
buyer: creates purchase order

open → available
automated: check availability and reserve items

available → under preparation
shipping: create packaging list and trigger manual processing

under preparation → shipped
shipping: clear for shipping and remove reserved items from store

online book store



atomic service for buyers

- *buyer*: places order
  - accumulation of input data (shopping cart, credit card #, ...)
- no TX, no persistent data
- *acceptor*: creates purchase order
  - creates persistent representation of order for further processing
- container-managed TX comprises:
  - processing by acceptor bean
- non-transactional dialog with buyer
  - implemented as servlet or stateful session bean
  - minimal burden on app server, maximum burden on buyer

CMT for automated availability check

- *acceptor*: triggers availability check
- container-managed TX comprises:
  - creation of persistent representation of purchase order
  - creation of message for availability check
- *availability checker*: performs availability check
- container-managed TX comprises:
  - check for availability and reservation of items in store
  - modification of status of order (to *available* or *pending*)
- sending and receiving a message protected under CMT
  - uses persistent message queue
  - e.g.:
    - failure to send message rolls back creation of order and results in user-visible error
    - rollback in case of reservation failure puts message back into queue

atomic service for administration of shipping

- *shipping*: creates packaging list
  - triggers manual labor of assembling & packaging
- container-managed TX comprises:
  - creation of packaging list
  - modification of status of order (to *under preparation*)
- *manual labor*: physical act of assembling & packaging
  - packaging list serves as "lock"

- short unit of work
  - no dialog necessary

atomic service for administration of shipping

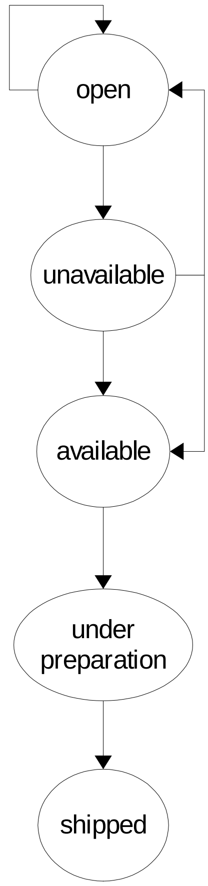
- *shipping*: clearance for transport
  - ends manual process of packaging
  - confiscates packaging list (alias "lock")
- container-managed TX comprises:
  - update of storage: remove reserved items
  - modification of status of order (to *shipped*)

- short unit of work
  - no dialog necessary
  - if so, no persistence needed
  - e.g.:
    - does the packaging list belong to an order that is "under preparation"?

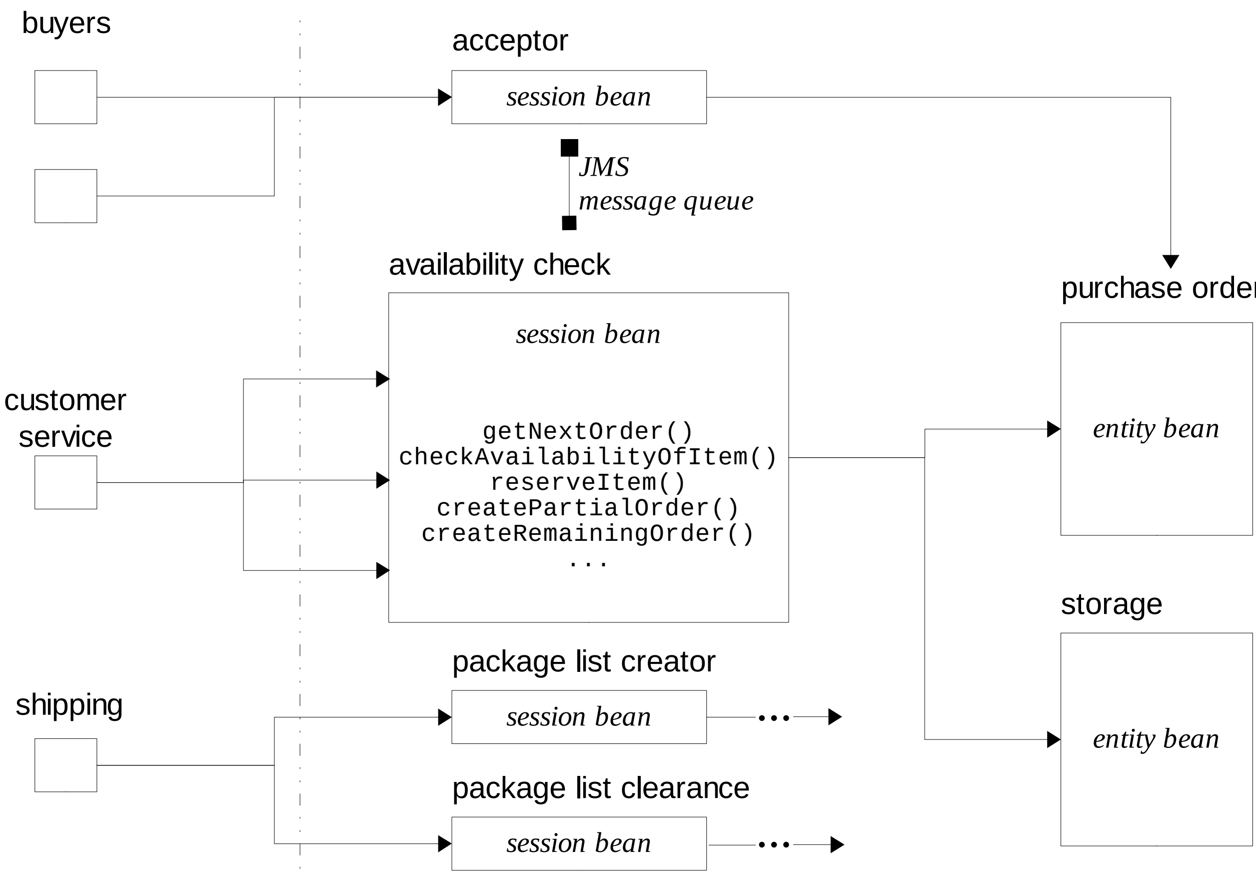
different scenario

- what if not all items are available?
- improve customer service:
  - call buyer and ask for further action
  - split order / cancel order / new order / postpone shipping
- requires human worker

- introduce new actor
  - *customer service*
- add new status
  - for partially (or fully) unavailable orders
    - further status conceivable: cancelled, pending, ...



online book store



client-initiated TX for customer service

- *customer service*: handles "unavailable" orders
  - calls buyer and decides on further action
- client-initiated TX comprises:
  - retrieval of "unavailable" purchase order
  - manual labor (phone call)
  - creation of partial order or removal of cancelled order
  - reservation of further items or cancellation of reservations
  - modification of status of order (to *available*, *open*, or *pending*)

discussion

- in favor of client-initiated TX
  - "unit of work" spans several activities
    - made atomic via long-lasting TX
  - status of order expressed in terms of TX lock
    - rather than in terms of status chance of order in DB
  - customer service intervention is rare
    - most orders can be processed automatically

wrap-up

- discussed implementation of logic TX on top of EJB
  - EJB supports fine grained TXs tied to bean methods
  - of actual interest are TXs tied to end-user interactions
- natural EJB approach:
  - fine-grained bean-level TXs with CMT / BMT
- alternative:
  - coarse-grained client-initiated TXs with JTA/JTS
- user-implemented TX are more complex
  - optimistic locking
  - pessimistic locking

contact info

Angelika Langer

Training/Consulting

Email: `info@AngelikaLanger.com`

`http://www.AngelikaLanger.com`

Klaus Kreft

Siemens Business Services

Email: `klaus.kreft@siemens.com`