UNIVERSITY OF Cincinnati

Offline Concurrency Patterns (Chapter 16)

Mike Helmick
Large Scale Software Engineering
Spring 2014

Chapter 16

Chapter 16

- Optimistic Offline Lock
- Pessimistic Offline Lock
- Coarse-Grained Lock
- Implicit Lock

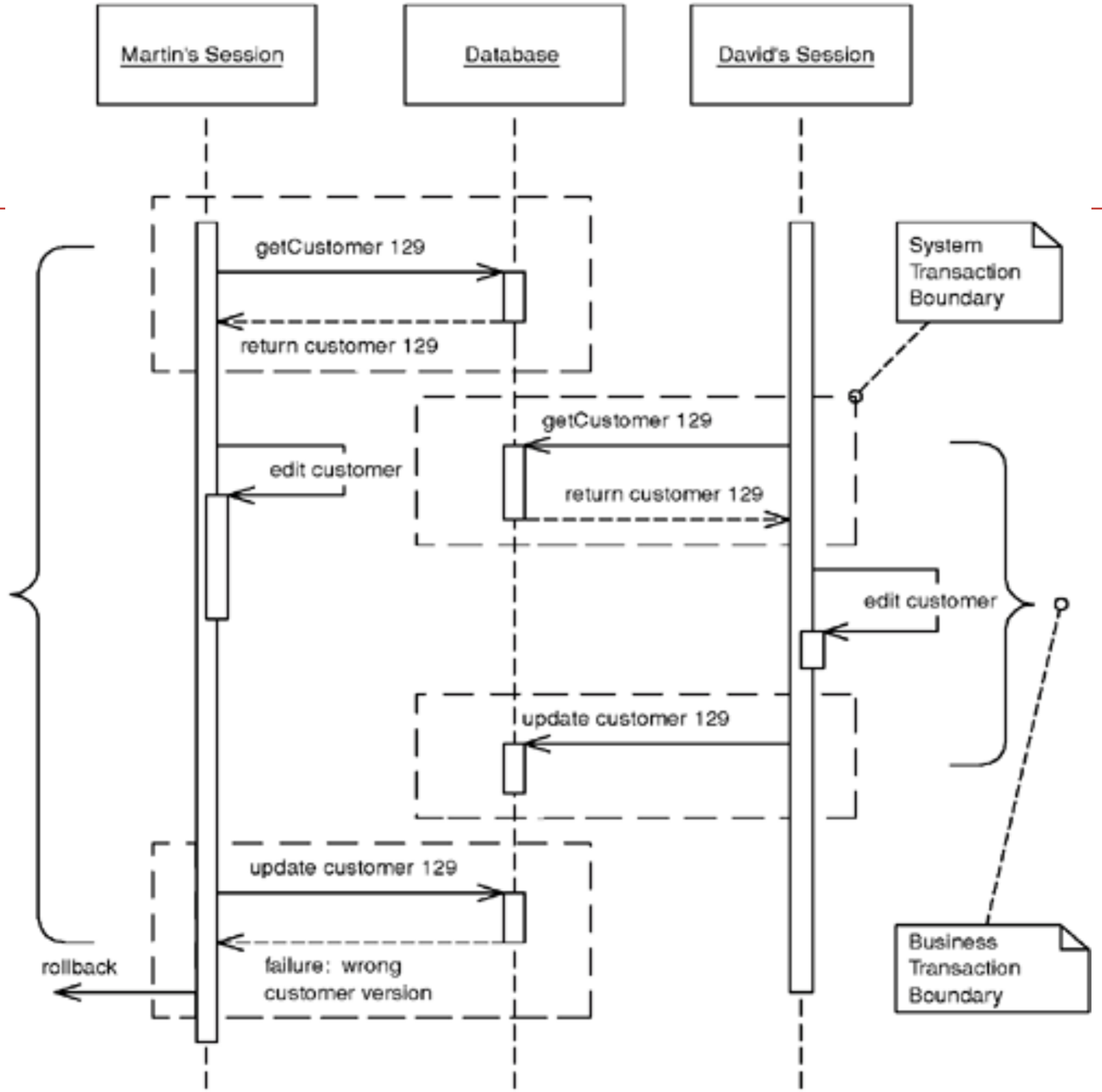
Optimistic Offline Lock

Optimistic Offline Lock

Optimistic Offline Lock

- “Prevents conflicts between concurrent business transactions by detecting a conflict and rolling back transactions.”

Optimistic Offline Lock



Optimistic Offline Lock

Optimistic Offline Lock

- Business transactions will normally span several system items
    - Some service calls will trigger several changes in the domain model
    - Two (or more) long running transactions occurring simultaneously

Optimistic Offline Lock

Optimistic Offline Lock

- Solves the concurrency problem by validating that the changes of this session do not conflict with other sessions
    - just before commit
- Assumes that the risk of conflict is low

How It Works

How It Works

- Optimistic Locks are not really locks at all
    - Validate that the data that was read is still the data that is in the database before the update
- Will ensure that no inconsistent data is written
    - but only after the transaction is finished processing

How It Works

How It Works

- Generally
  - this is implemented with a version number field on each row
  - compare the version number that was read on update
  - update the version number to a new one

SQL

SQL

- update table set ....., version_number=?
  where version_number=?

- using verNbr + 1 and verNbr as the bind variables

- Do this on updates and deletes

Row Count

Row Count

- In order to verify that the execution was successful, verify the row count
- For an update or delete of an identity row
  - the row count should be 1
- execute update returns a row count

Audits

Audits

- It can also be useful to store the user id of the person logged in when the row was last updated
- Then - you can also give the user a good error message
    - “Cannot update row - recently updated by X”

Clock

Clock

- Generally timestamp doesn’t work
- why?

Alternative

Alternative

- Always use all fields in the where clause
  - then you don’t need to have a version number column
  - this can cause major performance problems
  - and unnecessarily long where clauses

Reads

Reads

- Offline lock allows for inconsistent reads to occur
    - these are detected on updates
- but not detected on fields that are read and then used in calculations
    - so - a mechanism should be built to verify that the version number is still correct on commit

Business

Business

- Often concurrency is a business issue
- If a customer changes their service plan for instance
  - When their next billing cycle occurs which one do we use
    - the old one or the new one
    - do we have to use both

SCM

SCM

- Source code management systems (mostly) use optimistic offline lock
- generally OK because commit conflicts can be automatically (or manually) merged

When to Use It

When to Use It

- Most useful when the change of conflict in two database transactions is relatively low
- You don’t want a user to continuously get version conflict errors
    - In my experience, I haven’t seen this error occur twice in a row

When to Use It

When to Use It

- Easier to implement
    - doesn’t have same runtime consequences as pessimistic offline lock
- You should always consider using optimistic lock
    - and escalate to pessimistic lock in certain circumstances

Example

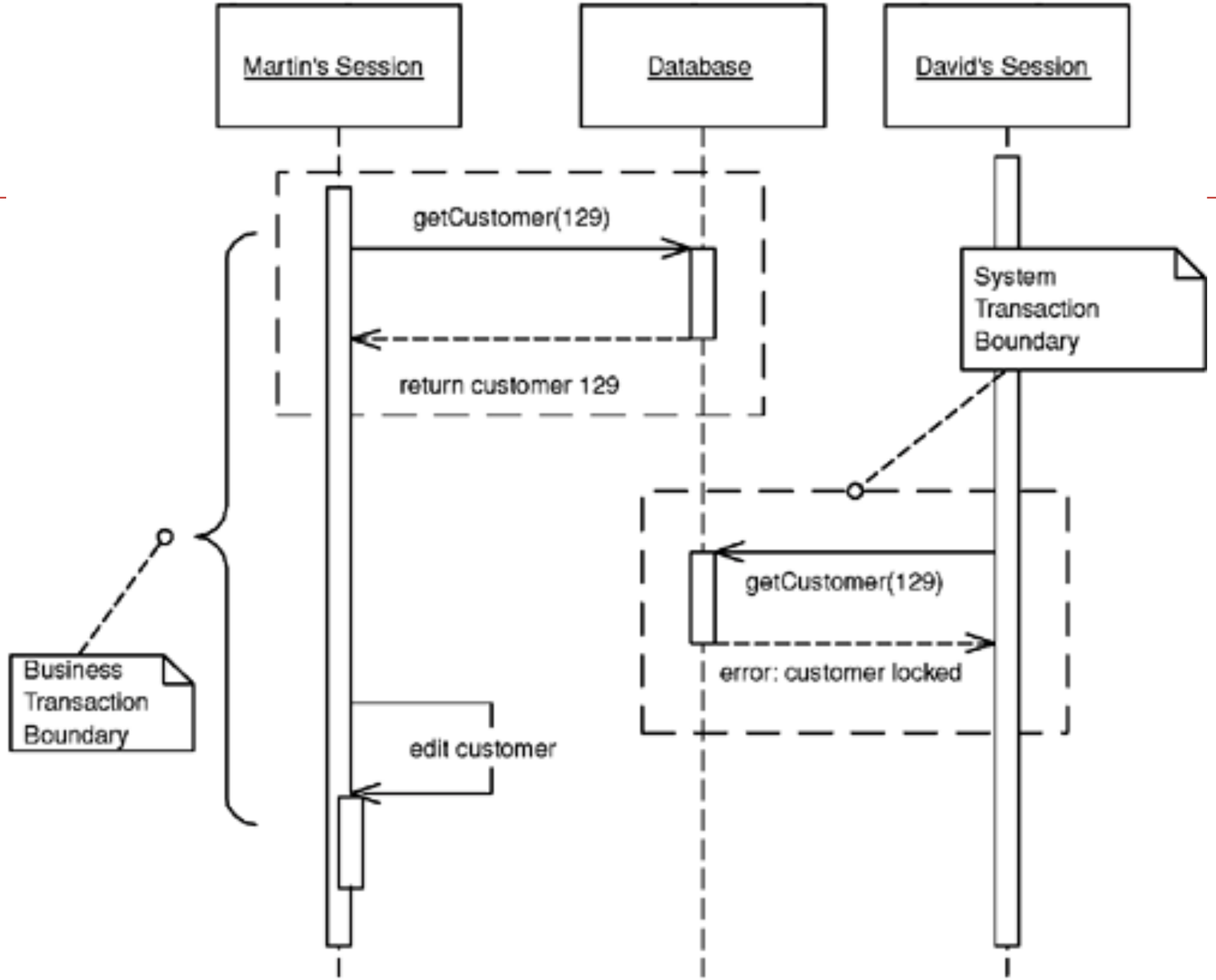
Pessimistic Offline Lock

Pessimistic Offline Lock

Pessimistic Offline Lock

- “Prevents conflicts between concurrent business transactions by allowing only one business transaction at a time to access data.”

Pessimistic Offline Lock



Pessimistic Offline Lock

Pessimistic Offline Lock

- Used for long business transactions that span multiple system transactions
- Optimistic offline lock allows a transaction to run to completion and fail at the last second
- Pessimistic lock prevents conflicts from occurring in the first place

How It Works

How It Works

- Implemented in 3 phases
  - determine types of locks
  - build the lock manager
  - define lock usage semantics

Exclusive Write Lock

Exclusive Write Lock

- Requires a business transaction to lock data only when editing
- Allows other processes to read locked data (and display it to the user)
    - but that user will get an error if they try to open the data for editing

Exclusive Read Lock

Exclusive Read Lock

- Used when you must always display the most accurate information
  - i.e. can’t display data if another process has requested it for writing - or reading and still has the page open
  - Severely restricts the concurrency of the system

read/write lock

read/write lock

- mutually exclusive: a single process can get a read lock or a write lock - no other process can get a lock if another has a write lock

- concurrent read locks are acceptable. As long as a read lock is in effect, anyone can open a read lock. Write locks block

- known as the readers / writers problem

concurrency

concurrency

- read/write locks improves concurrency
- major downsides
  - this is very difficult to implement
  - challenging in deciding which objects need read/write lock
  - challenging in creating an ordering that avoids deadlock

To Consider

To Consider

- Maximize concurrency
- Meet business rules
- Minimize complexity

Pessimistic Lock

Pessimistic Lock

- Can cause problems if implemented incorrectly
- Degrade system performance
- To many locks - escalating locks, etc....
    - in effect reduce you to a single user system

Lock Manager

Lock Manager

- The lock manager is the gateway for lock requests

- Must take in request and decide if it can be granted in the context of the business transaction, system transaction, and possibly the user (lock owner)

Lock Manager

Lock Manager

- A simple structure
    - a tale that maps locks to owners
    - can be in memory or on the database
- Memory - use a synchronized singleton
- Database - works in clustered application environments

Visibility

Visibility

- Locks need to be private to the lock manager
- Business transactions should not be able to create their own locks

Timing

Timing

- In general
  - locks should be required before (as) the data is read
  - order of locks doesn't matter much for read only transactions
  - order is important for write locks!
    - deadlock

What

What

- You need to also decide what to lock
  - Best to lock the IDs - rather than the actual object
  - Then you can get the lock before reading the object (can’t do this on search queries)

Releasing

Releasing

- Locks should be released when the business transaction completes
    - either success or failure
- Some locks can possibly be released early
    - a write lock obtained, but the the object doesn’t change

Errors

Errors

- Since locks will be acquired early in the transaction
    - the user can be notified quicker
- Can possibly implement blocking locks
    - this will cause transactions to sit and wait until they are able to execute

Database

Database

- Storing your locks in the database causes some problems
  - The lock table itself needs to have concurrency control
  - Can help to use database locks in this instance

Blocking

Blocking

- Using something like “SELECT FOR UPDATE” can lead to deadlock
    - Where as offline lock will give an error in this situation

Timeouts

Timeouts

- the last thing to consider is lock timeout
- occasionally a client will crash after locks have been obtained
    - some reasonably timeout should exist

When to Use It

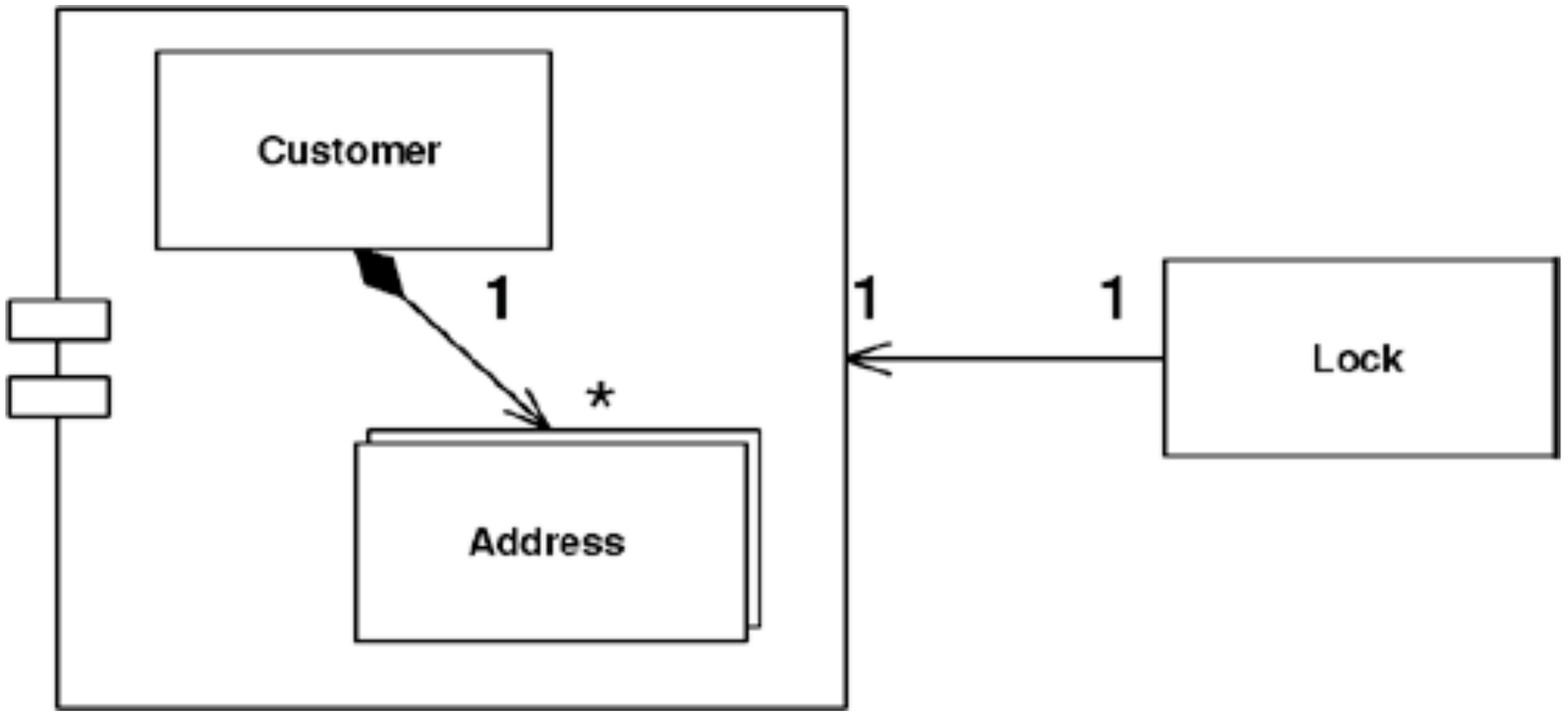
When to Use It

- When the chance of conflict is high
- When you want a client to never sit and wait
    - i.e. the get quicker error messages
- When you have long running transactions

Example

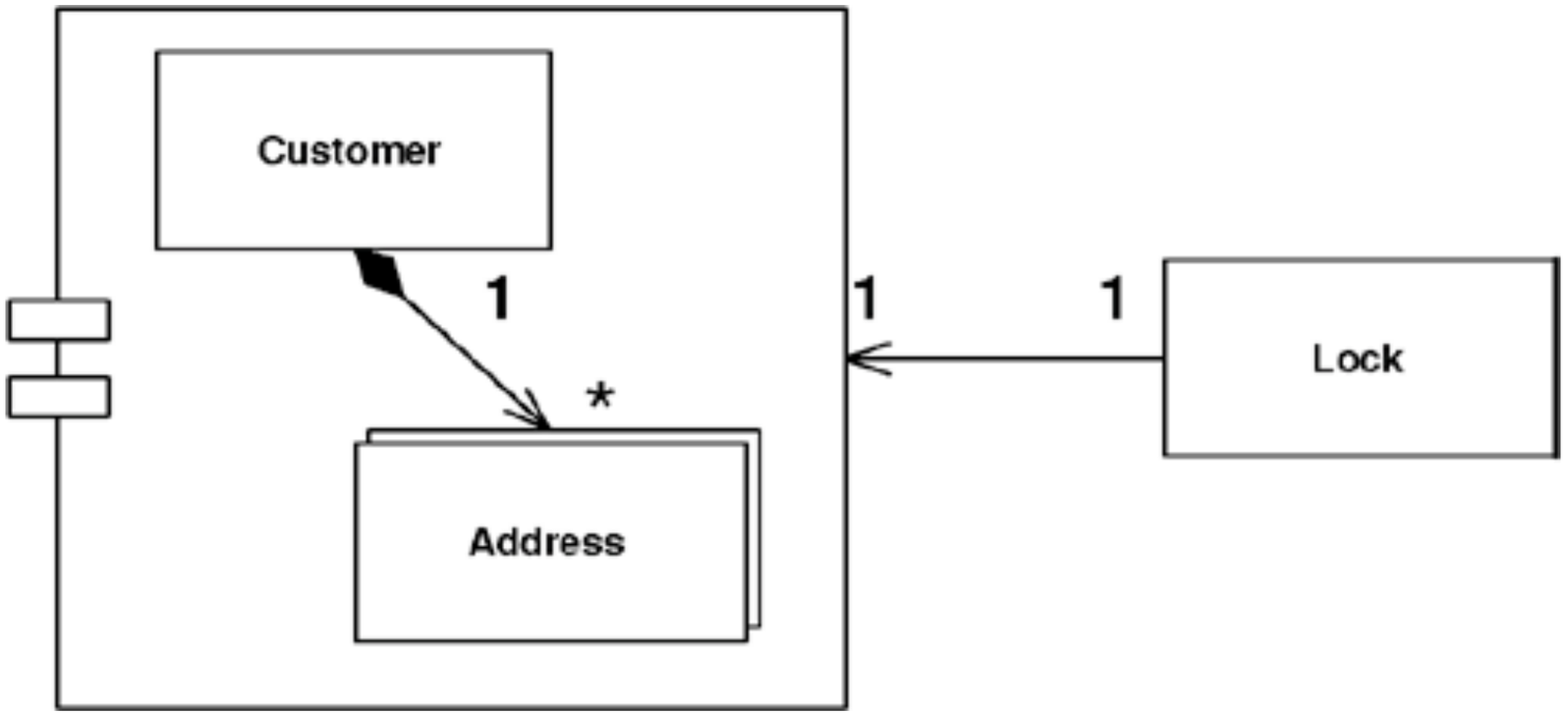
Coarse-Grained Lock

Coarse-Grained Lock



Coarse-Grained Lock

- “Locks a set of related objects with a single lock.”



How It Works

How It Works

- Create a single lock point for a group of related objects
    - Then only one lock is necessary, rather than several

Optimistic

Optimistic

- Use a shared version number for all records in the set
    - when any record in the group is used the shared lock is checked

Pessimistic

Pessimistic

- Every group member shares the same lock id
- Again - shared version number / object is a good candidate for a shared pessimistic lock id

Aggregate

Aggregate

- Data groups have been defined as an aggregate
    - has a root object
        - the lock boundary for the entire group
- Coarse grained lock mediated through the root is appropriate in this case

Aggregate

Aggregate

- Requires that subordinate objects to know the root of the hierarchy

When to Use It

When to Use It

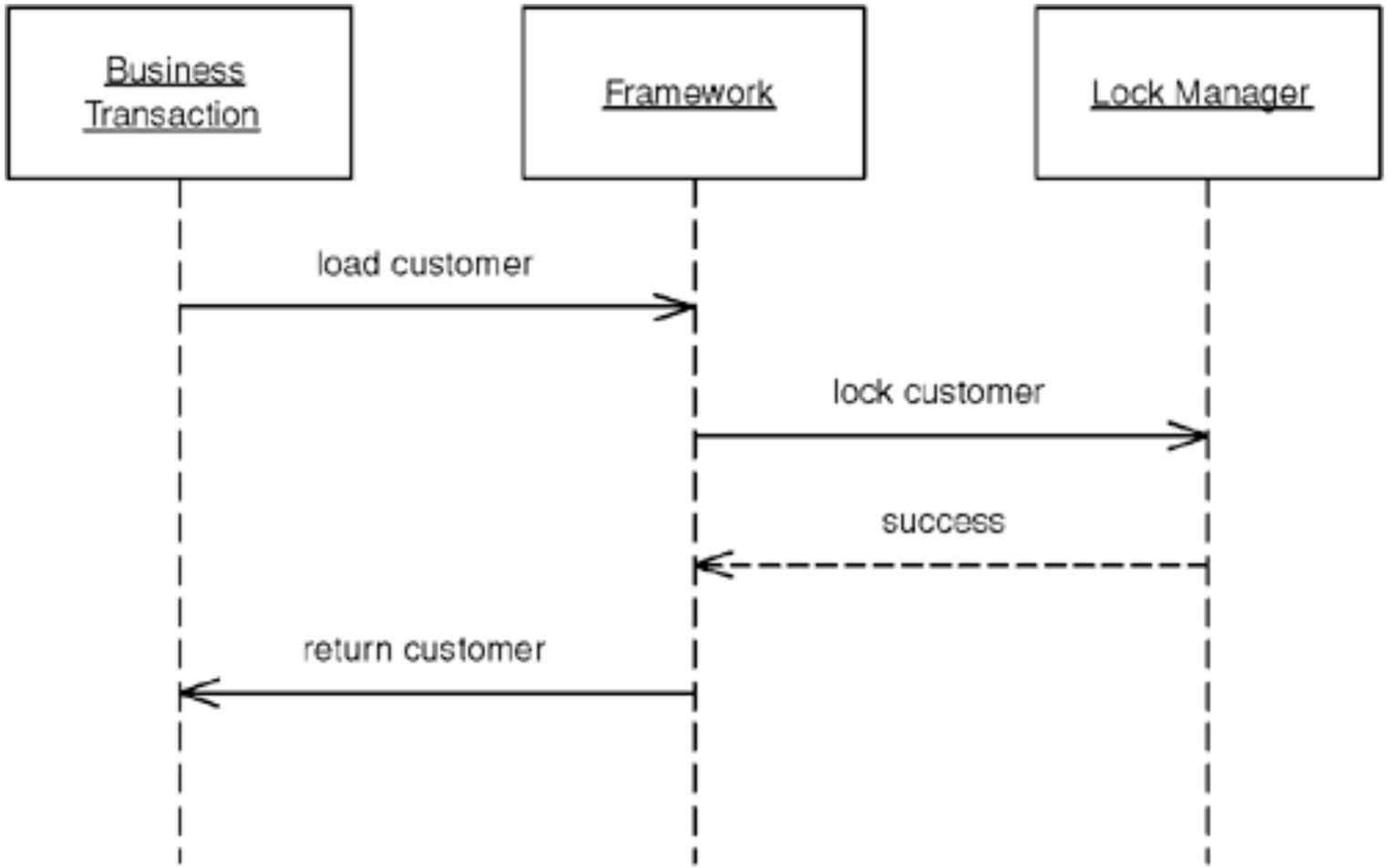
- When business requirements call for it
- The act of locking is quicker - since the amount of records included in the lock is larger
    - so less locks are needed

Chubby

- This is where a distributed lock manage, like Chubby, will be useful

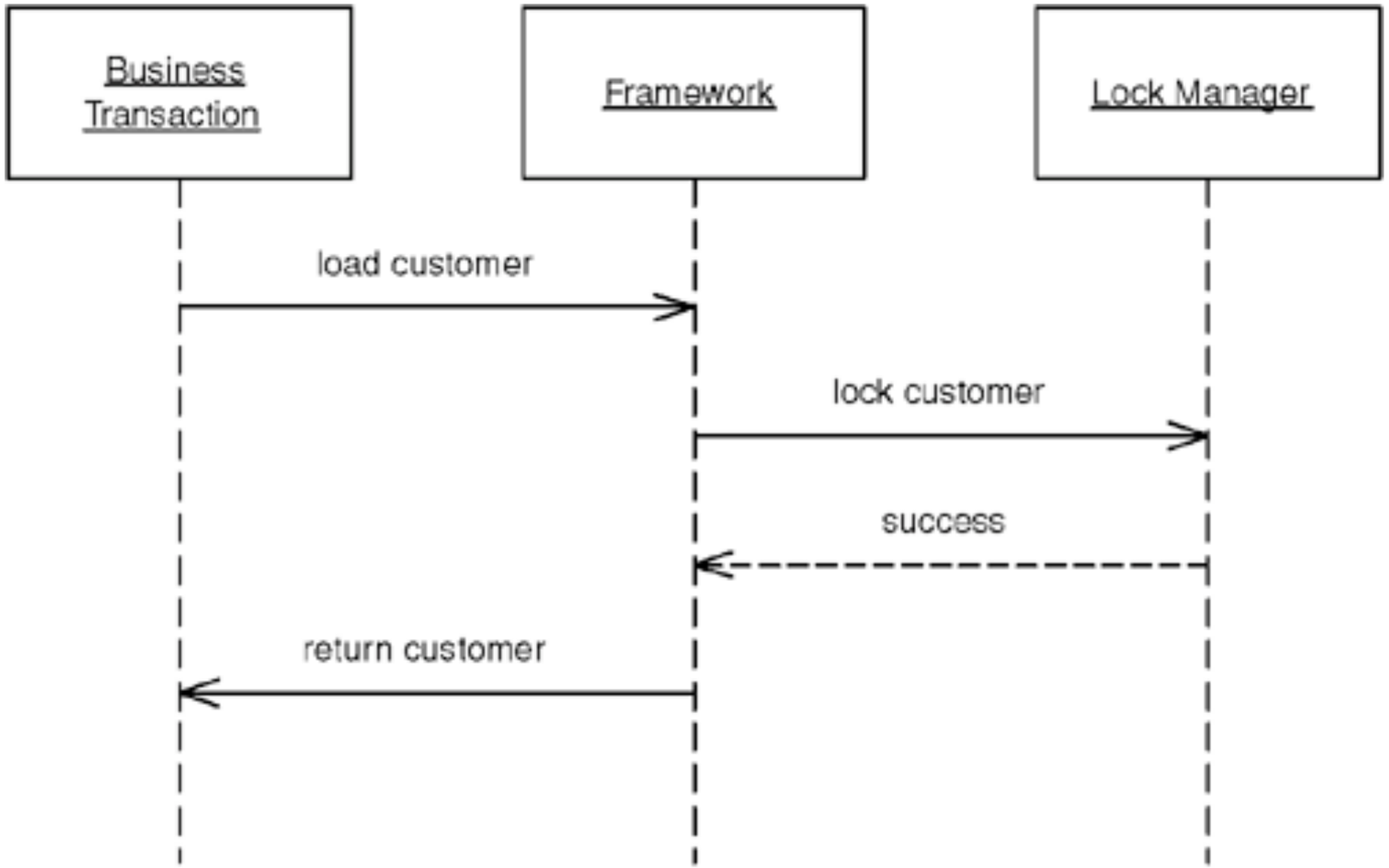
Implicit Lock

Implicit Lock



Implicit Lock

- “Allows framework or layer supertype code to acquire offline locks.”



Implicit Lock

Implicit Lock

- A successful locking mechanism is always in force
    - Best of locking can be done automatically
    - If developers must explicitly code locks, they are likely to forget

How It Works

How It Works

- Implicit lock
  - embedded in your code so that locks can not be circumvented in any way in the code
  - obviously direct SQL on the database can get around this

How It Works

How It Works

- Best if you can hook this into your data source layer framework

Concurrency

Concurrency

- Always using implicit lock can cause problems
    - It is possible that you want to move the locking out of the technical area into the domain area

Danger

Danger

- If using blocking pessimistic locks
    - the developers need to read everything in the correct order

When to Use It

When to Use It

- Whenever you use locks
- One missed lock cause the whole application to have the potential for failure