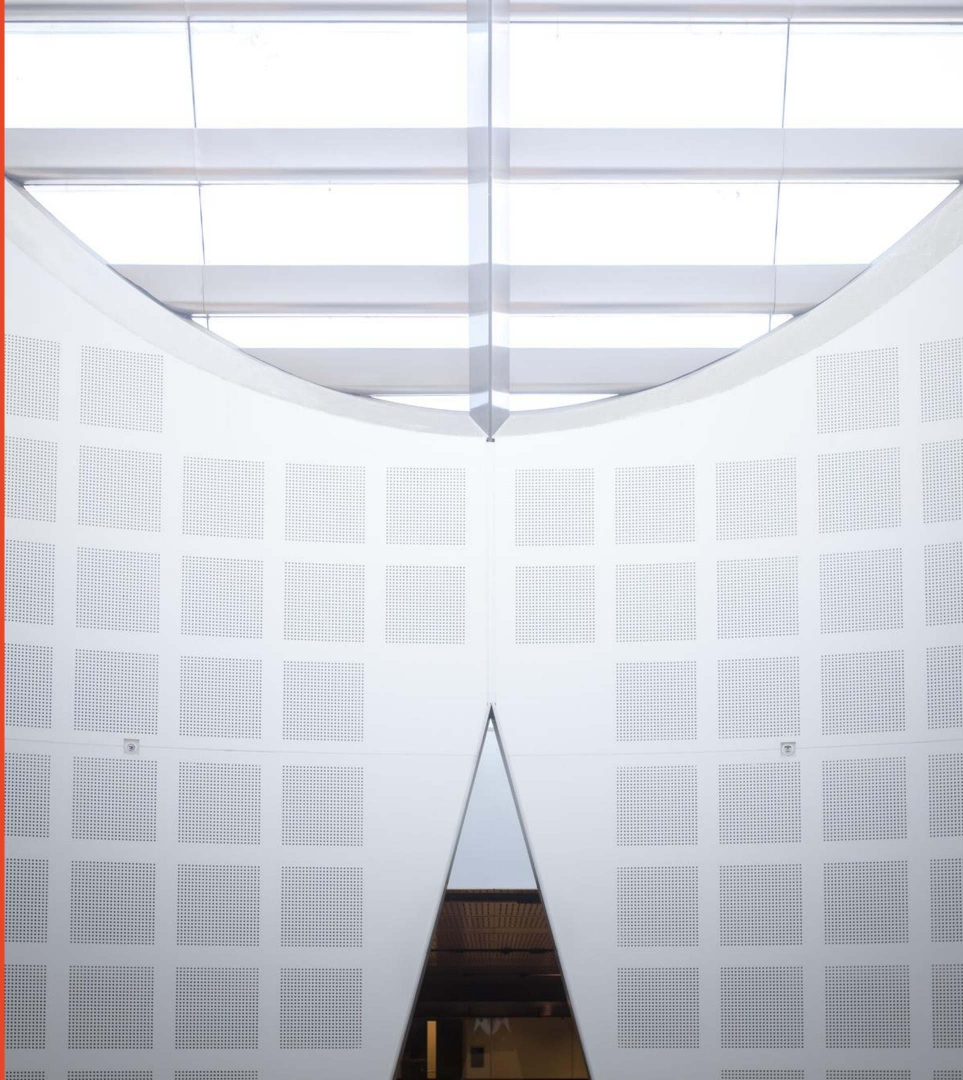
Software Construction and Design 2
SOFT3202 / COMP9202
Enterprise Design Patterns (Concurrency)

Dr. Basem Suleiman

School of Information Technologies

THE UNIVERSITY OF SYDNEY

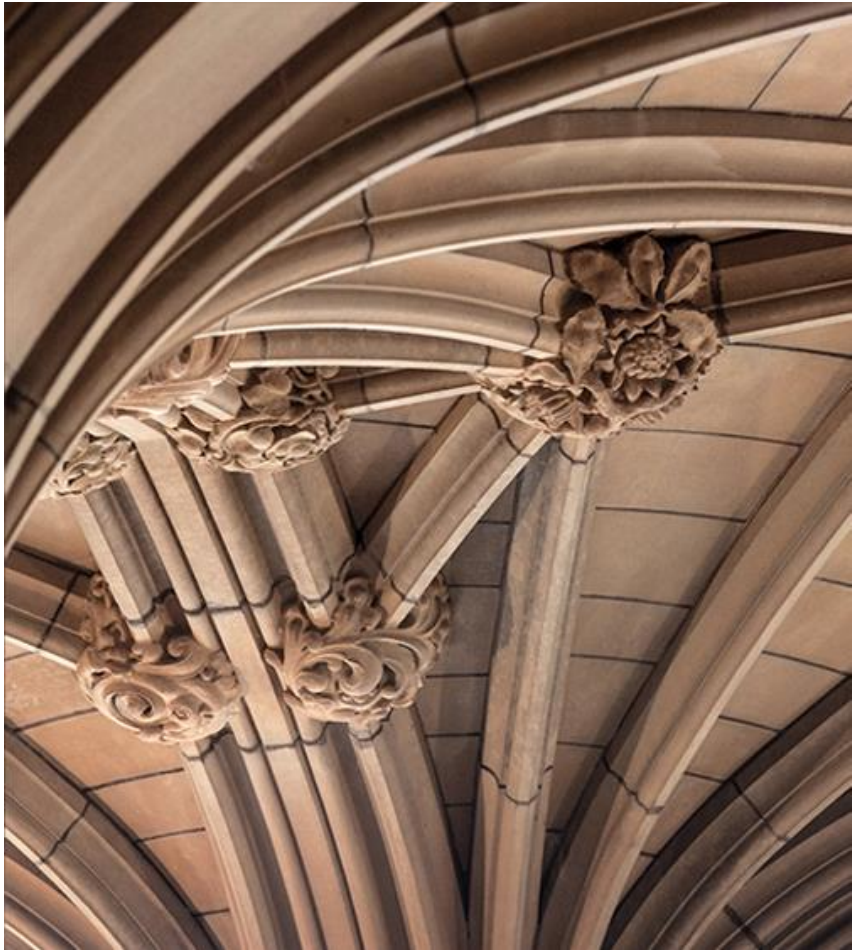


Copyright Warning

Agenda

- Introduction
  - Concurrency in Enterprise Applications
- Enterprise Application Design Patterns
  - Lock
    - Optimistic Lock
    - Pessimistic Lock
  - Thread Pool

Enterprise Applications



THE UNIVERSITY OF SYDNEY

Enterprise Applications

- Manipulate lots of persistent data
  - DBMS (relational DB and NoSQL)
- Concurrent data access and manipulation
  - Design with concurrent programming thinking not only transactions management systems
- Data access and manipulation via different views and contexts
- Integrate with other enterprise applications
  - Legacy and modern ones
  - Using different software and hardware technologies

State

- All imperative programming involves altering state (the values linked to names eg variables)

- In enterprise applications we distinguish several varieties of data that has state which is important
  - Infrastructure state
  - Resource state
  - Session state
  - Local computation state
  - Other

Type of States

- Infrastructure: provides general support for computation (connection pool)

- Resource: data about the real-world domain (salary, manger)

- Session: set of related operations over a period of time (business transaction)
  - Operation may change values of resource state
  - Client session state, server session state or database session state

- Local computation state: temp variables created in programs

Lifetime of State

– Resource state lasts from creation to explicit deletion, through many sessions
– Session state lasts from start of session to completion, through many operations
– Local computation state lasts during one operation

Sharing of State

- Resource state means something about the domain
  - It will be accessed by many sessions
  - Possibly at the same time

- Session state is used by different steps of the session
  - Usually, one after another (but with gaps)

- Local computation state is not shared

Concurrency

- Techniques and mechanisms that enable several parts of a program be executed simultaneously without affecting the outcome
  - Several computations that share data run at the same time
  - Parallel execution of program tasks

Why Concurrency?

- Allow simultaneous access/usage of enterprise applications

- Improve performance (speed)!
  - Allow for parallel execution of the concurrent parts
    - Doing more than one operation at a time
  - Processing while waiting for I/O
  - Using multiple processes

Mechanisms for Concurrency

- Multiple machines/nodes
- One machine with separate processes
- Threads within a process

Concurrency Problems

- When interleaved computations have any shared state that they operate on, things can go badly wrong
  - Interference between programs/transactions
- Conflicts over shared resources
  - Cannot sell the same seat twice
- The state on which the computation depends can be changed unexpectedly
  - By another thread, between steps of the computation

Interleaving

- Each computation follows the code that is written for it
  - Line by line, one after another

- Observed from outside, the computations are interleaved
  - Computation 1 does steps C1.1 then C1.2 then C1.3
  - Now Computation 2 does C2.1 then C2.2
  - Computation 1 resumes with C1.4 then C1.5
  - Computation 2 resumes with C2.3

Concurrency Problems – Lost Update

- Transactions T1 and T2
- T1 wants to increase x by 1, and T2 wants to increase x by 2
- T1 sees x=10
- T2 sees x=10
- T2 sets x=12
- T1 sets x=11
- Final state is increased by 1 not by 3

Concurrency Problems – Check-use Gap

- T1 checks that the room has space (class.enrollment < class.room.capacity)
- T2 sees that the class in underfull and reallocates it to another room (class.room = new_room)
- T1 inserts a new student (class.enrollment = class.enrollment+1)

Concurrency Problems – Inconsistent Read

- T1 wants to calculate total enrolment number for INFO units; T2 changes student 53 from INFO1103 to INFO1003
- T1 reads INFO1003.enrollment (100)
-     T2 does INFO1003.enrollment++ (now 101)
-     T2 does INFO1103.enrollment-- (now 89)
- T1 reads INFO1103.enrollment (89)
- T1 returns 189 (not correct answer of 190)

General Rules

- If instances of two code segments can interleave,
  - and both access the same item of shared state
  - and at least one of the instances modifies that item
- Then some mechanism is needed to avoid concurrency problems
- Also, same applies for two instances of the same code segment
- Do not try to reason out why some particular case is safe, just use proper protective measures

Concurrency Control

- For realistic applications, concurrency and shared data are both unavoidable

- So need mechanisms to restrict the interleaving and prevent bad things happening
  - Without reducing to a single-threaded computation
  - So allow harmless concurrency, but not harmful concurrency

What is a Transaction?

- A collection of one or more operations on one or more data resources, which reflect a *discrete unit of work*
    - In the real world, this happened (completely) or it didn't happen at all (Atomicity)
    - Can be read-only (e.g. RequestBalance), but typically involves data modifications
- Database technology allows application to define a transaction
- A segment of code that will be executed as a whole
  - System infrastructure should prevent problems from failure and concurrency
- Transaction should be chosen to cover a complete, indivisible business operation

What is a Transaction?

- Examples
  - Transfer money between accounts
  - Purchase a group of products

- Why are transactions and transaction processing interesting?
  - They make it possible to build high performance, reliable and scalable computing systems
  - Transactions aim to maintain integrity despite failures

Transaction – ACID Test

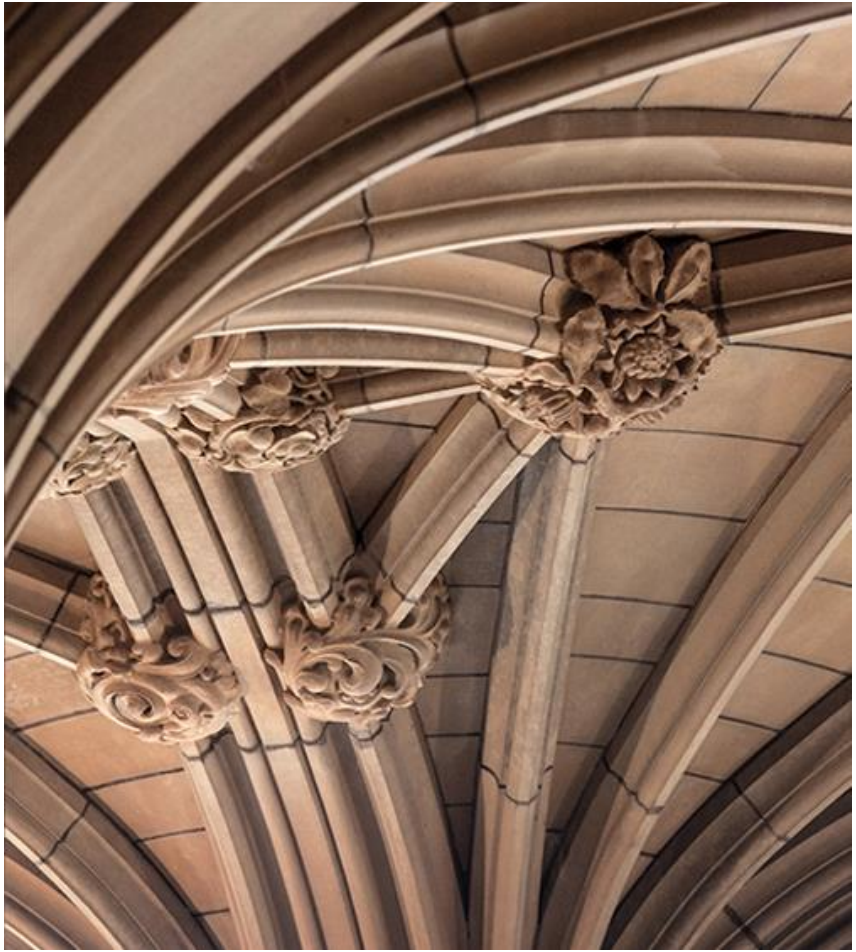
- The transaction support should pass the ACID test
  - Atomic (all, or nothing)
    - Either transaction commits (all changes are made) or else it aborts (no changes are made)
    - Abort may be application-requested rollback, or it may be system initiated
  - Consistent
  - Isolated (no problems from concurrency; details are complicated)
    - DBMS provides choices of “isolation level”
      - Usually good performance under load is only available with lower isolation
      - But lower isolation level does not prevent all interleaving problems
  - Durable (changes made by committed transactions are not lost in failures)

Transaction Systems

- Efficiently and quickly handle high volumes of requests
- Avoid errors from concurrency
- Avoid partial results after failure
- Grow incrementally
- Avoid downtime
- And … <u>never lose data</u>

Concurrency Patterns

Enterprise Application Patterns



THE UNIVERSITY OF SYDNEY

Enterprise Applications – Concurrency Patterns

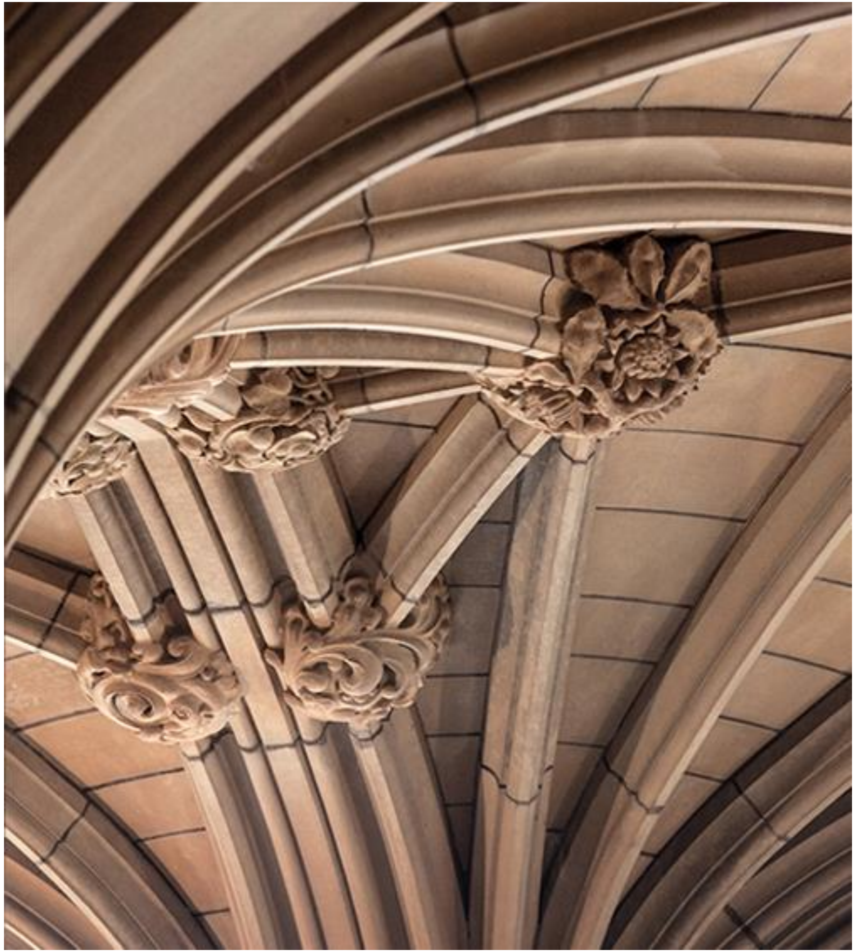
- Design patterns concerned with the multi-threaded programming paradigm including:

  - Thread pool
  - Read write lock
  - Optimistic Lock
  - Pessimistic lock
  - Active Object
  - Others

Thread Pool Pattern

Enterprise/Web Application Patterns

THE UNIVERSITY OF SYDNEY



Thread Pool Pattern

- Design pattern for achieving concurrent execution in an application

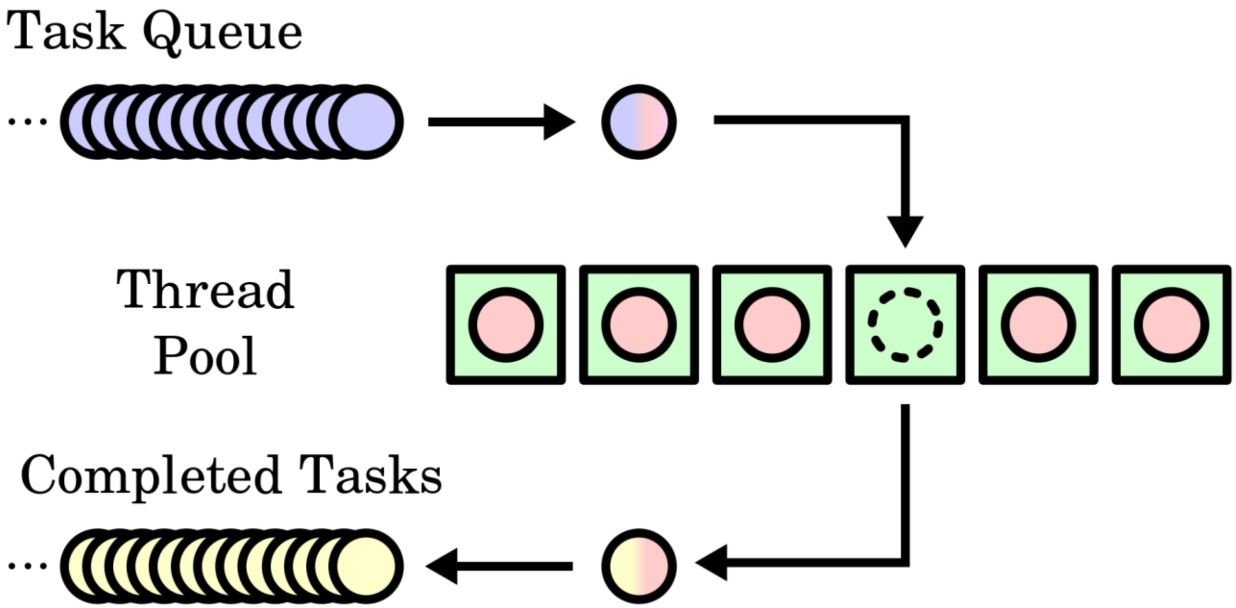
- Also known as replicated works

- Executing tasks sequentially leads to poor performance
  - Some tasks may take longer time than others
  - Some tasks may require communicating with other components (e.g., database)

- Several of tasks need to be executed concurrently to improve performance

Thread Pool Pattern – How it Works

-



Thread Pool Pattern – How it Works

- Multiple *threads* are maintained by a *thread pool*
- Threads wait for *tasks* to be allocated for concurrent execution
- Tasks are scheduled into a *task queue* or *synchronized queue*
- A thread picks a task from the task queue and once the execution of the task is completed it places it in a *completed task queue*

Thread Pool – Performance

- Thread creation and destruction
  - Expensive in terms of times and resources
  - Threads that are initially created can reduce the overhead

- Thread pool size
  - Number of pre-created threads reserved for executing tasks
  - Should be tuned carefully to achieve best performance
    - Right number reduce time and impact on resources
    - Excessive number would waste memory and processing power

Thread Pool – Configuration and Performance

- Number of threads can be adapted dynamically
  - E.g., A web server can add threads if it receives numerous web requests and remove them when number of requests drop

- Performance affected by the algorithm used to create and destroy threads
  - Creating/destroying too many unnecessary threads
  - Creating too many threads consumes resources
  - Destroying many threads require more time later when they are needed again
  - Creating small number of threads gradually may result in long waiting times
  - Destroying small number of threads gradually may waste resources

Thread Pool Pattern – Implementation

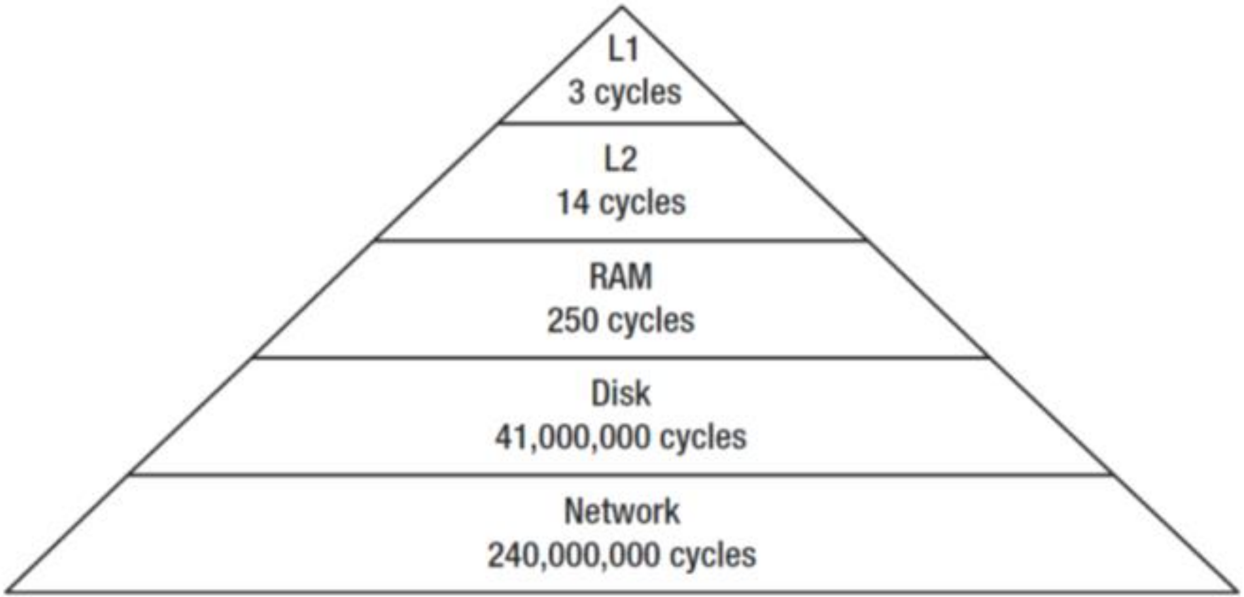
- Typically implemented on a single computer

- Can be implemented to work on server farms
  - Master process (thread pool) distributes tasks to work processes on different computers

Web/Enterprise Applications

- Most enterprise applications deployed in distributed environment
- Accessed online by employees/staff within an enterprise and/or by end clients/customers from the Web
- Developed by web programming languages and deployed on Web servers (e.g., Nginx)
- Web servers uses process-based or thread-based execution of requests/tasks to achieve better performance
- Understanding performance of such systems require understanding I/O

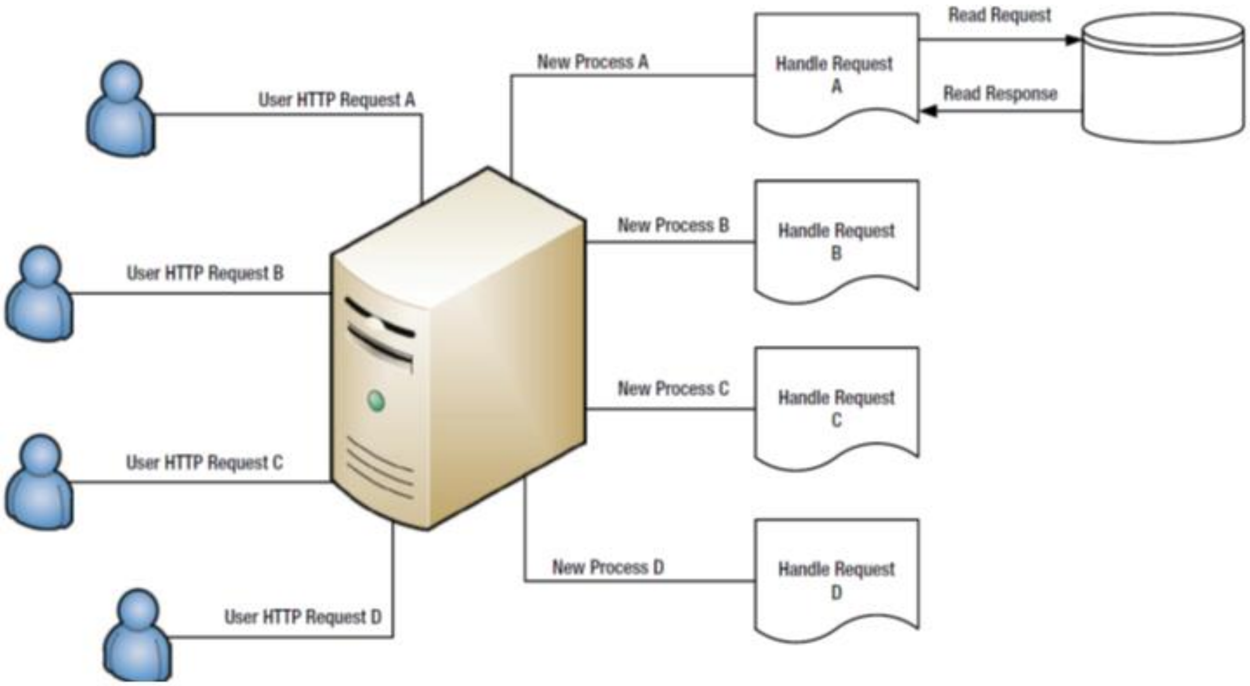
The IO Scaling Problem



Rough estimate of the speed at which we can access data from various sources in terms of CPU cycles

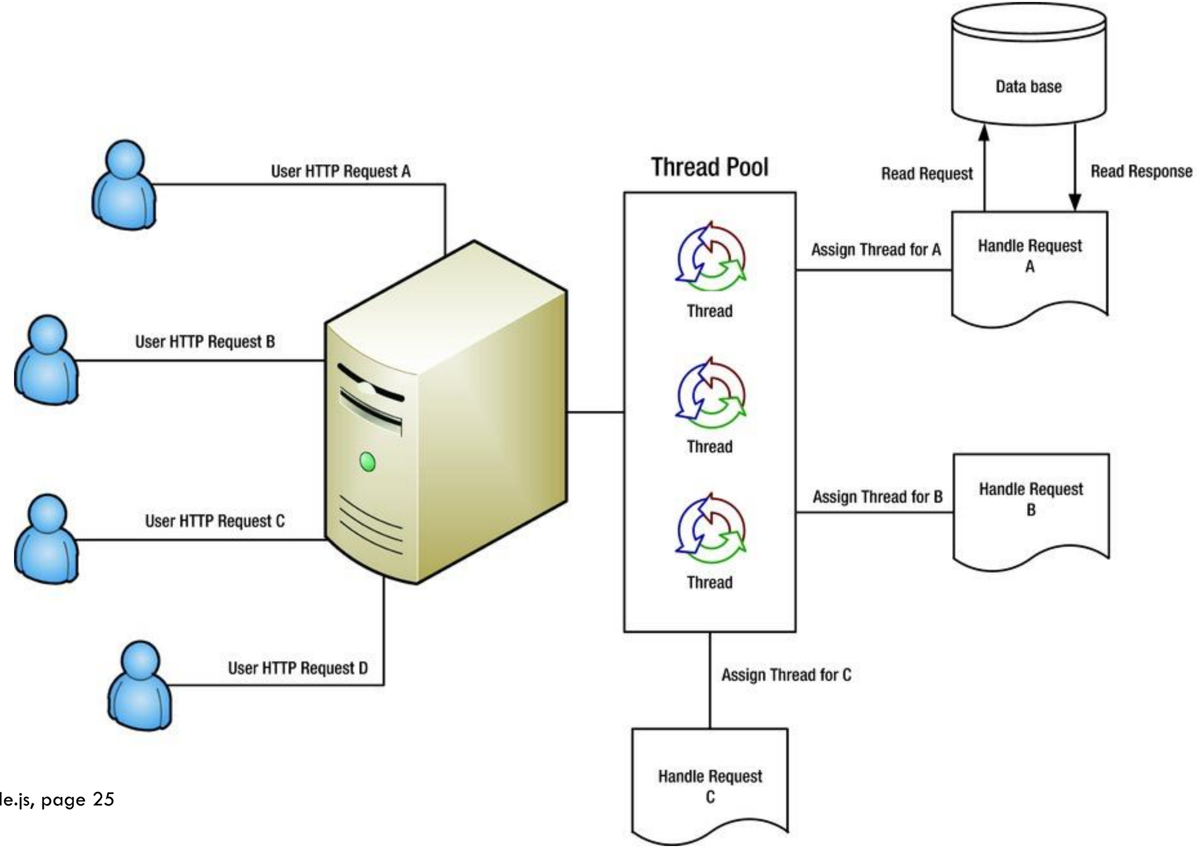
Basarat Ali Syed: Beginning Node.js, page 23

Traditional Server using Processes



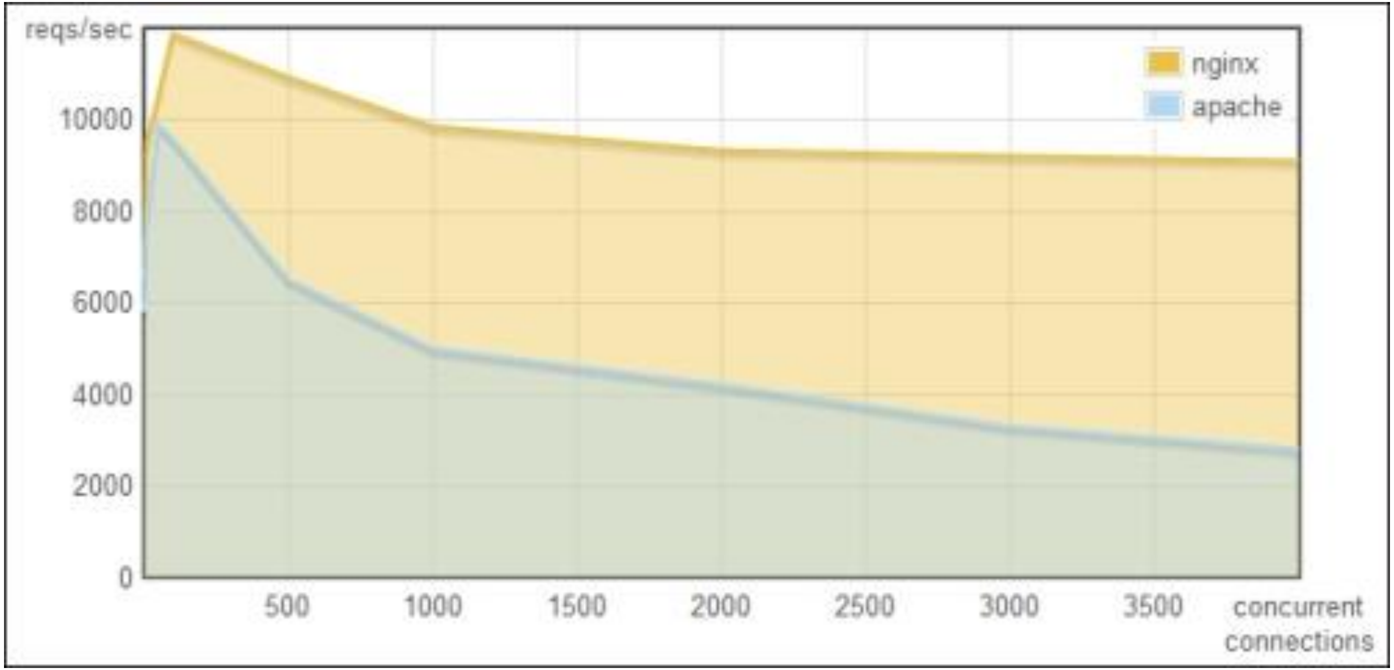
Basarat Ali Syed: Beginning Node.js, page 24

Servers with Multi-Threaded Execution



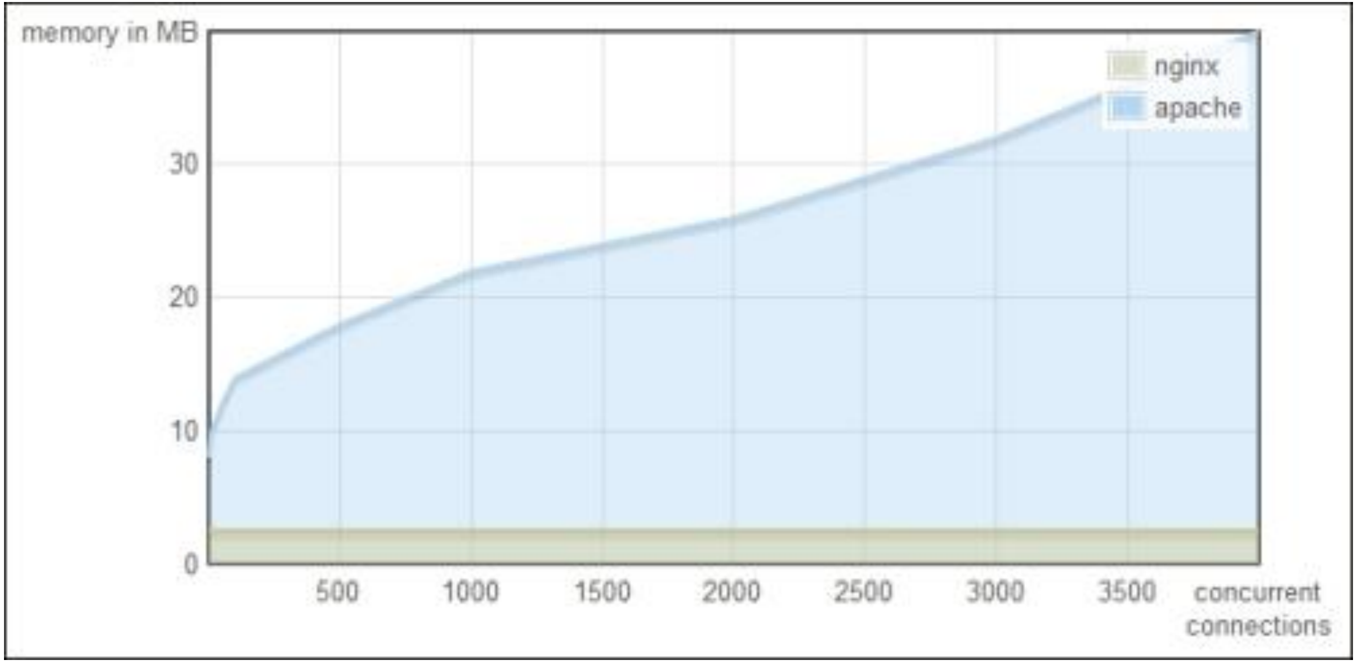
Basarat Ali Syed: Beginning Node.js, page 25

Single vs. Multi-threaded – Performance Comparison



Basarat Ali Syed: Beginning Node.js, page 26

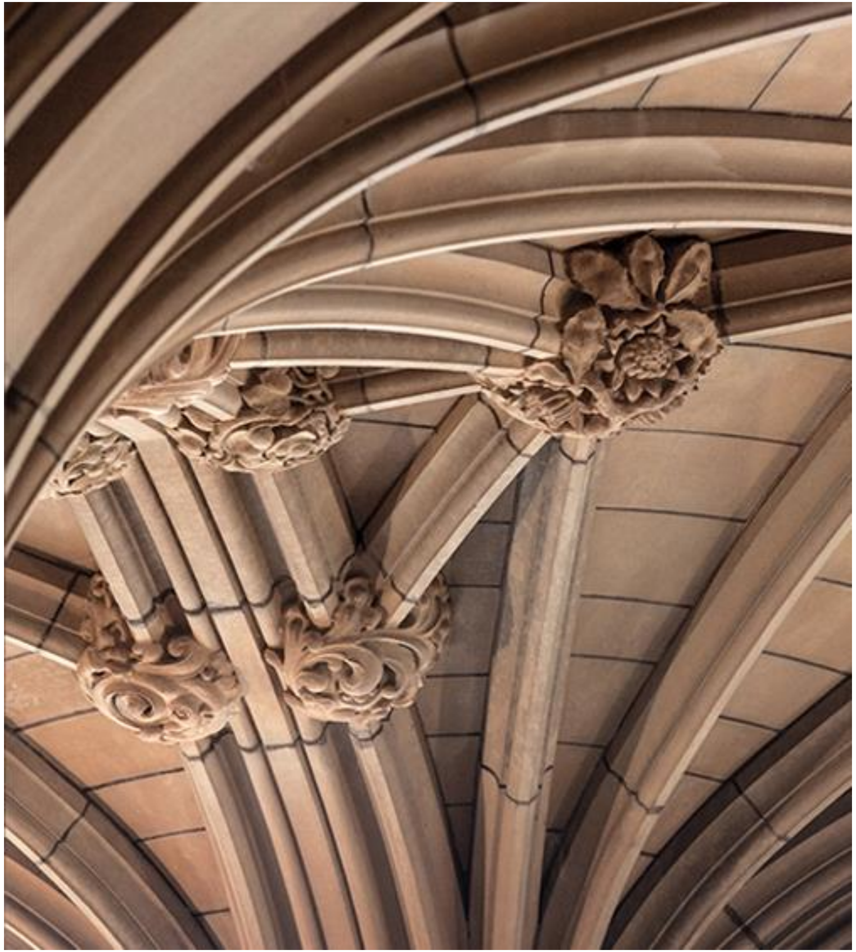
Single vs. Multi-threaded – Performance Comparison



Basarat Ali Syed: Beginning Node.js, page 26

Optimistic (Offline) Lock

Enterprise Application



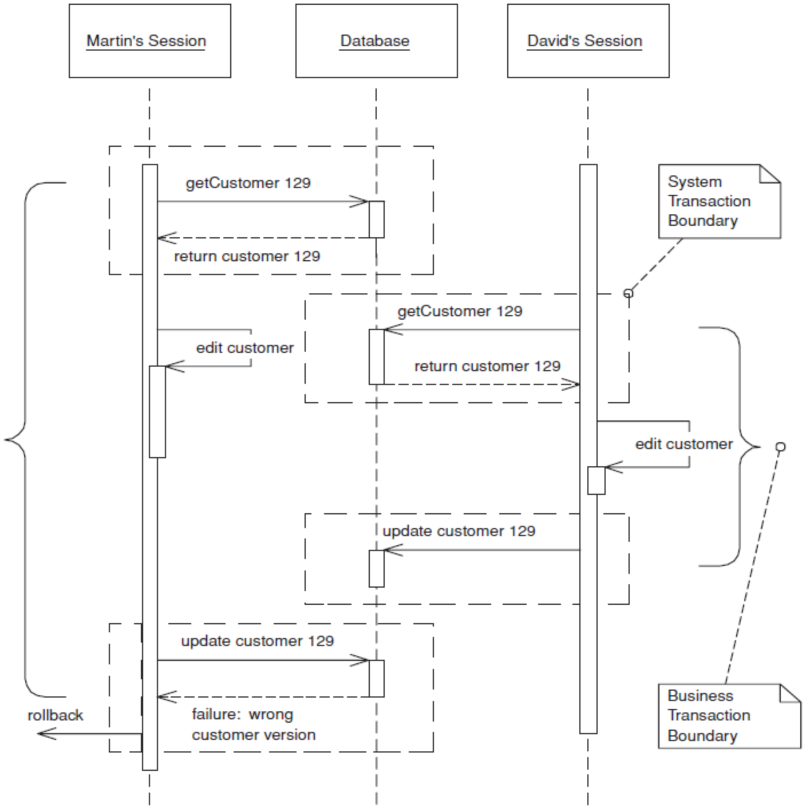
Optimistic (Offline) Lock Pattern

- *"Prevents conflicts between **concurrent** business transactions by detecting a conflict and rolling back the transaction."*

Optimistic (Offline) Lock – When to Use it

- Optimistic concurrency management depends on the chance of conflict
  - Low (unlikely): use optimistic offline lock
  - High (likely): use pessimistic lock

Optimistic (Offline) Lock – How

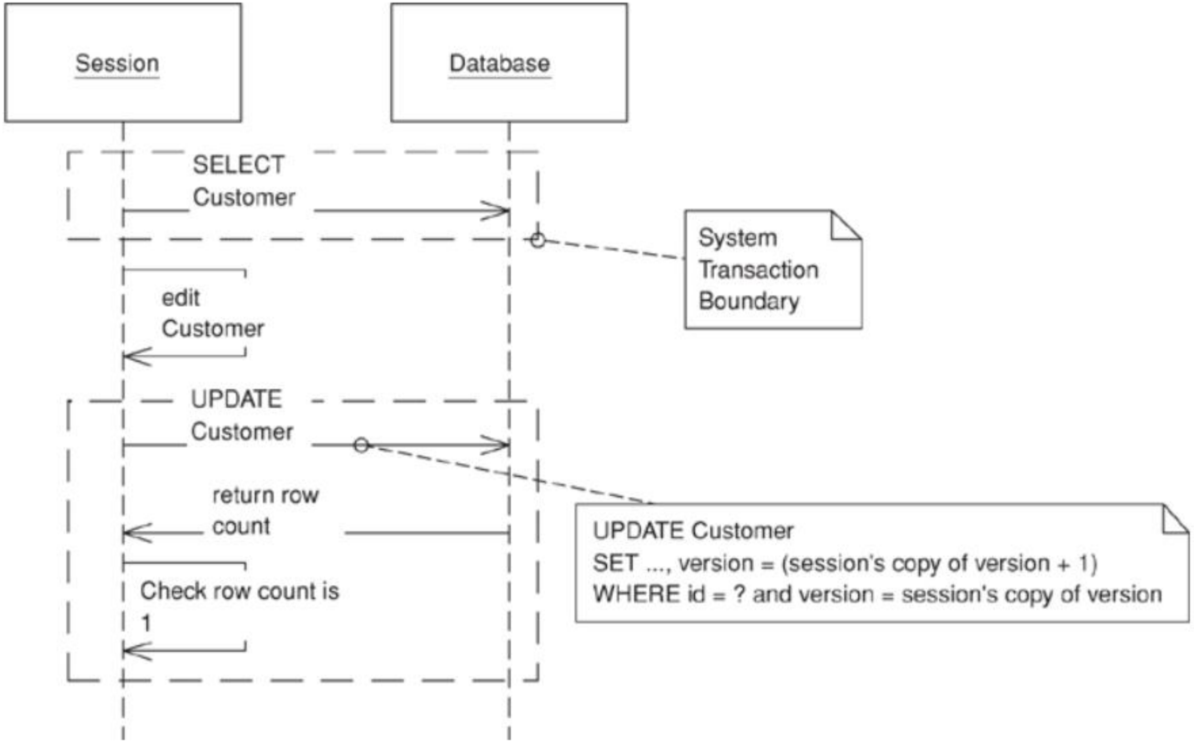


Optimistic Lock – How it Works

- Validate when a session loaded a record, another session hasn't altered it
- Associate a version number (increment) with each record
- When a record is loaded that number is kept by the session along with all other session states
- Compare the session version with the current version in the record data
  - If successful, commit the changes and update the version increment
  - A session with an old version cannot acquire the lock

Update Optimistic Check

– •



Optimistic Lock – Issues

- The version included in Update and Delete statements

- Scenario:

  - A billing system creates charges and calculate appropriate sales tax based on customer’s address
  - A session creates the charge and then locks up the customer’s address to calculate the tax
  - Another session edits the customer’s address (during the charge generation session)
  - Any issues?

Optimistic Lock – Inconsistent Read

- Optimistic lock fails to detect the inconsistent read

- Scenario:

  - The rate generated by the charge generation session might be invalid
  - The conflict will be gone undetected (charge session did not edit the customer address)

  - How to solve this issue?

Optimistic Lock – Inconsistent Read

- Recognize that the correctness of a session depends on the value of a certain data field
  - E.g., charge session correctness depends on the customer's address

- Add the customer address to the change set

- Alternatively, maintain a list of items to be version-checked (bit more code but clearer implementation)

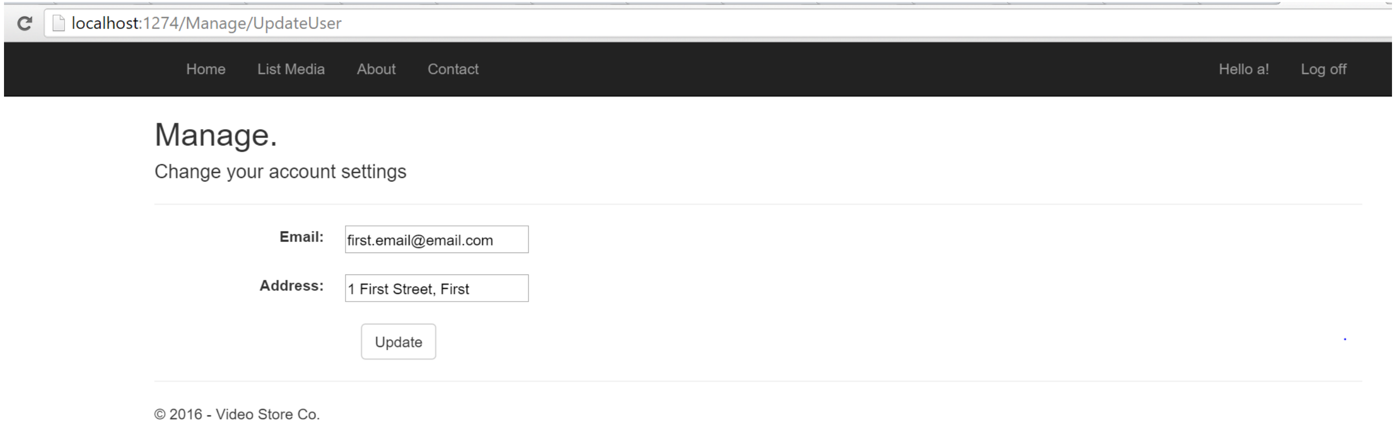
Optimistic Lock – Dynamic Query

- Inconsistent read could be more complex

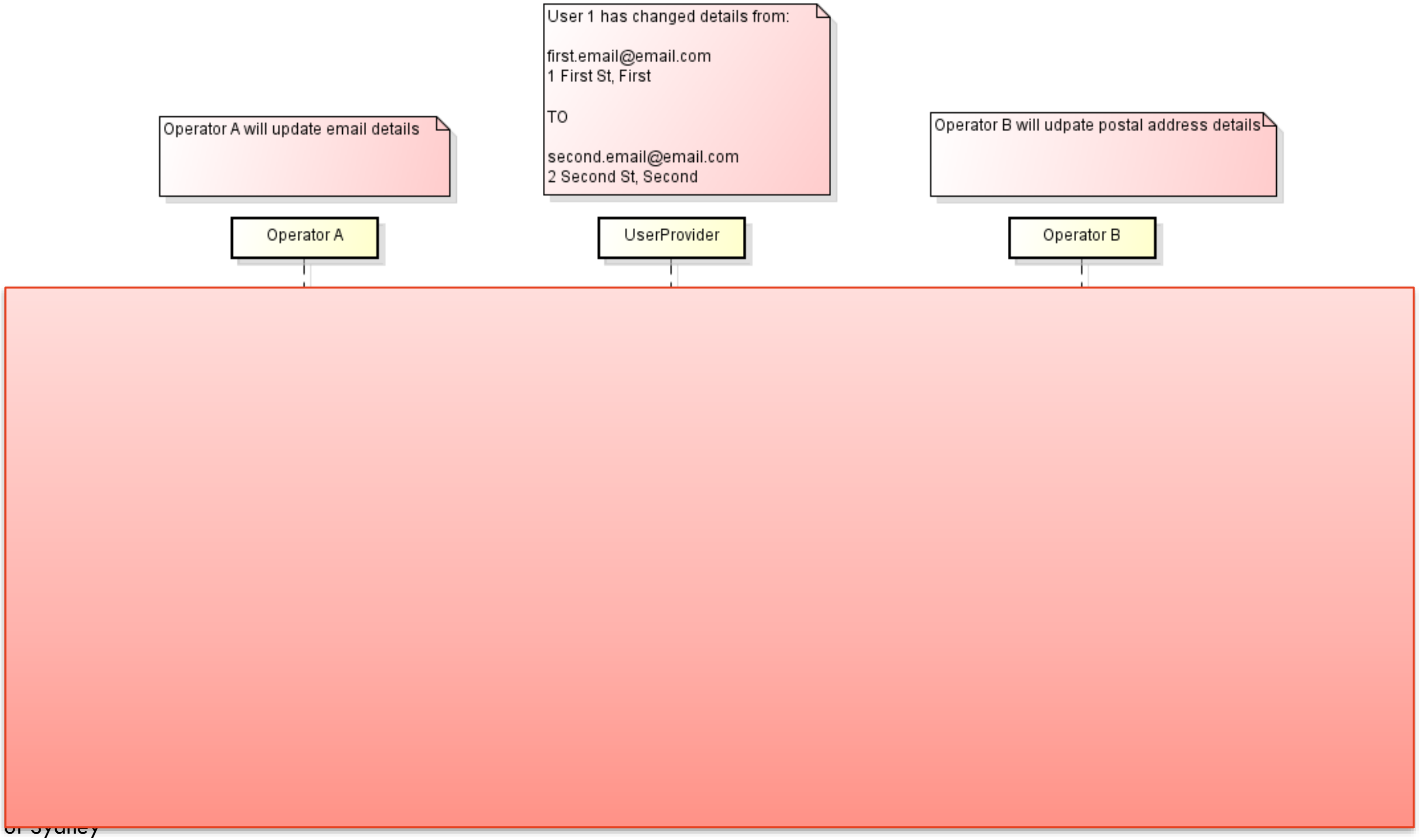
- E.g., a transaction depends on the results of a dynamic query

  - Save the initial results and compare them to the results of the same query at commit time

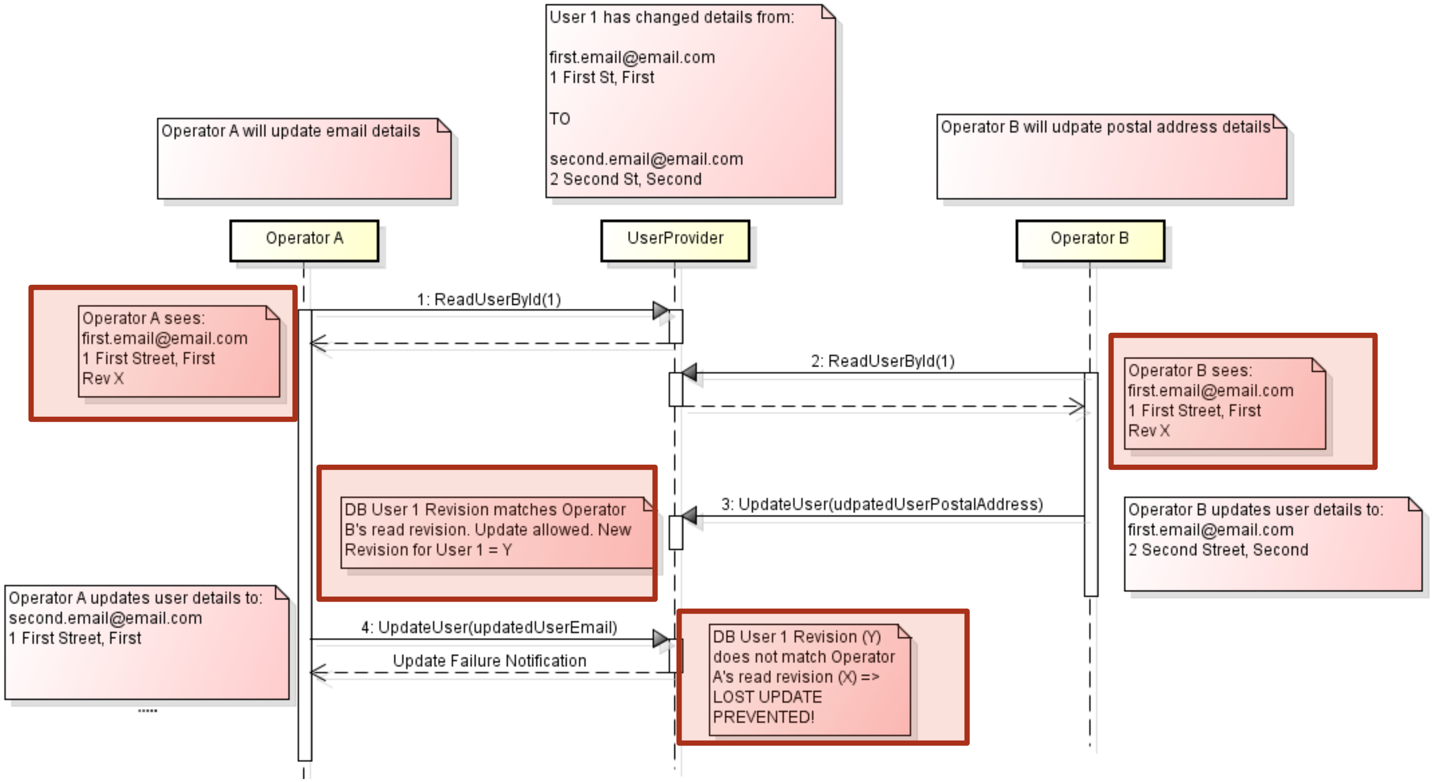
Optimistic Concurrency Control



Optimistic Concurrency Control

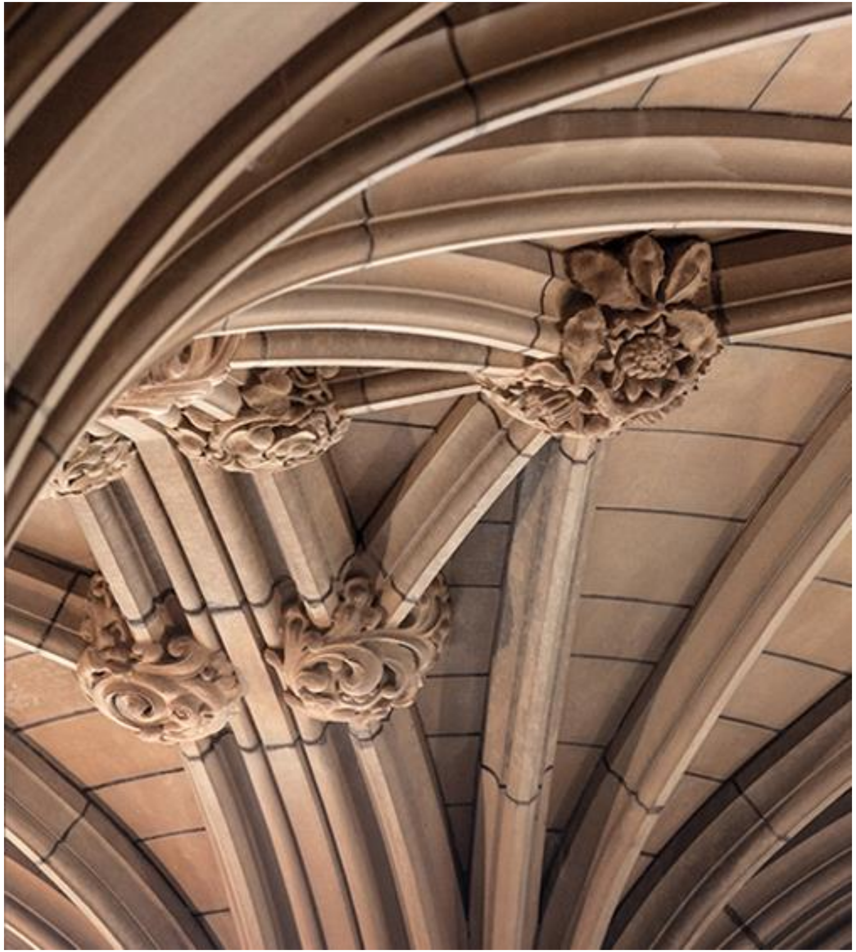


Optimistic Concurrency Control



Pessimistic (Offline) Lock

Enterprise Application Pattern



Concurrency and Long Transactions

- Concurrency involves manipulating data for a business transaction that spans multiple requests
- Transaction management systems cannot help with long transactions, so multiple system transactions should be used
- Why not using optimistic lock pattern?

Concurrency and Long Transactions

- Concurrency involves manipulating data for a business transaction that spans multiple requests

- Transaction management systems cannot help with long transactions, so multiple system transactions should be used

- Why not using optimistic lock pattern?
  - Several users access the same data within a business transaction, only one will commit but the others will fail
  - Other transaction processing time will be lost (conflict is only detected at the end)

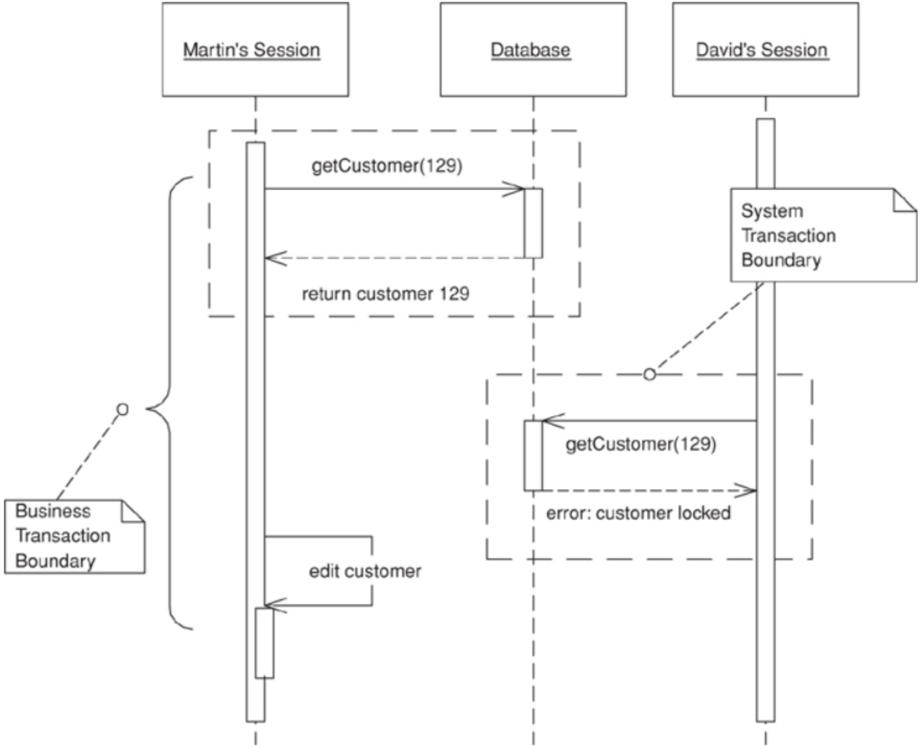
Pessimistic Lock Pattern

*"Prevents conflicts between **concurrent business transactions** by allowing only one business transaction at a time to access data."*

Pessimistic Lock – When to Use it

- When the chance of conflict between two concurrent business transactions is high
- Cost of conflict is too high regardless of its likelihood
- Use when it is really required (it has high performance costs)
- Business transactions spans across multiple system transactions

Pessimistic Lock – How it Works



Pessimistic Lock – How

1. Determine type of locks needed
2. Build a lock manager
3. Define Procedures for a business transaction to use locks

Note: determine which record types to lock if using pessimistic lock to complement optimistic lock

Pessimistic Lock – Types of Lock

- **Exclusive write Lock:** a transaction acquire a lock to edit a session
  - Does not allow 2 transactions to write to the same record concurrently
  - Does not address reading data
  - Allow much more concurrency

- **Exclusive read Lock:** a transaction acquire a lock to load a record
  - Obtain most recent data regardless of the intention to edit data
  - Restrict concurrency

- **Read/Write Lock:** combines both read and write locks
  - Restrict concurrency by Read lock and increased concurrency by Write lock

Pessimistic Lock – Types of Lock

- **Read/Write Lock:** combines the benefit of both lock types:
  - *Read and write locks are mutually exclusive*: a record can't be write-lock if any other transaction owns a read lock on it and vice versa
  - *Concurrent read locks are acceptable*: several sessions can be readers once one has been allowed as read-lock
    - Increase system concurrency
    - Hard to implement

Pessimistic Lock – Which Lock to Choose

- Factors to consider:
  - Maximize system concurrency
  - Satisfy business needs
  - Minimize code complexity
    - Should be understood by data molders and analysts

- Avoid choosing the wrong locking type, locking everything, or locking the wrong records

Pessimistic Lock – Lock Manager

- Define your lock manager to coordinate acquiring/releasing lock requests
  - Need to maintain which transaction (session) and what is being locked
  - Use only one lock table (hash or database table)
    - Use Singleton (GOF) for in-memory lock table
    - Database-based lock manager for clustered application server environment
  - Transaction should interact only with the lock manager but never with a lock object

Pessimistic Lock – Lock Manager

- In-memory lock table
  - Serialize access to the entire lock manager

- Database lock table
  - Interact via transaction-based system
  - Use database serialization
  - Read and Write lock serialization can be achieved by setting a unique constraint on the column storing lockable item's ID
  - Read/write locks are more difficult
    - Reads the lock table and insert into it so you need to avoid inconsistent read

- Use a separate serializable system transaction for lock acquisition to improve performance

Pessimistic Lock – Lock Manager

- **Lock protocol** defines how a transaction can use the lock manager including:

- What to lock?
- When to lock and release?
- How the lock should behave when a lock cannot be acquired?

Lock Manager – When to Lock

- General rule: a transaction should acquire a lock before loading the data

- Order of lock and load is not a matter in some cases:
  - E.g., Frequent read transaction
  - Perform an optimistic check after you acquire the Pessimistic lock on an item to ensure having the latest version of it

Lock Manager – What to Lock

- Lock an ID, or a primary key, of the object/record
- Obtain the lock before you load the object/record (to ensure the object is current)
- Lock release
  - When transaction is complete
  - May release before transaction completion in some cases (understand what is the lock)

Lock Manager – Lock behaviour

- Abort when a transaction cannot acquire a lock
  - Should be acceptable by end users (early warning)

References

- Martin Fowler (With contributions from David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford). 2003. *Patterns of Enterprise Applications Architecture*. Pearson.
- Enterprise-Scale Software Architecture (COMP5348) slides
- Web Application Development (COMP5347) slides
- Basarat Ali Syed 2014, Beginning Node.js. E-book, accessible online from USYD library
- Wikipedia, Concurrency Pattern https://en.wikipedia.org/wiki/Concurrency_pattern
- Wikipedia, Thread Pool, https://en.wikipedia.org/wiki/Thread_pool

W10 Tutorial: Practical Exercises/coding
W10 Lecture: More Design Patterns
Design Pattern Assignment