Issues in the Development of Transactional Web Applications

R. D. Johnson

D. Reimer

IBM Systems Journal

Vol 43, No 2, 2004

Presenter: Barbara Ryder

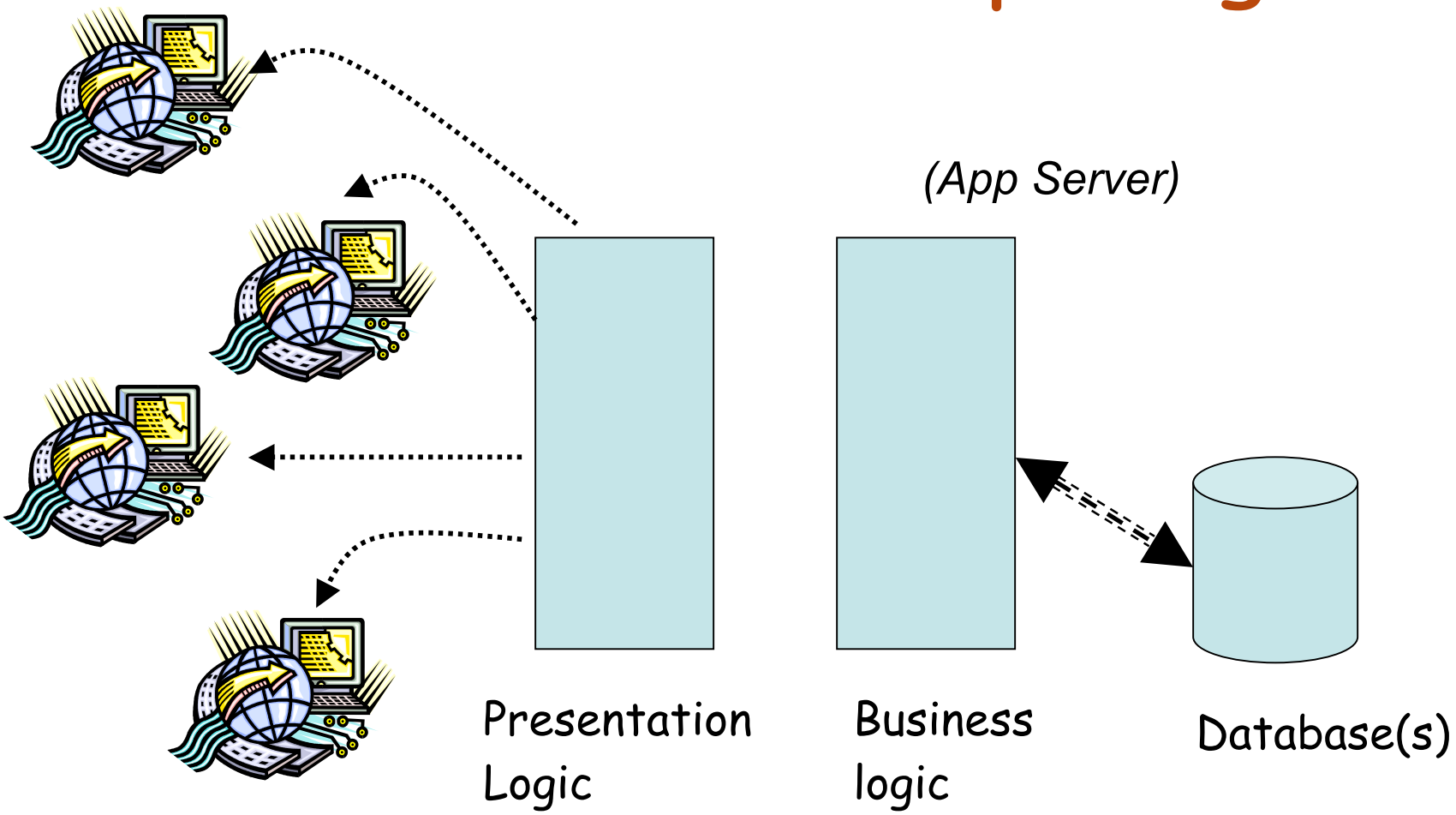
Web Applications

- Transactional processing at their core
  - Resource management & utilization
  - Concurrency & parallelism
  - Failure management
  - Persistent data
  - Configuration management
- Non-transactional issues
  - Memory leaks, Data thru-put, Dealing with overly general frameworks/components

Topology of Web Apps

- Web server between application servers and Internet
  - Static content handled by web server
  - Dynamic content handled by requests to application server (e.g, servlets, JSPs, EJBs, back-end data)
- Choices
  - Vertical scaling: several app servers on one node
  - Horizontal scaling: several app server nodes

Web-based computing



(Fig 1, in Websphere paper)

Transactional Processing

- Different from client-side software
- Unit-of-work properties:
  - Executes with only initial user input
  - Output not done until completion
  - Once started, must complete, unless aborted
  - Concurrency model is complete serializability, but this is not always true
  - Server *runs forever*
  - Uses notion of requests and responses

'Best Practices' Rules for Web SW Developers

- Many client-side or native libraries should not be used
  - Are not re-entrant or are multi-threaded
- Must adhere to 2-phase resource-acquisition discipline (more to come)
- Need to know resource budget (e.g., CPU, memory, I/O, locks) per transaction
- Avoid concurrency
- Legitimate multi-threading use for requests to multiple back-end systems to overlap latencies
- Do not retry failures or errors

Issues

- Resource management & utilization
- Concurrency & parallelism
- Failure management
- Persistent data
- Configuration management

Resource Management

- Pool shared resources to avoid creation/ deletion during execution
  - E.g., threads, network or DB connections
- Consider true resource consumption of code
  - Diff from client code where response time emphasized over resource usage
  - Need to know per transaction CPU secs, #DB operations, read/writes wrt DB, scope/duration of locks etc
    - Important to avoid some transactions 'freezing' entire system

Resource Management

- Use 2-phase resource management
  - Upon acquiring a resource, do not release it until it is no longer needed by the remainder of the transaction
  - Likewise, make sure all sub-operations can find the resource when needed

Issues

- Resource management & utilization
- Concurrency & parallelism
- Failure management
- Persistent data
- Configuration management

Concurrency

- Server code should be re-entrant
- Server code assumes that any two instructions can be separated by a context switch
  - Programmers should not explicitly manage shared state
  - Programmers should avoid writing code for DB connection pools or thread pools

Concurrency

- Issues involving contended objects, locking and hot locks
  - If shared state accessed frequently and locks used and held for long time periods, then there is contention among threads for access to the locked shared state
    - Degrades performance

Issues

- Resource management & utilization
- Concurrency & parallelism
- Failure management
- Persistent data
- Configuration management

Failure Management

- Client apps options in face of failure
  - Automatically retry
  - Communicate with user for instructions
  - Exit, crash, cancel or stop
- Server apps are different
  - Need to determine whether failure is transient or persistent, whether retries may aggravate something else
  - Some operations cannot be retried
  - Programmers should abort their transaction and allow higher-level mechanisms to retry
    - Need to unwind any side effects, unless the transaction is a non-failure

Failure Management

- Exception handling
  - Need to record exception info in log files BEFORE attempting remedial action
  - Never catch one exception and raise another, without logging the original exception or wrapping it in the 2nd exception
  - Never use high-level serv ices (e.g., DBs, publish/subscribe or messaging mechanisms) to log or deal with exceptions
  - Otherwise, problem determination when failure occurs can be impossible

Failure Management

- Minimize number of components and complexity involved
  - Independent failures in multiple components may lead to complex failures
  - Note this goes against design principles of data abstraction which encourages the 'separation of concerns' in implementation
    - E.g., presentation vs business logic

Issues

- Resource management & utilization
- Concurrency & parallelism
- Failure management
- Persistent data
- Configuration management

Persistent Data

- DBs have to contend with persistent and non-persistent (derived) data
  - Cost of transferring data from persistent storage may be prohibitive
  - Cost of converting data to and from persistent form may be prohibitive
  - Data may be 'captured' as persistent when it does not need to be
  - Confusion may result if an object is persisted and then brought back while a live original version still exists

Persistent Data

- High-level frameworks shield programmer from details
  - Container-managed persistence (but may persist more info than necessary)
  - Automatic HTTP persistence may result in saving of arbitrarily large data objects and even illegitimate objects (e.g., DB connection)

Problems with Hiding Details of Persistence

- Programmers unaware of sizes of serialized objects (and its resource needs)
- Programmers use session state as a place to store 'transient' objects, which end up persisted -- bad for performance
- Objects can be unsafe to persist (e.g., objects with native components)

Issues

- Resource management & utilization
- Concurrency & parallelism
- Failure management
- Persistent data
- Configuration management

Configuration

- How to deploy and update a Web app?
  - Depends on configuration data stored at various nodes of distributed system
    - Cached in central DB with local caches per node
      - Live update possible
    - Stored on each node separately
      - May require address space restarts
  - Neither ensures *atomic* update across system or even atomicity for a node

Configuration

- Observations
  - Updates to config parameters at high-load times
  - Updates to program files (e.g., JSP) during high-load periods
  - Configuration changes under operator verification
  - All may be causes of instability and lack of reproducibility of consequent errors

Configuration Updates

- Best practices
  - Do configuration operations for Web apps during nonload periods (I.e., service window)
  - To minimize time taken, use scripts for configuration operations and deploy NEW modified version (do not modify in "real time")
- Web app is complex distributed DB
  - Extremely difficult to impossible
    - Ensure that the running app reflect DB config at all times
    - Allow arbitrary series of updates to be reflected aomically in running app configuration
    - Allow arbitrary rollback of changes, automatically reflected in running app configuration

Configuration Updates

- Hot-swap app
  - Build new version of app and hot-swap in running code (cannot always do with data)
    - Difficult, requires more HW
- Scripted upgrades
  - Use automated scripts to apply and undo upgrades, and use backups
- Both require a service window to minimize effect on users

Non-transactional Issues

- Native memory leaks
  - C/C++ apps which do not free memory
  - Fragmentation due to native allocators
- GC memory leaks
  - For Java, having long-lived object point to memory and never discard
    - E.g. HTTP table w non-expiring session objects
- Recomputation of temporary values

Non-transactional Issues

- Inadequate reuse of complex intermediate values or buffers
  - Reuse of objects within and across transactions (e.g., formatters)
- Overly general and factorized frameworks
  - Too many layers of interface
- Many good client-side tools cannot be used with transactional apps due to
  - Performance cost
  - Difficulty start/stopping transactional servers in production environment
  - Issues of distribution

Research Issues

- Can analysis help?
  - Framework layers limits abilities of static analysis given loose coupling and dynamic binding used
  - Dynamic analysis of app-only seems insufficient
  - Possible combination of techniques for specific problem areas
  - Possible pre-analysis of frameworks layers?
- Performance issues
  - How to observe a transaction-based system discreetly?
  - How to separate out effects of different layers?
  - What areas of CS are involved here? O/S, PLs, DBs?
- Testing approaches
  - How to get realistic moderate-sized systems to test ideas?
  - What aspects of problem would be the best to address?
  - How is all this related to problems in general distributed systems?