Issues in Distributed Architecture

Simon Roberts
Simon.Roberts@earthlink.net

Why Do We Need Architecture?

- Network programming systems usually aren't 100% OO
- Key mistakes can be hard to fix
- Performance/reliability of distributed system can be much harder to achieve than local system

Why Do We Do OO?

- Reuse
- Reliability
- Robustness
- Maintenance
- ...
- Lets us survive some mistakes in early design

How Do We Do OO?

- Design “objects”
  - State and behavior to make a cohesive “thing”
  - Keep related stuff together

Principle #1

- Keep unrelated stuff apart
(Just “Good OO”)

Principle #1 — Example

- Depending on various conditions, one web page proceeds to one of several others
- Decision making logic is not "part of" the presentation of a page, nor is it necessarily unique to one page
- Solution: Separate out the unrelated part (decision making) "Front Controller"

Principle #1 — Example

- Business processes are not inherently part of the business objects they work on (sales process is not part of Order or Invoice)
- Solution: Separate business processes from business data "Session and Entity beans" "Session Facade"

Principle #1 — Example

- The harsh reality of computing is not logically part of any business object
- Solution: Separate persistence handling from the business object itself “EJB container/CMP entity beans” “Data Access Object”

How Does OO Achieve Its Goals?

- Models the world (fairly) realistically
  - OO model helps us find things
- Encapsulation limits consequences of change
  - Simplifies correction
  - Simplifies changing algorithms
- Encapsulation/interface simplifies (re)use
  - Make a call, trust that all necessary consequences occur

Is OO Perfect?

- Well, no...
- OO focuses on correctness, maintainability
  - Not on efficiency
- OO tends to be less efficient than spaghetti
- Calls commonly cause unnecessary work
  - All consequences calculated, not just needed 5%
- Normally this is a good thing (a fair trade)
  - Hardware, is cheaper than Software, is cheaper than fixing a bug

What Happens When a Complex Object Is Modified?

- Can still have trouble finding the wood amongst the trees
- Sometimes break other things

Principle #2

- Keep separate stuff that changes separately (Just "Good OO")

Principle #2 — Example

- Web page presentation
  - Marketing regularly change:
    - Look & feel
    - Colors
    - Banner ads
    - Page structure (location of nav bars, *etc.*)
- Solution: Compose page from elements "Composite View"

Principle #2 — Example

- Processes often change without altering the fundamental business data (Order, Invoice)
- Solution: Separate business processes from business data "Session and Entity beans" "Session Facade" (again)

Principle #2 — Example

- Data storage mechanism (DBMS or schema) is liable to change
- Solution: Separate interfacing with DBMS from data representations "EJB container / CMP entity beans" "Data Access Object"

What Happens When OO Is Distributed?

- The network is slow
  - Round-trip time per call
  - Bandwidth used by argument/return values
- Transaction duration may be extended
- Any state associated with longer-lived operations builds up
- These are consequences of the network, not failings of OO *per se*.

Handling Slow Networks

- If round trips are expensive, make fewer of them — put more work in each request
- This is potentially “bad OO”
  - Design it right first, then “denormalize”

Principle #3

- Minimize round trips

Principle #3 — Example

- UI presentation requires many small data items from middle tier
- Solution 1: Put helper object on presentation tier that offers granular interface, but makes bulk requests over network "Business Delegate"
- Solution 2: Use a struct-like object to pass all data at one time "Value Object"

Handling Low Bandwidth Networks

- Sending large chunks of data that might not be needed is wasteful of bandwidth
- This contradicts principle #3, so might cause trouble
- Determining just what's needed can be hard

Principle #4

- Send appropriate data

Principle #4 — Example

- Particular operations require subsets of data
- Solution 1: Provide specific operations as single methods that collect and format only necessary data "Session Facade"
- Solution 2: Provide mechanism for building just the right data into an object "Value Object Assembler"
- Solution 3: Provide smart, possibly caching, local proxy "HOPP"

Transaction Life

- Long-lived database transactions reduce parallelism, reduce throughput
- If transaction life gets extended by network delays, throughput reduces further

Network Reliability and Transactions

- Networks are not 100% reliable
- If a client starts a transaction, but then loses contact with server, server must time out
- Further throughput reduction results

Principle #5

- Transactions should span few systems and be as brief as possible

Principle #5 — Example

- User wants to perform multi-step operation from web browser
- Operation must be executed in a single transaction
- Solution: Collect multiple steps into a single operation before network request "Business Delegate" or after network request "Session Facade"

Handling State

- State uses storage in memory or disk
- Total space is product of:
  space per client * concurrent client count
- Extending client total process time increases concurrent client count
- Network delays increase client total process time
- State might be held for defunct clients

Why Keep State?

- User interfaces are often easier to use with state
- If you reasonably can, avoid state and perform operations in a single request/response cycle

If State Is Required, Where Should It Live?

- Choices are (typically):
  - Client (*e.g.* Browser)
  - Presentation tier (*e.g.* Web server)
  - Business tier (*e.g.* EJB server)
  - Persistence tier (*e.g.* Database)
- What are the relative benefits/costs?

State on the Browser

- Pro:
  - Self-scaling, more clients, more storage
- Con:
  - Clients disable cookies
  - Possible privacy/security issues
  - Network drowns under state-in-transit

State in the Webserver

- Pro:
  - Increasing server count increases storage
- Con:
  - Load balancing must be session aware
  - Session tracking usually uses cookies, other techniques (URL rewriting) are less reliable

State in the Business Tier

- Pro:
  - Stateful session beans have JVM/EJB-aware "virtual memory" mechanism built in
- Con:
  - EJB servers are less often replicated than webservers

State in the Persistence Tier

- Pro:
  - Almost unlimited storage
  - Resistant to crashes
  - Can survive between connections
- Con:
  - Extra load on DB
  - State must be copied to/from DB

What About Where It's Used?

- Recall: "State must be copied to/from DB"
- But if state is in webserver but required in business process, that too must be copied
- Or *vice-versa* if state used in presentation is stored in business tier
- Networks present significant bottlenecks

Principle #6

- Minimize state, and store necessary state where it will be used

Principle #6 — Example

- Session state controls the appearance and flow of screens
- Solution: Keep session state in the webserver "`HTTPSession`"

Principle #6 — Example

- Session state controls business processes
- Solution: Keep session state in the business tier "Stateful Session Bean"

Principle #6 — Exceptions

- If there is no business tier
  - Not really an exception, store it where it's used
- State must be preserved indefinitely (6 month old shopping cart still full)
  - State in database (perhaps additional to elsewhere)
- Small state used in few transactions by very large numbers of clients
  - State in web browser

Summary

- Principle #1: Keep unrelated stuff apart
- Principle #2: Keep stuff that changes separately, separate
- Principle #3: Minimize round trips
- Principle #4: Send appropriate data
- Principle #5: Transactions should span few systems and be as brief as possible
- Principle #6: Minimize, but store necessary state where it will be used

Imagine a Technology

- Define interfaces for network services
- Freedom to alter implementations
  - Even in highly distributed environment
- Same service may be implemented differently by different providers
- Network protocols hidden behind interfaces
- Modified implementations automatically and transparently installed on “client” system

Imagine a Technology

- Freedom to alter implementations according to changing needs, particularly:
  - Principle #3: Minimize round trips
  - Principle #4: Send appropriate data
  - Principle #6: Minimize, but store necessary state where it will be used
- HOPP pattern, smarts in the network proxy
  - Local cache? Server push on update? Local computation? You decide, when your system is live and you have real data...

A Real Technology

- Jini — RMI, done right, on steroids, and more
- Much more than just mentioned, spontaneous, self healing
- Enabling, not mandating, technology
  - Compatible with everything
- In real use