Concurrency Requirements

Hubert Matthews, Oxyware Ltd
ACCU Conference, April 2005
hubert@oxyware.com

Why this talk?

- Last year's conference - Andrei's talk
  - 1960s: semaphores, mutexes, critical sections
  - 2004: semaphores, mutexes, critical sections
- Why no progress?
  - Is it just hard?
  - Do we lack the right language and tools?
  - Are we thinking about it in the wrong way?
- Emphasis usually from implementors' viewpoint
  - But what do users want and expect?

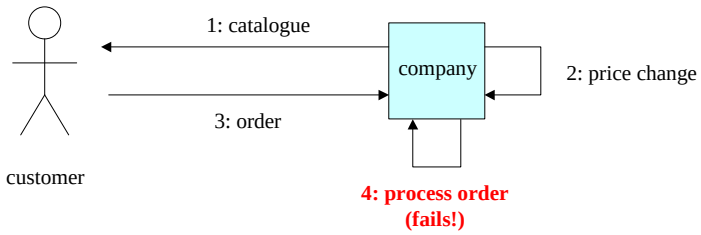
Concurrency is natural

- Human beings seem to be able to handle concurrency
  - You don't often see two skeletons locked in a deadly embrace!
  - Sharing "stuff" - car, kids, etc…
  - Project management, CVS, etc…
- So why the problems with machines?

What is this talk about?

- Requirements and the implications for implementation and design
  - Where design by contract fails
  - Rely/guarantee clauses - temporal logic
  - Ways of discussing concurrency with users
    - Business problems, not technology problems
  - Coherency and synchronisation of data
- Threads, locks, etc as mechanisms to implement concurrency requirements

Concurrency requirement example



- No threads, databases or locks involved
- It's not a technology problem
  - And technology can't solve it, either
- User gets a "surprise" - hidden assumptions

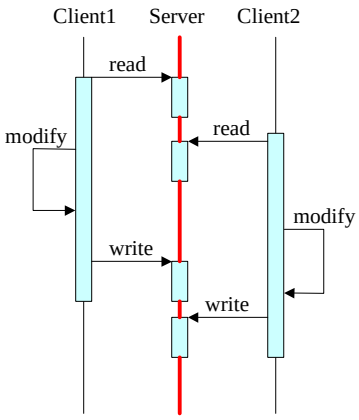
Unexpressed requirements

- Users usually say:
  - "The system must do X"
- Users rarely say:
  - "The system must not do Y"
- Concurrency requirements are in this latter category
  - Rarely made explicit
- C.f. security requirements and unintended consequences

Solution to catalogue problem

- Catalogue has an expiry date
  - The company promises not to change prices for a given period (the lifetime of the catalogue)
- An example of a **guarantee**
- Customer **relies** on this **guarantee**
  - Without it, the transaction *may* fail
  - Hard to test
  - Implicit assumption of how things work
- An example of temporal logic
  - A time-based **contract**

Design by contract doesn't help



- DbC contracts relate to single operations only
  - All server contracts fulfilled
  - Point in time
- Client cannot rely on server
  - no temporal guarantees offered
- Invariant is true only when nothing is happening (red)!
  - During operation all bets are off
  - Re-entrancy not handled
- Classic lost-update problem

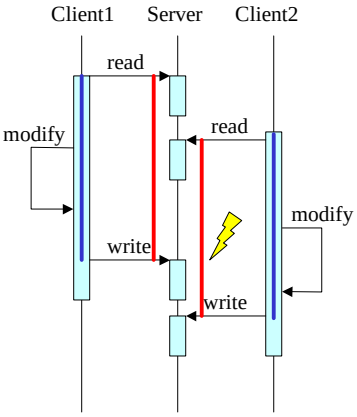
What clients want

- Guarantee between operations (time span)
  - This is the “hidden assumption”
- Can be stated in logic
  - **rely:** catalogue price valid throughout period
  - Testable mechanically but only instantaneously
- Server may offer such **guarantees**
- If not, there is a potential race condition
  - Whenever **guarantee**s do not span **rely** clauses

Catalogue guarantee

- Company must guarantee price does not change during period
- How?
- And what should happen if someone tries?
- Example of an exclusive guarantee
  - Akin to locking or acquire/release
- (Still no technology involved….yet :-)

Guarantee and rely clauses



- Client wants a guarantee that spans time
- This crosses multiple server operations
  - Cannot be implemented at the level of individual server operations
  - Guarantees may need to overlap
- We need some form of state machine
  - Either a stateful server or state must be passed around (token)

Types of guarantee

- Exclusive
  - acquire/release, locks
- Time-based
  - Expiry dates: caching, DHCP leases
- Optimistic locking
  - We'll say sorry afterwards
- None - pot luck
- c.f. exception-safety guarantees

Database isolation as guarantees

- Database isolation levels
  - Read committed, repeatable read, etc
  - Mostly couched in terms of server guarantees
- So, how do you choose?
  - Nothing in the literature (i.e. Google)
  - “It depends on the application”
  - But how?
  - We need to know the client’s rely clause!

Types of rely clauses

- Partitioning of domain may mean no need for server guarantee
  - Separate books of raffle tickets
- Links to transactional integrity
  - Clients execute a “long transaction” by calling sub-transactional services
  - Might need compensating or reversing operations if rely clause can’t be met
  - Or maybe queue - waiting list, back order
  - Again, a business decision

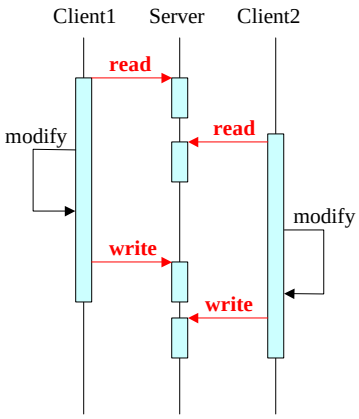
Failure of guarantees

- As a client, when do you find out that a guarantee has failed?
  - At the next operation (sockets)
  - At the end (optimistic locking)
  - Interrupt (cache snooping)
  - Polling

Client view and business view

- Up to now, we've been dealing with the client's view of the world
- How do we deal with conflicting demands from clients?
  - Not everyone can be satisfied all of the time
  - Business policies need to be applied
  - We need to choose who "wins"

Whose write should persist?



- Business choice - who gets to write?
  - First to read
    - First in queue gets first refusal
  - First to write
    - First person to do the deal
  - Last to read
    - Freshest information
  - Last to write
    - Default “do nothing” case
- Not a technology choice
- And what about timeouts?

How to implement these

- First to read (common for transactions)
  - Acquire/release - hold ticket for someone
  - Not a database lock, but an application-level lock
  - Timeout of lock, need to clean up stale locks
  - Can lead to deadlock
- First to write (common for non-txn operations)
  - Optimistic locking, version increment on write
- Last to read (rare)
  - Optimistic locking, version increment on read
- Last to write (all too common)
  - Default “do nothing” case
  - Lost update “problem”

Simple analysis method

1) Work out client's rely clause
2) Work out what guarantees server can offer
3) Compromise!
   - Either make client aware of potential for failure
   - Or make strengthen server guarantees (at cost of concurrent throughput)

Things not covered

- Deadlock, livelock, fairness
- Concurrency as cache coherency problem
  - Idempotent operations
  - Reduction of state synchronisation problems
  - Relationship to volatile and memory model
- Consistency, availability, isolated working
  - Acceptable window of unsynchronisation
- Test-driven design doesn't work for concurrency
- Tradeoff of concurrency reqts and performance

Questions

- Does this make sense?
- Is it important?
- Does this provide language and patterns for thinking about these sorts of problems?
- Does it provide a way of structuring questions for users?
- If so, why has this approach lain dormant for years?

Summary

- Concurrency requirements are usually omitted
  - …until some important user gets a surprise
- The logical framework to handle this has been around for many years (Cliff Jones)
  - It's not even that difficult
  - You just have
    - to think about it, and
    - do it