L1: Complexity, Enforced Modularity, and client/server organization

Frans Kaashoek and Dina Katabi

6.033 Spring 2013
http://web.mit.edu/6.033
http://web.mit.edu/6.033

- Schedule has all assignments
  - Every meeting has preparation/assignment

- On-line registration form to sign up for section and tutorial times
  - We will post sections assignment Thursday evening
What is a system?

• 6.033 is about the design of computer systems
• System = *Interacting set of components with a specified behavior at the interface with its environment*
• Examples: Web, Linux
• Much of 6.033 will operate at design level
  • Relationships of components
  • Internals of components that help structure
Challenge: complexity

- Hard to define; symptoms:
  - Large # of components
  - Large # of connections
  - Irregular
  - No short description
  - Many people required to design/maintain

- Complexity limits what we can build
  - Not the underlying technology
  - Limit is usually designers’ understanding
Problem Types in Complex Systems

- Emergent properties
  - surprises
- Propagation of effects
  - Small change -> big effect
- [ Incommensurate ] scaling
  - Design for small model may not scale
- Problems show up in non-computer systems
Emergent Property Example: Ethernet

- All computers share single cable
- Goal is reliable delivery
- Listen while sending to detect collisions
Will listen-while-send detect collisions?

1 km at 3 Mbit/s

- 1 km at 60% speed of light = 5 microseconds
- A can send 15 bits before bit 1 arrives at B
- A must keep sending for 2 * 5 microseconds
- To detect collision if B sends when bit 1 arrives
- Minimum packet size is 5 * 2 * 3 = 30 bits
3 Mbit/s -> 10 Mbit/s

- Experimental Ethernet design: 3Mbit/s
  - Default header is: 5 bytes = 40 bits
  - No problem with detecting collisions
- First Ethernet standard: 10 Mbit/s
  - Must send for $2 \times 20 \, \mu \text{seconds} = 400 \text{ bits}$
  - But header is 14 bytes
    - Need to pad packets to at least 50 bytes
- Minimum packet size!
A computer system scaling example
Scaling the Internet

- Size routing tables (for shortest paths): $O(n^2)$
  - Hierarchical routing on network numbers
  - Address is 16 bit network # and 16 bit host #
- Limited networks ($2^{16}$)
- Network Address Translators and IPv6
Sources of Complexity

- Many goals/requirements
- Interaction of features
- Performance
Example: more goals, more complexity

- 1975 Unix kernel: 10,500 lines of code
- 2008 Linux 2.6.24 line counts:
  - 85,000 processes
  - 430,000 sound drivers
  - 490,000 network protocols
  - 710,000 file systems
  - 1,000,000 different CPU architectures
  - 4,000,000 drivers
  - 7,800,000 Total
Example: interacting features, more complexity

- Call Forwarding
- Call Number Delivery Blocking
- Automatic Call Back
- Itemized Billing

- A calls B, B is busy
- Once B is done, B calls A
- A’s number on appears on B’s bill
Interacting Features

hidden

- Each feature has a spec.
- An interaction is bad if feature X breaks feature Y.
- ...
- The point is not that these bad interactions can’t be fixed.
- The point is that there are so many interactions that have to be considered: they are a huge source of complexity.
- Perhaps more than \( n^2 \) interactions, e.g. triples.
- Cost of thinking about / fixing interaction gradually grows to dominate s/w costs.
- The point: Complexity is super-linear
Coping with Complexity

- Simplifying design principles
  - E.g., “Avoid excessive generality”

- Modularity
  - Split up system, consider separately
  - Abstraction (e.g., RPC, Transactions)
    - Interfaces/hiding
    - Helps avoid propagation of effects

- Hierarchy (e.g., DNS)
- Layering (e.g., Internet)
A modularity tool: procedure call

- Defines interaction between F and G
- F and G don’t expose internals
- How well does this enforce modularity?
Implementation using stack

- Calling contract between F and G
  - F sets stack pointer for G
  - G doesn’t modify F’s variables
  - G returns
  - G doesn’t wedge environment
    - Use all heap memory, crash, etc.
Is calling contract enforced?

- C, C++: No
  - Callee can overwrite anything
- Java, C#, Haskell, Go: Somewhat
  - Callee may run computer out of resources
- Python: No
  - A type error in callee can fail caller

- Can we do more?
Client/server organization

- Modules interact through messages

Client

- Put args in msg
- Send msg
- Wait for a reply
- Return

Server

- Wait for msg
- Get args from msg
- Compute
- Put results in msg
- Send reply

Internet
C/S enforced modularity

- Protects memory content
- Separates resources
  - Heap, cpu, disk, etc.
- No fate sharing
  - But, client might not get a response
- Forces a narrow spec, but:
  - Bugs can still propagate through messages
  - Programmer must implement spec correctly
Usages of client/server

- Allows computers to share data
  - AFS, Web
- Allows remote access
  - Two banks transferring money
- Allows trusted third party
  - E-bay provides controlled sharing of auction data
Simplifying C/S with remote procedure call

- Stubs make C/S look like an ordinary PC

```python
def main:
    count = inStock(isbn)
    print(count)

def inStock(isbn):
    ....
    return count

def inStockStub:
    msg <- isbn
    send request
    wait for reply
    cnt <- reply
    return cnt

Client

Server

def inStockStub:
    wait for request
    isbn <- request
    cnt = inStock(isbn)
    reply <- cnt
    send reply
```

Stub

Stub
RPC != PC

InStock(isbn) -> count
Ship(isbn, address)
Challenge 1: network looses requests

- Approach: Retry after time out
- Doesn’t work for Ship()
Filter duplicate requests

What if server plus table fail?
Challenge 2: server fails

• “Unknown” outcome for ship(isbn):
  • If server fails before sending reply

• Removing “unknown” outcome requires heavy-duty techniques
  • Check back in April

• Practical solution: RPC != PC
  • Users can check account later
  • Amazon can correct by crediting account
public interface ShipInterface extends Remote {
    public String ship(Integer) throws RemoteException;
}

public static void main (String[] argv) {
    try {
        ShipInterface srvr = (ShipInterface) Naming.lookup("//amazon.com/Ship");
        shipped = srvr.ship("123");
        System.out.println(shipped);
    } catch (Exception e) {
        System.out.println("ShipClient exception: "+ e);
    }
}
Summary so far

- Designing systems is difficult
- Systems fail due to complexity
- New abstractions for system design
  - Enforced modularity through client/server
  - Remote procedure call
  - But, RPC != PC
- Failures will be a central challenge in 6.033
- No algorithm for successful system design
6.033 Approach to system design

- Lectures/book: big ideas and examples
- Hands-ons: play with successful systems
- Recitations: papers describing successful systems
- Design projects: you practice designing and writing
  - Design: choose problem, tradeoffs, structure
  - Writing: explain core ideas concisely
- Exams: focus on reasoning about system design
- Ex-6.033 students: papers and design projects
Example 6.033 systems

- Therac-25
  bad design, at many levels. detailed post-mortem
- UNIX
- MapReduce
- System R
Class plan

• Client/server: Naming
• Operating systems:
  • Enforced modularity within a machine
• Networks:
  • Enforced modularity between machines
• Reliability and transactions:
  • Handling hardware failures
• Security: handling malicious failures