6.033
COMPUTER SYSTEMS ENGINEERING
LECTURE 1

JOHN WANG

1. Complexity: The Essential Challenge

- Large number of components, connections
- Irregular
- Many people required to maintain/build

Complexity limits what we can build (not the underlying technology). The limit is usually the designers' understanding. We need to think of structured ways to understand designs.

1.1. Problems due to Complexity.

- Emergent properties. Surprises. Example: Ethernet. All computers share single cable. Goal is reliable delivery and listen while sending to detect collisions. Listen while send detects. There is a global constraint that emerges in ethernet which requires the header byte to be larger than some minimum threshold. Otherwise, it is impossible to detect collisions.
- Propagation of effects. Small changes lead to big effects,
- Scaling. Design for small model may not scale to larger systems. Example: Internet. When the internet was designed, only a couple hundred computers were a part of the system. Routing tables require $O(n^2)$ time for computing shortest paths. This lead to hierarchical routing on network numbers, only route on the top 16 bits of the 32 bit address. Shrinks the size of routing tables. Scaling further was done with IPv6 where the protocol is completely redesigned.

1.2. Sources of Complexity.

- Many goals/requirements. Example: Unix Kernel. In 1975, there were 10,500 lines of code in the Unix Kernel, but as of 2008 there are 8 million lines of code. There are more lines of code because people want more out of their OS than they wanted in the 70's. New processes and drivers and protocols.
- Interaction of features. Example: Call forwarding. If you are busy in a phone call, forward call to another phone number. This single feature has interactions. What if two phones have call forwarding and an infinite loop ensues. You can add more features: itemized billing and number blocking. If someone wants to get itemized billing but the person who's calling has number blocking, what happens?
- Performance.

1.3. Coping with Complexity.

- Simplifying design principles. Example: avoid excessive generality.
- Modularity. Split up system and consider parts separately. Example: Procedure call or methods. Two procedures F and G don't expose internals. In the stack, there is a calling contract between F and G, where F sets the stack pointer for G and G doesn't modify F's variables. Another example: Client server organization. Client does a particular thing and sends a request to the server. Server is a black-box and returns a response. Modules interact through messages. Internals are never exposed. Enforced modularity by protecting memory content - each has its own separate resources. There is no fate sharing, if the server fails, the client is still alive and vice versa. Forces a narrow specification but bugs can still propogate through messages and implementation could be flawed.
- Abstraction. Interfaces and hiding help avoid propagation of effects.
- Hierarchy and Layering. Example: DNS and the Internet. Take abstractions and stack them on top of each other.

2. Client/Server Modularity

2.1. **Modularity.**

- Client does a particular thing and sends a request to the server. Server is a black-box and returns a response.
- Modules interact through messages. Internals are never exposed.
- Enforced modularity by protecting memory content - each has its own separate resources.
- There is no fate sharing, if the server fails, the client is still alive and vice versa.
- Forces a narrow specification but bugs can still propogate through messages and implementation could be flawed.

2.2. **Uses of Client/Server.**

- Allows computers to share data
- Allows remote access
- Allows trusted third party (E-bay provides controlled sharing of auction data).

2.3. **Remote Procedure Calls.** Use Client/Server idea in procedure calls. The procedure sends a response to the "server" or other procedure. Wait until the procedure sends back a response and continue.

RPCs are not the same as procedure calls. There are some challenges:

- Network can lose requests. Approach: retry after time out, but doesn't always work because you might get two packets. Second approach: send a unique id with each message. The server and client can keep tables of uid.
- This is not sufficient because tables can fail. If server fails, then table disappears. Server has to respond with an "unknown" outcome.

3. Summary

- Designing systems in difficult.
- Systems fail due to complexity, but there's no algorithm for successful design.