Designing and Implementing Internet Protocols

S. Keshav

University of Waterloo

TECS Week, Pune

January 2009

Overview

- Module 1: Introduction
- Module 2: Requirements and challenges
- Module 3: Implementation techniques
- Module 4: Techniques for system design
- Module 5: Testing
- Module 6: Pitfalls

Module 1: Introduction

Outline

- What is the Internet?
- What is an Internet protocol?
- A running example: BuyLocal Service

What is the Internet?

- Set of host interfaces reachable using the Internet Protocol (IP)
- A loose interconnection of networks that
  - carry packets addressed using the Internet Protocol
  - that route packets using a standard Internet protocol (BGP)

A bit more detail…

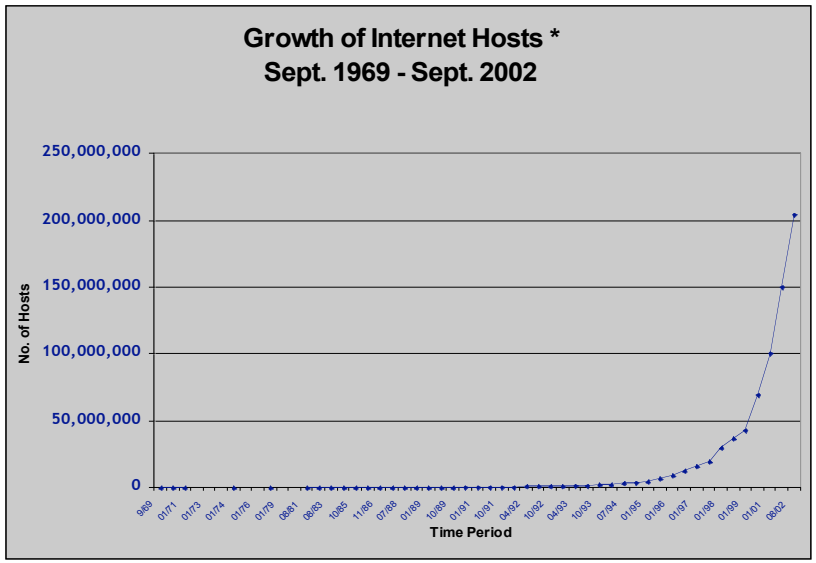
- Loose collection of networks organized into a multilevel hierarchy
  - 10-100 machines connected to a *hub* or a *router*
    - service providers also provide direct dialup access
    - or over a wireless link
  - 10s of routers on a *department backbone*
  - 10s of department backbones connected to *campus backbone*
  - 10s of campus backbones connected to *regional service providers*
  - 100s of regional service providers connected by *national backbone*
  - 10s of national backbones connected by *international trunks*

Example of message routing

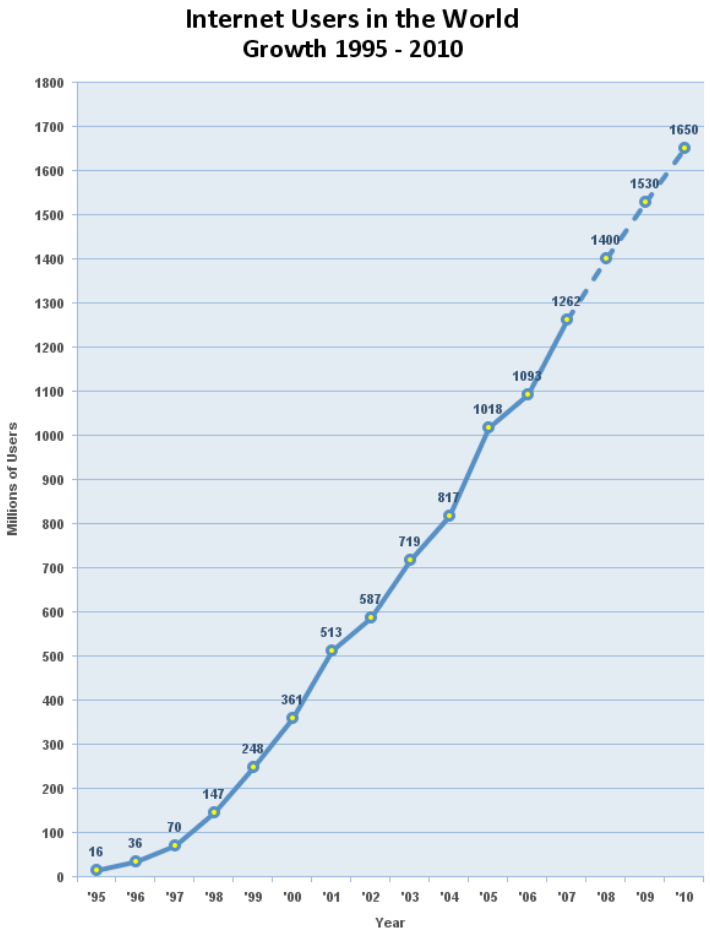
```
 1  dccore-nsfw02-csresearch1net.uwaterloo.ca (129.97.7.1)  0.363 ms  0.308 ms  0.234 ms
 2  dc3558-cs2-csfwnet.uwaterloo.ca (172.19.5.1)  0.475 ms  0.468 ms  0.484 ms
 3  dc-cs2-trk1.uwaterloo.ca (172.19.1.17)  0.478 ms  0.475 ms  0.474 ms
 4  mc-cs2-trk2.uwaterloo.ca (172.19.1.1)  0.491 ms  0.465 ms  0.484 ms
 5  mc-cs1-trk1.uwaterloo.ca (172.19.1.10)  0.604 ms  0.466 ms  0.485 ms
 6  v719-cn-rt-mc.uwaterloo.ca (129.97.1.73)  0.477 ms  0.472 ms  0.512 ms
 7  ext-rt-mc-cn-rt-mc.uwaterloo.ca (129.97.1.6)  0.703 ms  0.464 ms  0.483 ms
 8  gi9-22.mpd01.yyz02.atlas.cogentco.com (38.99.202.213)  6.851 ms  6.966 ms  6.866 ms
 9  te3-2.mpd02.ord01.atlas.cogentco.com (154.54.7.18)  20.954 ms  21.075 ms  20.970 ms
10  vl3499.ccr02.ord03.atlas.cogentco.com (154.54.5.10)  21.210 ms te8-2.ccr02.ord03.atlas.cogentco.com
11  if-9-1.icore1.CT8-Chicago.as6453.net (206.82.141.37)  32.440 ms  21.201 ms  31.589 ms
12 if-2-0-0-18.core1.CT8-Chicago.as6453.net (66.110.14.33)  21.243 ms  21.071 ms
13 if-7-1-0-17.core1.CT8-Chicago.as6453.net (66.110.27.49)  21.091 ms
14  66.110.27.6 (66.110.27.6)  72.039 ms  71.928 ms  72.059 ms
    MPLS Label=970 CoS=5 TTL=1 S=0
15  if-9-0-0.mcore3.PDI-PaloAlto.as6453.net (216.6.29.25)  105.024 ms  110.145 ms  150.389 ms
    MPLS Label=2240 CoS=5 TTL=1 S=0
16  if-4-0-0.mse1.SV1-SantaClara.as6453.net (216.6.29.2)  224.358 ms  202.963 ms  203.384 ms
17  ix-2-11.mse1.SV1-SantaClara.as6453.net (209.58.93.30)  71.903 ms  72.040 ms  72.310 ms
18  59.163.55.253.static.vsnl.net.in (59.163.55.253)  350.914 ms  350.773 ms  351.017 ms
19  203.200.87.72 (203.200.87.72)  350.618 ms  350.652 ms  350.879 ms
20      delhi-203.197.224-18.vsnl.net.in (203.197.224.18)  378.103 ms  466.955 ms  410.726 ms
21      …
```

Internet growth trends

- Number of hosts on the Internet doubled in size every year from 1969 to 1998
- Linear growth subsequently (~120 million/year)
- Roughly 1.2 billion hosts in 2008



Source: ISOC:www.isoc.org/internet/history/2002_0918_Internet_History_and_Growth.ppt



Source: www.internetworldstats.com - January, 2008

Growth continues in services

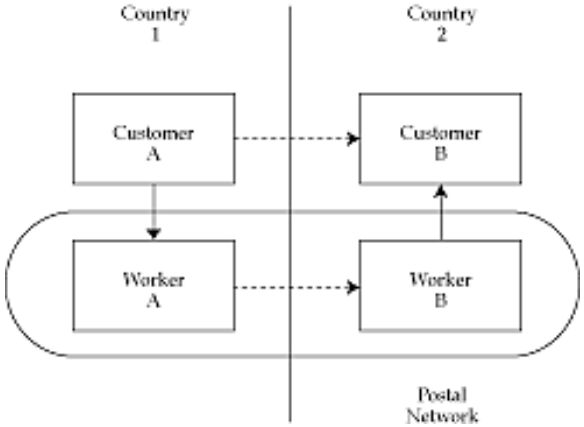
- Skype
- Facebook
- Search (Google, Yahoo, Microsoft)
- Internet email
- BitTorrent
- …

- Each have more than 100 million users daily!
- How should we design and implement the underlying protocols?

Protocols

- A *protocol* is a set of rules and formats that govern the communication between communicating peers
  - set of valid message formats (syntax)
  - meaning of each message (semantics)
  - actions to be carried out on receipt of all possible messages and message orderings
- Necessary for any function that requires cooperation between peers

Peer entities



- Customer A and B are *peers*
- Postal worker A and B are *peers*

Example: careful file transfer

- Exchange a file over a network that corrupts packets
  - but doesn't lose or reorder them
- A simple protocol
  - send file as a series of packets
  - send a *checksum*
  - receiver sends OK or not-OK message
  - sender waits for OK message
  - if no response, resends entire file
- Problems
  - single bit corruption requires retransmission of entire file
  - what if link goes down?
  - what if peer OS fails?
  - what if not-OK message itself is corrupted?

Another way to view a protocol

- As providing a *service*
- The example protocol provides *careful file transfer service*
- Peer entities use a protocol to provide a service to a higher-level peer entity
  - for example, postal workers use a protocol to present customers with the abstraction of an *unreliable letter transfer* service

What is an Internet protocol?

- Any protocol layered on IP
- Endpoints can be anywhere on the Internet
  - many non-trivial consequences

Example protocol suite for a service

- We'll design the 'BuyLocal' service
  - search for local supplier of a good or service
  - distributed searchable directory

Module 2: Requirements and challenges

Requirements (1)

- Universal access
  - anyone, anywhere, on any device
  - should scale to millions of users
    - geographically distributed
    - multi-lingual
    - multi-currency
    - can potentially have flash crowds

Requirements (2)

- Universal access
- Extensible
  - should allow new services to be added
    - advertising
    - social networks
    - …

Requirements (3)

- Universal access
- Extensible
- Robust
  - tolerant of failures in any component
  - results should be repeatable

Requirements (4)

- Universal access
- Extensible
- Robust
- Secure
  - privacy
  - integrity
  - rights management

Requirements (5)

- Universal access
- Extensible
- Robust
- Secure
- Accountable
  - should be able to measure usage
  - potentially allow billing

Requirements (6)

- Universal access
- Extensible
- Robust
- Secure
- Accountable
- Legacy-compatible
  - by far the most onerous requirement

Requirements summary

- Universal access
- Extensible
- Robust
- Secure
- Accountable
- Legacy-compatible

These are mutually incompatible!

Universal access (1)

- Centralization is impossible!
  - why?
- Distributed
  - multiple federated administrative entities (e.g. IP)
    - varying implementations on heterogeneous platforms
    - potentially non-cooperative
      - need incentive-compatibility
      - best possible outcome when each entity ‘does the right thing’
    - need inter-operability
      - openness

Universal access (2)

- Distributed
  - multiple federated administrative entities (e.g. IP)
  - or, single administrative control (e.g. Amazon, Google, eBay)
    - allows tight control
    - proprietary interfaces
      - but how to grow a developer community?

Universal access (3)

- Distributed
  - in both cases, have to deal with lack of global state
  - root cause of nearly all problems in distributed systems

Universal access (3)

- Distributed
- High performance
  - there is a standard set of tools and techniques
    - clusters
    - pseudo-processes
    - …

Universal access (4)

- Distributed
- High performance
- Multiple platforms
  - desktops, laptops, mobile phones, embedded devices, …
  - Windows, Linux, MacOS, …
  - different browsers
  - different languages
  - different currencies
  - …

Universal access (5)

- Distributed
- High performance
- Multiple platforms
- Deal with underlying problems
  - firewalls
  - gateways
  - VPNs
  - …

Extensible

- Future requirements are unknown
- Need to deal with incompatibilities with existing requirements and implementation
- Difficult to detect and deal with side effects

Robust

- Many failure modes
  - server failure
  - device failure
  - storage failure
  - link failure
  - bad implementations
  - or a combination!
- Improving robustness usually degrades performance

Secure (1)

- Assuring integrity
  - need to prevent or discover tampering
  - a variety of cryptographic techniques
  - problems
    - user incomprehension
    - reduced performance
    - key distribution

Secure (2)

- Integrity
- Assuring privacy
  - need to prevent eavesdropping
  - many known cryptographic techniques
  - same problems as with integrity

Accountable (1)

- Every action should be potentially attributable to a real-world entity
- Reduces to two sub-problems
  - identity
  - data management

Accountable (2)

- Identity
  - entities have (and need to have) multiple identities
    - anonymous, pseudonymous, and verinymous identities
  - should they be linked?
    - many open societal problems
    - no consensus

Legacy compatible

- Depends on what to be compatible with…

Module 3: Implementation techniques

Overview

- A service corresponds to a set of protocols that implemented in the wide area, in a cluster, and within a server
- Implementing protocols across the wide area
  - structured and unstructured state dissemination
  - gossip, centralization, P2P, and hierarchy
- Protocol implementation in a cluster
  - three-tier architecture
- Intra-server architecture
  - location
  - interfaces

Implementing protocols in the wide area

- Three challenges
  - deciding where to place functionality
  - bypassing firewalls
  - state coordination

Placing functionality

- End-to-end argument
  - highest protocol layer needs assurance semantics that only it can provide
  - so, lower layers need not try too hard to provide assurance
- Example: careful file transfer
  - application needs to know every block reached
  - cannot trust the network because crashes could happen at the peer OS
  - retransmission is needed at the application layer, so no need to try too hard in the network
- Fast and dumb pipes with intelligence pushed to ‘edges’
- Implications for BuyLocal service?

Dealing with firewalls

- Layer over HTTP
  - allows universal egress
- Use a public rendezvous server
  - each endpoint sets up a connection
  - rendezvous server does application-level routing
    - I3, STUN, HIP, Mobile IP, etc.
- Implications for BuyLocal service?

State coordination

- A node needs to know about state of some other node
  - e.g., what requests it has served, what data it has, its load, …
  - what is needed for BuyLocal service?
- Accomplished by communication
- Knowledge deteriorates due to event occurrences
- Need periodic updates
- Two choices
  - structured
  - unstructured

Impossibility result

- Perfect coordination is impossible if there can be message or node failures
  - we have to settle for approximate coordination and failure-safety

Structured coordination (1)

- Centralized solution
  - poor scaling and fault tolerance
  - outcomes are deterministic
  - ‘virtual centralization’ works well
    - using clustering

Structured coordination (2)

- Centralized solution
- Tree-based solution
  - nodes form a tree overlay on IP
    - e.g., DNS
  - better scaling
  - fault tolerance possible with redundant links
  - outcomes are deterministic
  - allows delegation
  - most widely used solution in practice

Unstructured coordination (1)

- Each server (node) knows only about its neighbours
- General algorithm
  - global computation is divided into a sequence of local computations
    - local computation fuses local state to in-progress state
  - node does local computation then sends message to a neighbour
  - computation aggregates local and in-progress state
- Robust to node and link failures
  - but outcomes are probabilistic
  - and need to prevent double counting
- Example
  - count number of nodes
  - count sum of node values

Unstructured coordination (2)

- Flooding
  - either *pull* updates from all neighbors or *push* updates to them
  - source-specific sequence numbers eliminate duplicates
  - examples: OSPF, BGP

Unstructured coordination (3)

- Flooding
- Random walks
  - node sends its state in a message to a randomly selected neighbor
  - neighbor updates its local state, adds its local value to the message's state, and forwards to a random neighbour
  - parallelizable
  - each walk does a distributed computation over a random sample of node states

Unstructured coordination (4)

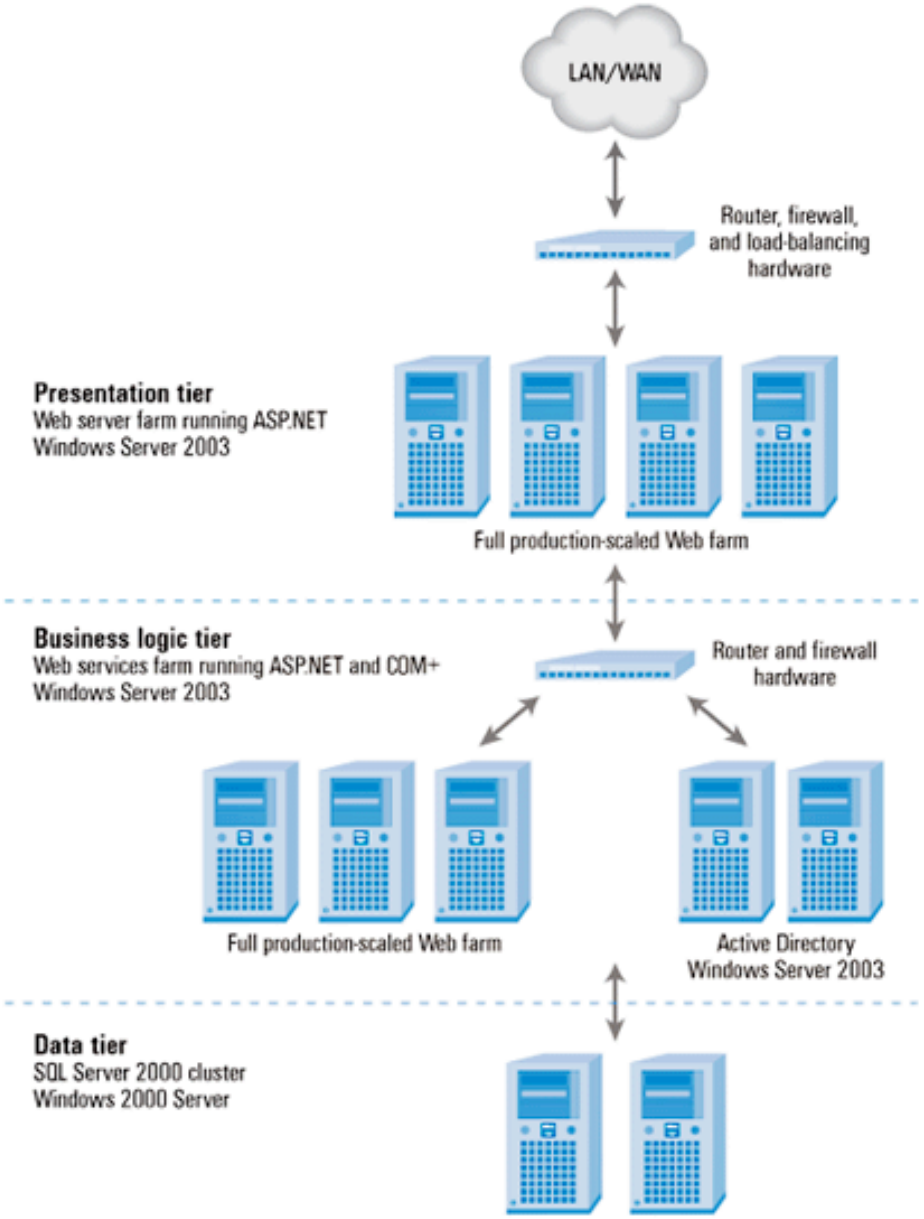
- Flooding
- Random walks
- Gossip
  - computation proceeds in rounds
  - in each round, each node either pushes data to or pulls data from a random neighbor
    - typically network is a clique
  - after *log N* rounds, with high probability, all nodes know everything
  - push better in early stages, pull in late stages
  - termination is an open problem

Aggregation

- Need to prevent double counting
- Three approaches
  - carry node IDs
    - does not scale
  - use order and duplicate insensitive *sketches*
    - can have high errors
  - use *push synopses*
    - each node has an initial weight
    - when sharing a value, share part of the weight
    - using mass conservation, can show that double counting is avoided
    - elegant, but poor fault tolerance

(2) Cluster-based computing

- Set of geographically close nodes on a high-speed interconnect form a cluster
- Elements
  - Redundant servers
  - Network interconnect
  - Shared storage
  - Load balancers



Source: Dell Computers

Key features

- Fault tolerant
- Highly scaleable
- Great diversity of implementation environments
  - J2EE, ASP, scripting
- Incrementally expandable
- Industry-standard components
- Multiple vendors

(3) Protocol implementation within a server

- Two main topics
  - Layering and protocol stacks
  - Implementing a protocol stack

Protocol layering

- A network that provides many services needs many protocols
- Turns out that some services are independent
- But others depend on each other
- Protocol A may use protocol B as a *step* in its execution
  - for example, packet transfer is one step in the execution of the example reliable file transfer protocol
- This form of dependency is called *layering*
  - reliable file transfer is *layered* above packet transfer protocol
  - like a subroutine

Protocol stack

- A set of protocol layers
- Each layer uses the layer below and provides a service to the layer above
- Key idea
  - once we define a service provided by a layer, we need know nothing more about the details of *how* the layer actually implements the service
  - information hiding
  - decouples changes

The importance of being layered

- Breaks up a complex problem into smaller manageable pieces
  - can compose simple service to provide complex ones
  - for example, WWW (HTTP) is Java layered over TCP over IP (and uses DNS, ARP, DHCP, RIP, OSPF, BGP, PPP, ICMP)
- Abstraction of implementation details
  - separation of implementation and specification
  - can change implementation as long as service interface is maintained
- Can reuse functionality
  - upper layers can share lower layer functionality
  - example: WinSock on Microsoft Windows

Problems with layering

- Layering hides information
  - if it didn't then changes to one layer could require changes everywhere
    - *layering violation*
- But sometimes hidden information can be used to improve performance
  - for example, flow control protocol may think packet loss is always because of network congestion
  - if it is, instead, due to a lossy link, the flow control breaks
  - this is because we hid information about reason of packet loss from flow control protocol

Layering

- There is a tension between information-hiding (abstraction) and achieving good performance
- Art of protocol design is to leak enough information to allow good performance
  - but not so much that small changes in one layer need changes to other layers
- Always allow bypass

BuyLocal protocol stack

- What protocols are needed?
- How should they be layered?

Implementing a protocol stack

- Depends on *structure* and *environment*
- Structure
  - *partitioning* of functionality between user and kernel
  - separation of layer processing (*interface*)
- Environment
  - data copy cost
  - interrupt overhead
  - context switch time
  - latency in accessing memory
  - cache effects

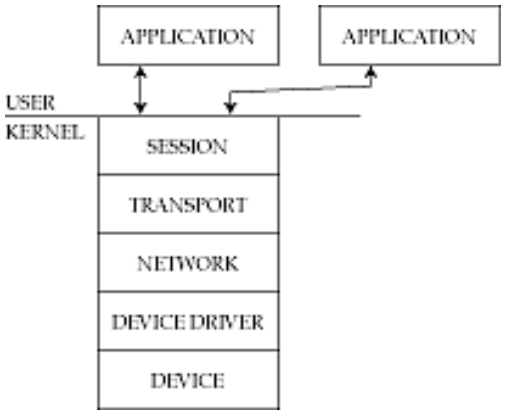
Structure: partitioning strategies

- How much to put in user space, and how much in kernel space?
  - tradeoff between
    - software engineering
    - customizability
    - security
    - performance
- Three choices
  - monolithic in kernel space
  - monolithic in user space
  - per-process in user space

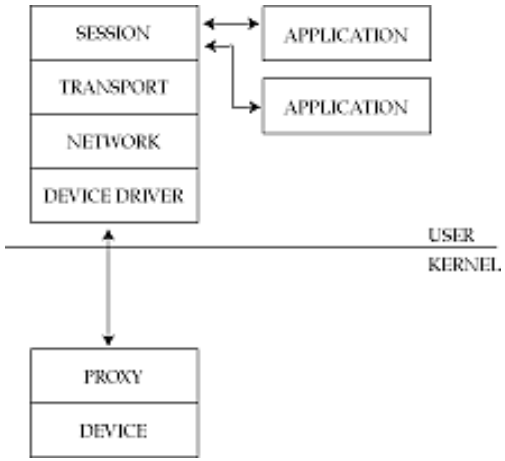
Structure: interface strategies

- Again, three well-known alternatives
  - single-context
  - tasks
  - upcalls

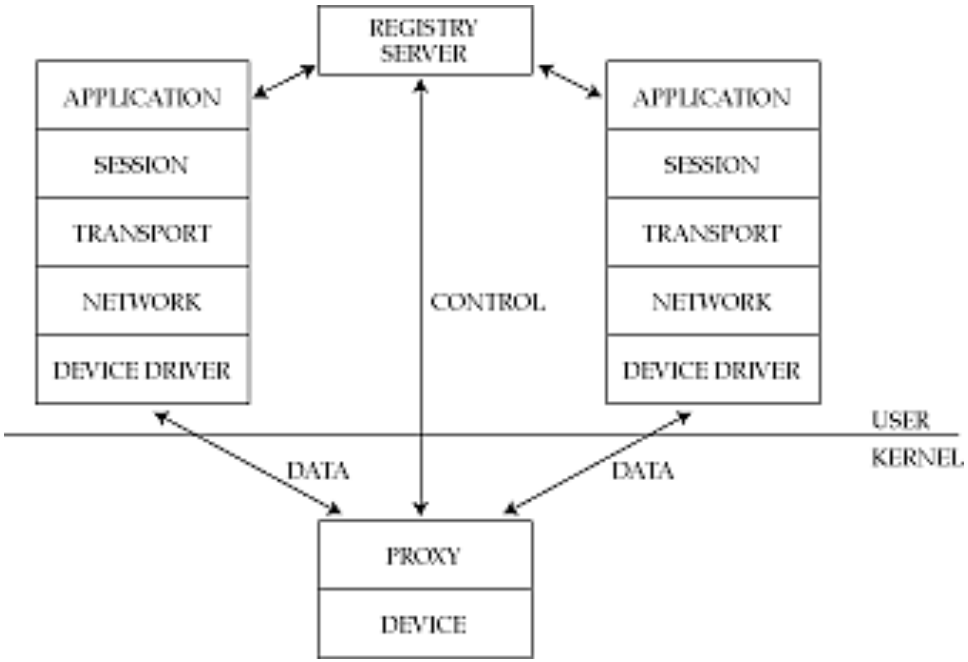
Monolithic in kernel



Monolithic in user space



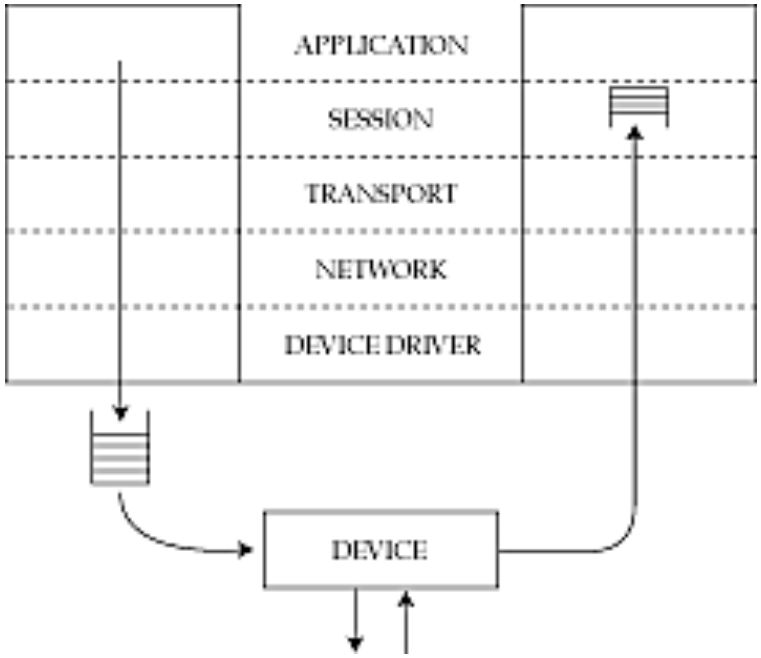
Per-process in user space



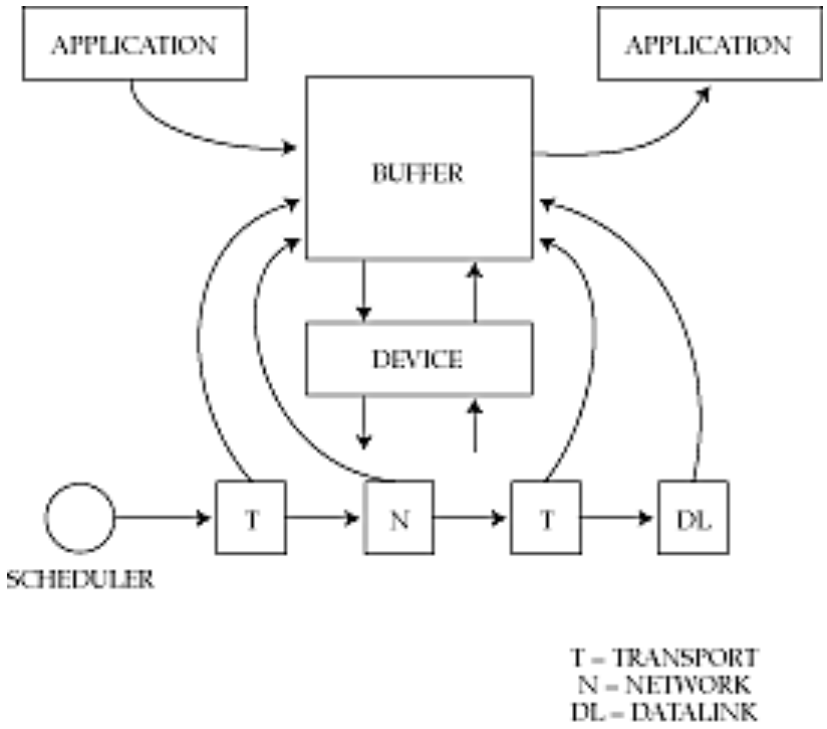
Interface choices

- Single-context
- Tasks
- Upcalls

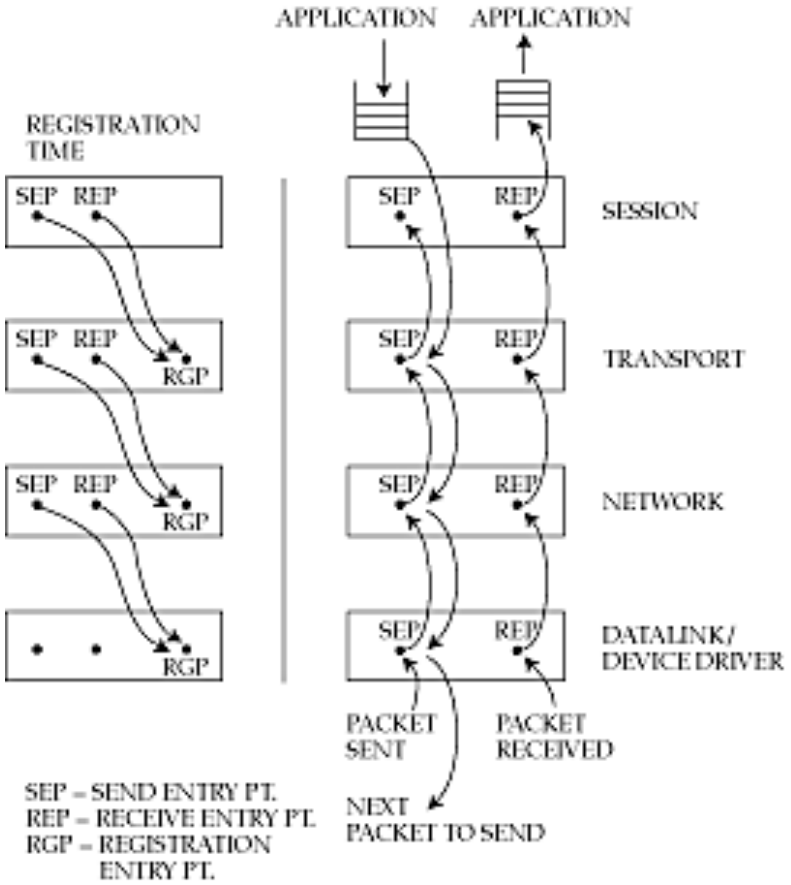
(1) Single context (shepherd threads)



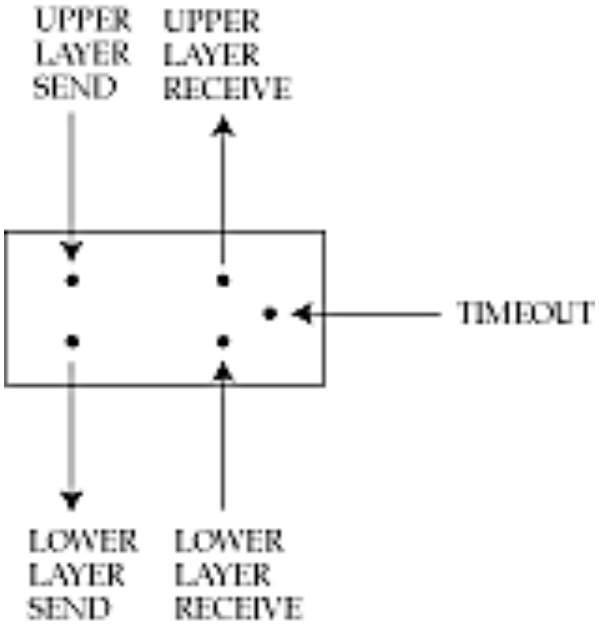
(2) Tasks (pseudo-processes)



(3) Upcalls


APPLICATION
APPLICATION
REGISTRATION TIME
SEP
REP
RGP
SESSION
TRANSPORT
NETWORK
DATALINK/ DEVICE DRIVER
PACKET SENT
PACKET RECEIVED
NEXT PACKET TO SEND
SEP – SEND ENTRY PT.
REP – RECEIVE ENTRY PT.
RGP – REGISTRATION ENTRY PT.

Implementation of each layer



Module 4: Techniques for system design

Overview

- What is system design?
- Critical resources
- Tools and techniques
- Rules of thumb

What is system design?

- A computer network provides computation, storage and transmission resources
- System design is the art and science of putting resources together into a harmonious whole
- Extract the most from what you have

Goal

- In any system, some resources are more freely available than others
  - high-end PC connected to Internet by a 28.8 modem
  - *constrained* resource is link bandwidth
  - PC CPU and and memory are *unconstrained*
- Maximize a set of performance metrics given a set of resource constraints
- Explicitly identifying constraints and metrics helps in designing efficient systems
- Example
  - maximize reliability and MPG for a car that costs less than $10,000 to manufacture

System design in real life

- Can't always quantify and control all aspects of a system
- Criteria such as scalability, modularity, extensibility, and elegance are important, but unquantifiable
- Rapid technological change can add or remove resource constraints (example?)
  - an ideal design is 'future proof'
- Market conditions may dictate changes to design halfway through the process
- International standards, which themselves change, also impose constraints
- Nevertheless, still possible to identify some principles

Some common resources

- Most resources are a combination of
  - time
  - space
  - computation
  - money
  - labor

(1) Time

- Shows up in many constraints
  - deadline for task completion
  - time to market
  - mean time between failures
- Metrics
  - *response time*: mean time to complete a task
  - *throughput*: number of tasks completed per unit time
  - *degree of parallelism* = response time * throughput
    - 20 tasks complete in 10 seconds, and each task takes 3 seconds
    - => degree of parallelism = 3 * 20/10 = 6

(2) Space

- Shows up as
  - limit to available memory (kilobytes)
  - bandwidth (kilobits)
    - Note: 1 kilobit/s = 1000 bits/sec, but 1 kilobyte/s = 1024 bits/sec!

(3) Computation

- Amount of processing that can be done in unit time
- Can increase computing power by
  - using more processors
  - waiting for a while!

(4) Money

- Constrains
  - what components can be used
  - what price users are willing to pay for a service
  - the number of engineers available to complete a task

(5) Labor

- Human effort required to design and build a system
- Constrains what can be done, and how fast
- Also, the level of training determines how much sophistication can be assumed on the part of the users

(6) Social constraints

- Standards
  - force design to conform to requirements that may or may not make sense
  - underspecified standard can faulty and non-interoperable implementations
- Market requirements
  - products may need to be backwards compatible
  - may need to use a particular operating system
  - example
    - GUI-centric design

(7) Scaling

- A design constraint, rather than a resource constraint
- Cannot use any centralized elements in the design
  - forces the use of complicated distributed algorithms
- Hard to measure
  - but necessary for success

Common design techniques

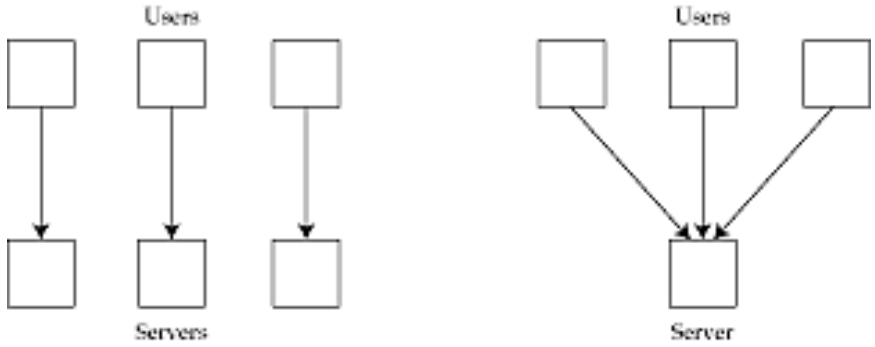
- Key concept: *bottleneck*
  - the most constrained element in a system
- System performance improves by removing bottleneck
  - but creates new bottlenecks
- In a *balanced* system, all resources are simultaneously bottlenecked
  - this is optimal
  - but nearly impossible to achieve
  - in practice, bottlenecks move from one part of the system to another
  - example: Ford Model T

Top level goal

- Use unconstrained resources to alleviate bottleneck
- How to do this?
- Several standard techniques allow us to trade off one resource for another

(1) Multiplexing

- Another word for sharing
- Trades time and space for money
- Users see an increased response time, and take up space when waiting, but the system costs less
  - economies of scale



(1) Multiplexing (contd.)

- Examples
  - multiplexed links
  - shared memory
- Another way to look at a shared resource
  - *unshared virtual resource*
- *Server* controls access to the shared resource
  - uses a *schedule* to resolve contention
  - choice of scheduling critical in proving quality of service guarantees

(2) Statistical multiplexing

- Suppose resource has capacity C
- Shared by N identical tasks
- Each task requires capacity c
- If Nc <= C, then the resource is underloaded
- If at most 10% of tasks active, then C >= Nc/10 is enough
  - we have used statistical knowledge of users to reduce system cost
  - this is *statistical multiplexing gain*

Statistical multiplexing (contd.)

- Two types: spatial and temporal
- Spatial
  - we expect only a fraction of tasks to be simultaneously active
- Temporal
  - we expect a task to be active only part of the time
    - e.g silence periods during a voice call

Example of statistical multiplexing gain

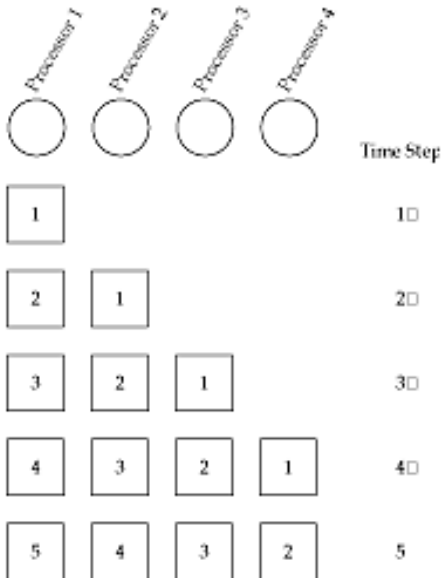
- Consider a 100 room hotel
- How many external phone lines does it need?
  - each line costs money to install and rent
  - tradeoff
- What if a voice call is active only 40% of the time?
  - can get both spatial and temporal statistical multiplexing gain
  - but only in a packet-switched network (why?)
- Remember
  - to get SMG, we need good statistics!
  - if statistics are incorrect or change over time, we're in trouble
  - example: road system

(3) Pipelining

- Suppose you wanted to complete a task in less time
- Could you use more processors/cores to do so?
- Yes, if you can break up the task into *independent* subtasks
  - such as downloading images into a browser
  - optimal if all subtasks take the same time
- What if subtasks are dependent?
  - for instance, a subtask may not begin execution before another ends
  - such as in cooking
- Then, having more processors doesn't always help (example?)

Pipelining (contd.)

- Special case of *serially dependent* subtasks
  - a subtask depends only on previous one in execution chain
- Can use a *pipeline*
  - think of an assembly line



Pipelining (contd.)

- What is the best decomposition?
- If sum of times taken by all stages = R
- Slowest stage takes time S
- Throughput = 1/S
- Response time = R
- Degree of parallelism = R/S
- Maximize parallelism when R/S = N, so that S = R/N => equal stages
  - *balanced pipeline*

(4) Batching

- Group tasks together to amortize overhead
- Only works when overhead for N tasks < N time overhead for one task (i.e. *nonlinear*)
- Also, time taken to accumulate a batch shouldn't be too long
- We're trading off reduced overhead and increased throughput for a longer worst case response time

(5) Exploiting locality

- If the system accessed some data at a given time, it is likely that it will access the same or ‘nearby’ data ‘soon’
- Nearby => spatial
- Soon => temporal
- Both may coexist
- Exploit it if you can
  - caching
    - get the speed of RAM and the capacity of disk

(6) Optimizing the common case

- 80/20 rule
  - 80% of the time is spent in 20% of the code
- Optimize the 20% that counts
  - need to measure first!
  - RISC
- How much does it help?
  - Amdahl's law
  - Execution time after improvement = (execution affected by improvement / amount of improvement) + execution unaffected
  - beyond a point, speeding up the common case doesn't help

(7) Using hierarchy

- Recursive decomposition of a system into smaller pieces that depend only on parent for proper execution
- No single point of control
- Highly scaleable
- Leaf-to-leaf communication can be expensive
  - shortcuts help

(8) Binding and indirection

- Abstraction is good
  - allows generality of description
  - e.g. mail aliases
- Binding: translation from an abstraction to an instance
- If translation table is stored in a well known place, we can bind automatically
  - indirection
- Examples
  - mail alias file
  - page table
  - telephone numbers in a cellular system

(9) Virtualization

- A combination of indirection and multiplexing
- Refer to a virtual resource that gets matched to an instance at run time
- Build system as if real resource were available
  - virtual memory
  - virtual modem
  - Santa Claus
- Can cleanly and dynamically reconfigure a system

(10) Randomization

- A powerful tool
  - allows us to break a tie fairly
  - immune to systematic failure in any component
- Examples
  - resolving contention in a broadcast medium
  - choosing multicast timeouts
  - gossip protocols

(11) Soft state

- State: memory in the system that influences future behavior
  - for instance, VCI translation table
- State is created in many different ways
  - signaling
  - network management
  - routing
- How to delete it?
- Soft state => delete on a timer
- If you want to keep it, refresh
- Automatically cleans up after a failure
  - but increases bandwidth requirement

(12) Representing state explicitly

- Network elements often need to exchange state
- Can do this implicitly or explicitly
- Where possible, use explicit state exchange
  - makes system easier to debug (reduces time) but can reduce efficiency

(13) Hysteresis

- Suppose system changes state depending on whether a variable is above or below a threshold
- Problem if variable fluctuates near threshold
  - rapid fluctuations in system state
- Use state-dependent threshold, or *hysteresis*
  - reduces efficiency but improves stability

(14) Separating data and control

- Divide actions that happen once per data transfer from actions that happen once per packet
  - Data path and control path
- Can increase throughput by minimizing actions in data path
- Example
  - connection-oriented networks
- On the other hand, keeping control information in data element has its advantages
  - per-packet QoS

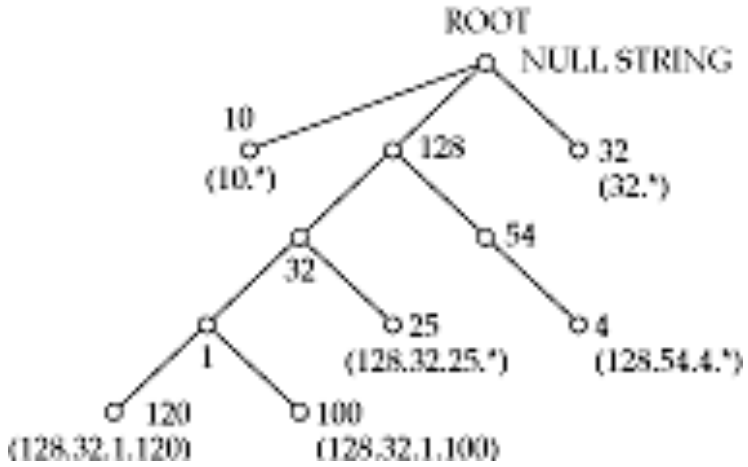
(15) Allowing extensibility

- Always a good idea to leave hooks that allow for future growth
- Design for the future because it will be here sooner than you think
- Examples
  - Version field in header
  - Modem negotiation
- Reduces performance

(16) Hashing and Bloom filters

- Hashes allow for rapid testing of membership of a string in a set
  - hash function H defined over an arbitrary string maps to array index h
    - H(String) = h
  - if(hash_table[h] ==1) then member else not
  - problem with collisions, i.e., H(S1) = H(S2) = h
- Bloom filter defines multiple hash functions H1, H2,…Hk
- S is a member iff H1(S) = H2(S) = …= Hk(S) = 1
- Reduces possibility of collisions even with small hash tables
- False positives remain a problem

(17) Tries



- Two ways to improve performance
  - cache recently used addresses in a CAM
  - move common entries up to a higher level (match longer strings)

(18) Ternary CAMs

- Allows strings with elements 0, 1, X
- Each TCAM entry can be wildcard
  - allows aggregation
- Fast lookup of maps from contiguous name sub-spaces
  - e.g., for routing

(19) Name-space encapsulation

- Clever trick to deal with legacy systems
- Pre-pend existing namespaces with an encapsulating string
  - ftp://
  - postal://

(20) Tolerating server and device failure

- All state stored in stable storage and updated when state changes
- To improve performance
  - keep a copy in RAM
  - reads are from RAM
  - writes are to RAM as well as stable store
- On reboot, in-memory state recreated from stable store

(21) Timing wheels

- Timers support four operations:
  - start timer
  - stop timer
  - timeout operations
  - per time-tick operations
- Using hashed and hashed hierarchical timing wheels (or a variant called calendar queues), these operations can be O(1) expected case
- Essentially, associate events with an array of time values rather than associating times with an array of events

More rules of thumb

- Design for simplicity, adding complexity only when you must
- Use ASCII instead of integers: clarity is better than cleverness
- Fine tune inner loops
- Choose good data structures
- Beware of data and non-data touching touching overheads
- Minimize number of packets sent
- Send the largest packets possible
- Use hardware if possible
- Exploit (persistent) application properties

Module 5: Testing

Overview

- Some techniques for protocol testing
  - formal verification
  - queueing analysis
  - emulation
  - simulation
  - use of the the 'live Internet'

(1) Formal verification

- Each peer is represented by a state machine
- Message transmission and receipt lead to state transitions at communicating peers
- Goal is to discover if some set of message exchanges, including losses, duplications, and re-orderings leads to deadlock (i.e., no progress possible) or entry into a bad state
- Problem is state explosion
- Many clever techniques to mitigate this problem
- Bottom line: useful but of limited use in practice
  - too hard for the average practitioner

(2) Queueing analysis

- Model arrivals to a server and departures from a server as a stochastic process
- If these processes are well-behaved (typically Poisson) we can compute the distribution of queuing delays
- Provides excellent insights into a system
- But makes too many assumptions to be useful in practice

(3) Emulation

- Exactly reproduces protocol behavior
  - by implementing the protocol in a controlled testbed and testing its behavior
- Tests the actual protocol and workload
- But difficult to set up and scale
- Realistic workload emulation is nearly impossible!

(4) Simulation

- Studies a software model of the protocol
  - in some cases, the protocol is emulated, but the rest of the system is simulated
- The most popular technique
  - complete control over environment
  - several standard simulator packages widely available
- Pitfalls
  - lack of validation
  - cold start
  - not running the simulation long enough for metric to achieve stability
  - statistical significance of results

(5) Testing in the live Internet

- ‘Just do it’ approach
- Completely uncontrolled environment
- Can lead to ‘success disasters’

Module 6: Pitfalls

Overview

- Things to watch out for
  - debuggability
  - race conditions
  - failing unsafely
  - corner cases
  - implementations that lie
  - performance problems

(1) Debuggability

- Bugs are inevitable
- Often the only way to debug a distributed system is by printing out events
- Invest in building a good logging system
  - standard event formats that can be post-processed
  - debugging levels
  - debugging node that collects events from all nodes

(2) Race conditions

- Protect all critical sections
- Multiple actions that stem from the same event should coordinate with each other
  - example: reader-writer should use a synchronized list

(3) Failing unsafely

- Consider the consequences of each failure
- Ideally, failures should only reduce performance without compromising correctness
- Examples
  - storing state in stable storage
  - link failure in OSPF

(4) Corner cases

- Always consider extremal values of input parameters ('corner cases')
  - as well as roll over of finite counters
- A quick way to test for protocol correctness
- Examples:
  - does the system work for zero-length packets as well as maximum size-packets?
  - what happens when sequence numbers reach the largest possible size?

(5) Implementations that lie

- Sometimes, values received from a peer may not be correct
  - buggy implementation
  - undetected data corruption
  - malicious nodes
- Two maxims apply
  - ‘trust but verify’
  - ‘be liberal in what you accept and conservative in what you send’
- Examples
  - TCP RST (‘I am confused’)
  - Byzantine agreement

(6) Performance problems

- Performance relevant only after correctness
- Collect good metrics
- Use the techniques described earlier
- In my experience, getting 10x improvements in any metric can be achieved with moderate effort

Conclusions

- Protocol design and implementation is a complex problem
- Many inherent challenges and incompatible requirements
- We have a number of tools at our disposal and many working systems that scale to hundreds of millions of users
- By studying these systems and some care, it is possible to build robust systems that scale well

The great aim of education is not knowledge but action.

Herbert Spencer