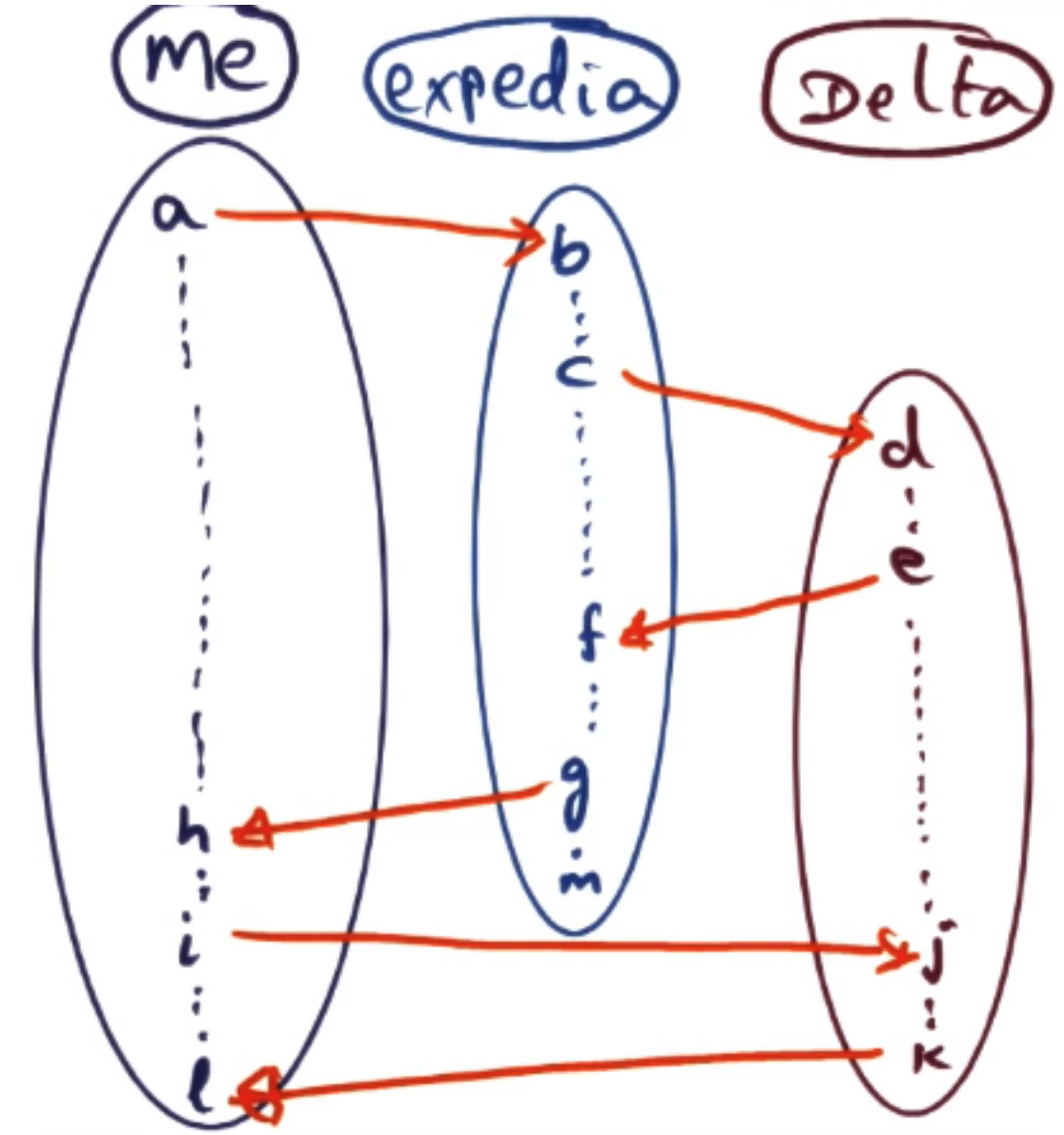
Distributed Systems

Definitions

```
* What is a distributed system?
    - Nodes connected by LAN/WAN
    - Communication only via messages (fiber, cable, satellite)
    - Message time >> Event time
    - Lamport's definition: A system is distributed if the message
    transmission time is not negligible compared to the time between events
    in a single process
        + Even a cluster is a distributed system
        + If the time to pass messages outweighs the computation time, there's no gain
* Distributed System Example
    - Booking a flight through Expedia; communication between user, Expedia, Delta
    - Assumptions
        + Processes are sequential, events are totally ordered (d->e, f->g, h->i)
        + Send before receive (a->b, e->f)
        + Called the "happened before" relationship
```



Distributed System Example

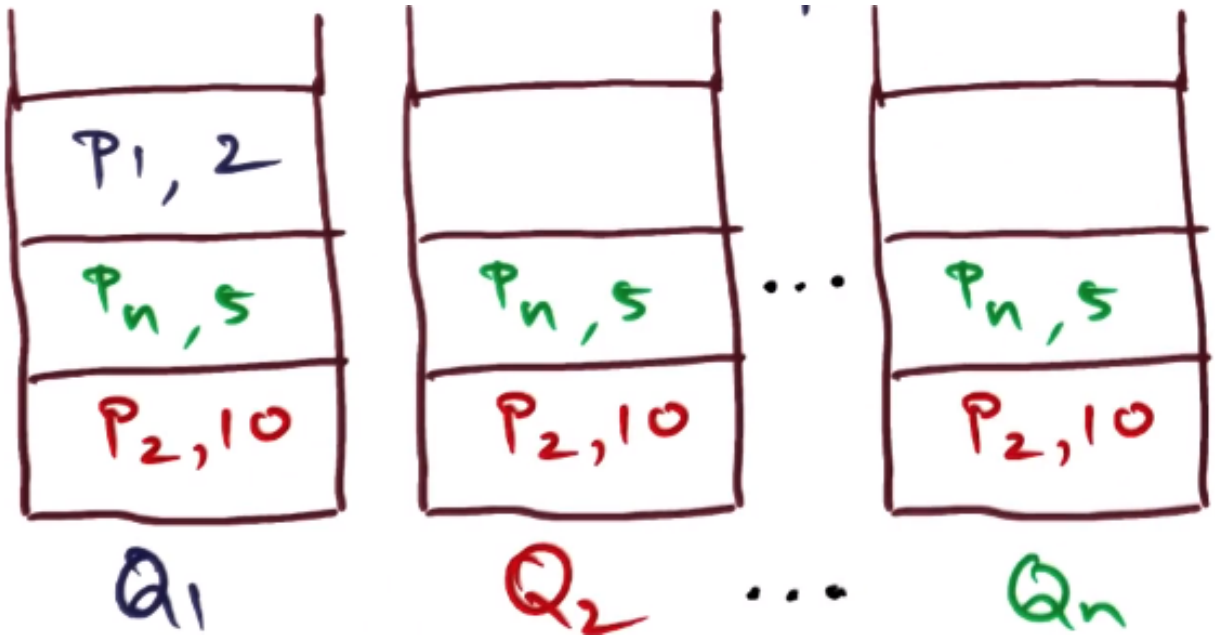
```
* Happened Before Relation
    - If a happened before b (a->b)...
        + a and b are executing in the same process, or...
        + A communication event happens that connects a and b
    - Transitive (a->b, b->c then a->c)
    - If two events aren't connected by a happened-before relation, they're
    concurrent events
    - Happened before is insufficient for creating a total order of events
```

Lamport Clocks

```
* Each node knows...
    - Its own events
    - Its communication events with other nodes
```

* Lamport's Logical Clock
    - Monotonic increase of own event times
        + Condition 1: Ci(a) < Ci(b)
    - Message receipt time greater than send time
        + Condition 2: Ci(a) < Cj(d)
        + Choose Cj(d) = max(Ci(a)++, Cj)
    - Timestamp of concurrent events?
* If C(a) < C(b), then...
    - a -> b (a happened before b) False
    - b -> a (b happened before a) False
    - a -> b (iff a & b are events in same process or a is a send event and b is the corresponding receive event) True
* Logical Clock Conditions
    - Monotonic increase of own event times
        + Condition 1: Ci(a) < Ci(b)
    - Message receipt time greater than send time
        + Condition 2: Ci(a) < Cj(d)
        + Choose Cj(d) = max(Ci(a)++, Cj)
    - Concurrent events (b and d) receive arbitrary timestamps
    - C(x) < C(y) does not imply x happened before y (x -> y)
* Need for a Total Order
    - Is partial order sufficient for deterministic distributed algorithms?
        + Sufficient for many situations
    - Situations exist where a total order of events is needed
        + Two requests for a resource with identical timestamps require some form of resolution (resource allocation)
        + Car example: Age wins
* Lamport's Total Order
    - Condition for total order (=>)
    - a => b iff
        + Ci(a) < Cj(b) or
        + Ci(a) == Cj(b) && Pi << Pj (<< arbitrary "well known" condition to break a tie)
    - No single total order
* Distributed Mutual Exclusion Lock Algorithm
    - No shared memory in a distributed system
    - Each process has its own private queue, ordered by a happened-before relationship
    - When a process wants to acquire the lock, it sends its local timestamp to all other processes
    - Each process adds it to its local queue in the appropriate place and sends an acknowledgement
    - Break ties by giving priority to process with lowest process ID
    - Queues aren't necessarily identical across all processes
        + Message times are variable
        + P1's message in transit
        + P1 received P2's and Pn's messages subsequently
    - How do you determine if you have the lock?
        + My request must be at the top of the queue
        + Received acks from others OR later lock requests
    - Releasing the lock
        + Remove entry from local queue
        + P1 sends unlock message
        + Peers remove P1's completed request from their respective queues

- Correctness
    + Messages arrive in order
    + No message loss
    + Queues are totally ordered => by Lamport's logical clocks plus pid to break ties



Distributed Mutual Exclusion Lock Example

* Distributed Mututal Exclusion Lock Message Complexity
    - Lock(L) => N-1 Request messages and N-1 Ack messages
    - Unlock(L) => N-1 Unlock messages
    - Total => 3(N-1) messages
    - No ack for unlock message due to the assumption of no message loss
    - Can we do better?
        + Defer Acks if my request precedes yours
        + Combine with unlock => 2(N-1)
    - Reducing message complexity is a popular research topic
* Real World Scenario
    - Logical time might not be sufficient in real world scenarios
    - P1 owes P2 money; out of band communication says the money will be available at 5 PM
    - P2 issues a request to debit the money at 8 PM
    - P2's time is far ahead of real time
    - P1 credits the account at 5 PM, but the debit request came earlier, so it was declined
    - Clocks are circuitry that can become out of sync
* Lamport's Physical Clock
    - a |-> b (a occurred before b in physical time) Ci(a) < Cj(b)
    - Physical clock conditions:
        1. PC1 (bound on individual clock drift)
            + abs(dCi(t)/dt - 1) < k for all i; k << 1
        2. PC2 (bound on mutual drift)
            + For all i,j: Ci(t) - Cj(t) < e

    - k and e must be negligible relative to IPC time
* IPC Time and Clock Drift
    - Let u be lower bound on IPC (time elapsed between message send/recv)
    - To avoid anomalies if a on Pi |-> b on Pj
        1. Ci(t+u) - Cj(t) > 0
        2. Ci(t+u) - Ci(t) > u(1-k) (difference equation formulation of PC1)
    - Using 1 and 2 and bound e on mutual drift
        + u >= e/(1-k) to avoid anomalies
    - To maintain a physical ordering, maintaining small bounds on individual and mutual clock drift is imperative (relative to IPC time u)
* Lamport Clocks Conclusion
    - Theoretical underpinning for achieving deterministic execution in distributed systems despite the fact that there are nondeterminisms due to the network and clock drift

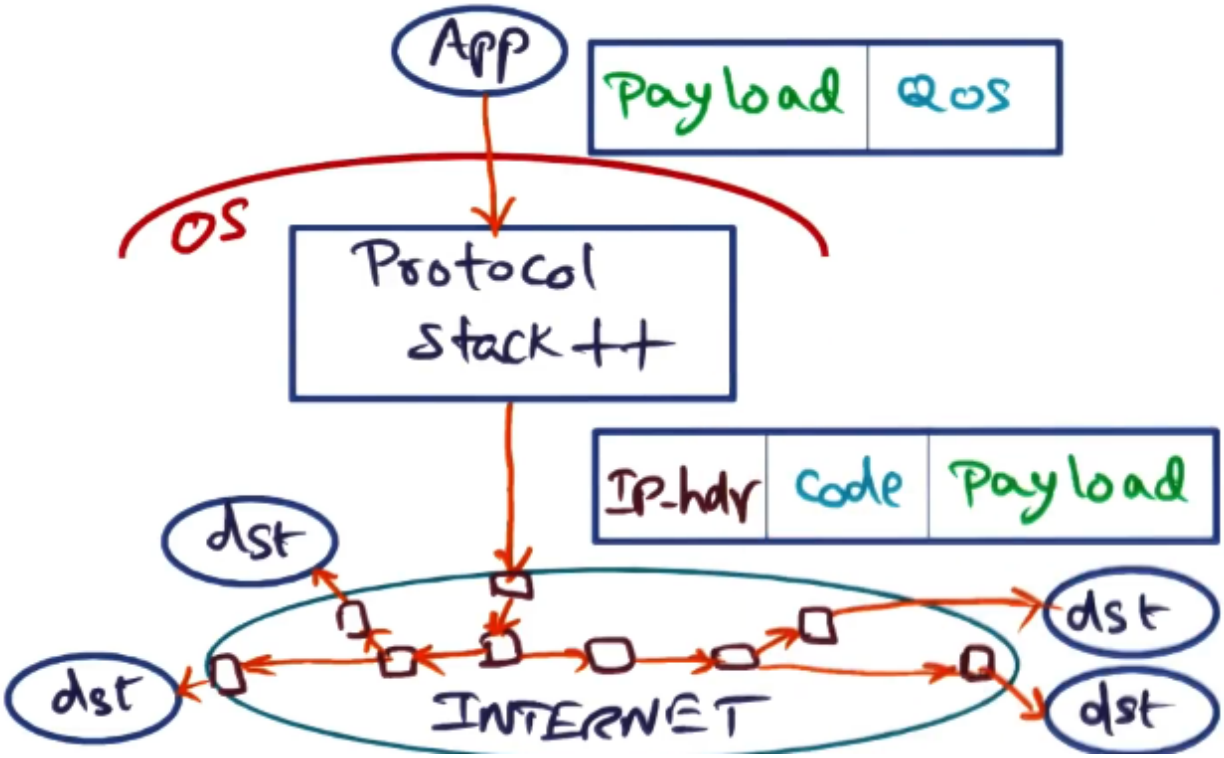
Latency Limits

* Latency vs Throughput
    - Latency: Elapsed time (time to walk from office to classroom)
    - Throughput: Events per unit time (number of people able to concurrently make the walk)
        + Bandwidth is a measure of throughput
    - RPC is the basis for client/server based distributed systems
    - RPC performance is crucial in building performant systems
        + Hardware overhead: Interface between network and computer (NIC)
        + DMA: NIC uses bus to transfer data from system memory to local memory without CPU intervention
        + Software overhead: Operating system must make message available in memory of process (OS designer must minimize this)
* Components of RPC Latency
    1. Client call: Client sets up arguments, calls kernel, marshals data
    2. Controller latency: NIC DMAs message into buffer and sends
    3. Time on wire: Transmission protocols take some amount of time
    4. Interrupt handling: OS must handle message receipt
    5. Server setup to execute call: Unmarshals, moves data, calls procedure
    6. Server execution + reply: Time depends on specific procedure
    7. Client setup to receive results and restart: Reverse of sending process
* Sources of Overhead on RPC
    - Marshaling
    - Data Copying
    - Control Transfer
    - Protocol Processing
    - Reducing overhead: Take advantage of what hardware gives you
* Marshaling and Data Copying
    - Marshaling: Accumulate arguments into a continuous network packet
        + Three copies
            1. Client stub
            2. Kernel buffer
            3. DMA to controller
        + First call can be eliminated by marshaling directly into the kernel buffer
        + Alternatively, implement a shared descriptor between client stub and kernel. This allows the client to alert the kernel that it has data ready to send and describes how the data is arranged in memory

        (starting point of data item and length).
* Control Transfer
    - Context switches that must happen to affect an RPC call
        + 4 switches
            1. Client to kernel (network call here)
            2. Kernel to server
            3. Server to kernel (network call here)
            4. Kernel to client
        + Only two are in the RPC critical path: kernel to server and kernel to client (2,4)
        + Other two are simply to ensure the node isn't being underutilized
    - Can overlap context switch with network communication to utilize node
    - On the client side, if we spin instead of context switch, we can reduce number of context switches to one
        + Should only do this if we know the call plus network latency will be fast
* Protocol Processing
    - Performance and reliability are often at odds with each other
    - What transport should be used for RPC?
        + LAN is reliable -> reduce latency
    - Choices for reducing latency in transport
        + No low level acks
        + Hardware checksum for packet integrity
        + No client side buffering since client blocked (if message is lost in transmission, retransmit from buffer)
        + Overlap server side buffering with result transmission (same as above, but on server. Recomputing may be expensive, so this is needed. Can overlap this computation with sending the message)
    - All of these issues are common in wide area networks, but considerably less likely in a LAN, so these extra checks and buffers are unnecessary

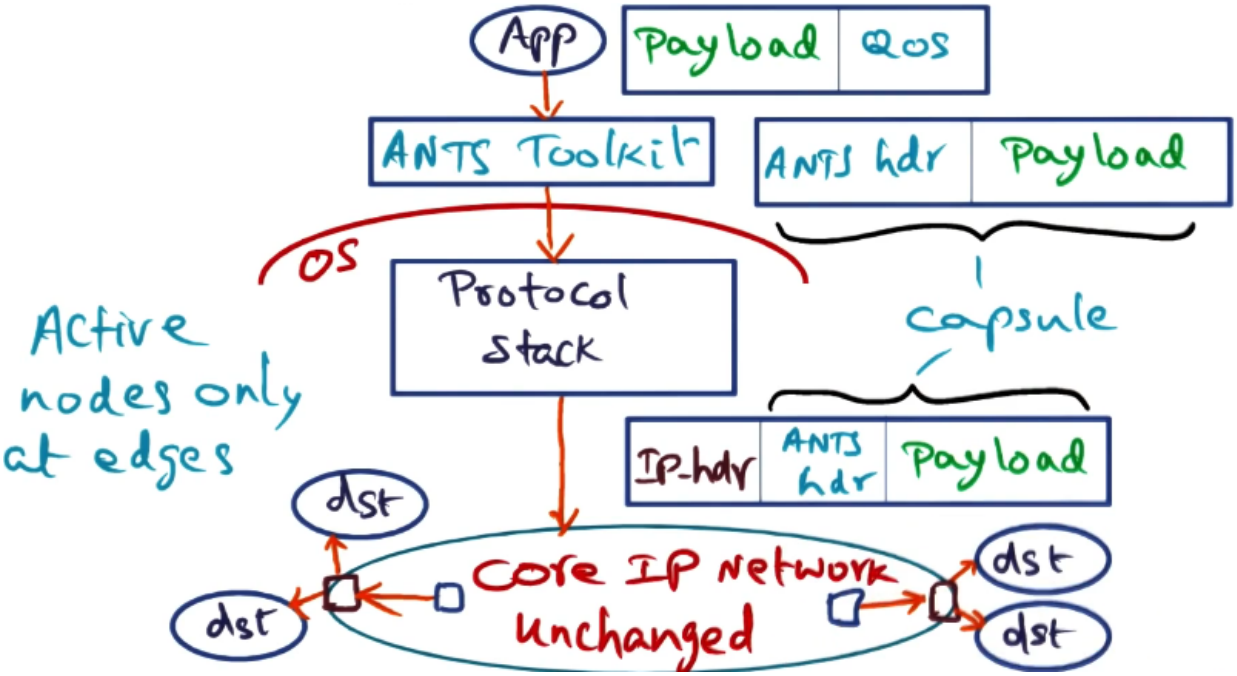
Active Networks

* Routing on the Internet
    - When a system sends a packet, it must go through many intermediate routers to reach its destination
    - Routers simply forward packets (just a table lookup)
    - Making a node active: Next hop isn't just a table lookup. Instead, the router actually executes code
        + How and who writes this code?
        + How do we guarantee that this code isn't malicious?
* Active Networks Example
    - Instead of sending N messages to N people that are located nearby, can we send a single message and have the final router demultiplex the data to send to each recipient?
* How to Implement the Vision
    - OS should provide quality of service APIs (real time constraints, etc)
    - OS will synthesize code to give hints to the network
    - OS takes QoS contraints and appends them as executable code to the network packet
    - No guarantee that every node will have all of the code to handle the injected code
        + Modifying the protocol stack is nontrivial



Active Network Layout

* ANTS Toolkit
    - ANTS: Active Node Transfer System
    - Appends a header to the payload before sending it to the OS
        + Lives above the OS
    - If a node isn't an active node, it simply uses the IP header
    - If a node is an active node, it interprets the ANTS header and does as instructed
    - Not all routers are open to processing specialized code
        + Active nodes are only at the edge of the network
        + Core IP network is unchanged



ANTS Toolkit

```
* ANTS Capsule and API
    - ANTS Capsule
        + IP header (normal)
        + Type: Identify the code that should be executed to process this
        capsule (MD5 hash of the code)
        + Prev: Identity of the upstream node that processed this capsule
        + Capsule doesn't contain code, only a type field to determine what
        the code should be
    - Purpose: Forwarding packets through the network intelligently
    - ANTS API
        + int getAddress(): Get local node address
        + ChannelObject getChannel(): Get receive channel
        + Extension findExtension(String ext): Locate extended services
        + long time(): Get local time
        + Object put(Object key, Object val, int age): Put object in soft-store
        + Object get(Object key): Get object from soft-store
        + Object remove(Object key): Remove object from soft-store
        + void routeForNode(Capsule c, int n): Send capsule towards node
        + void deliverToApp(Capsule c, int a): Deliver capsule to local app
        + void log(String msg): Log a debugging message
    - ANTS API is designed to be minimal
        + Routers are public Internet
        + Must be easy to program
        + Must be easy to debug, maintain, and understand
        + Must be fast
* Capsule Implementation
    - Action on capsule arrival
        + Type field: fingerprint for capsule code (MD5 hash originally)
        + Demand load capsule code from prev node by sending request
        + If node has seen capsules of this type before, it can simply
```

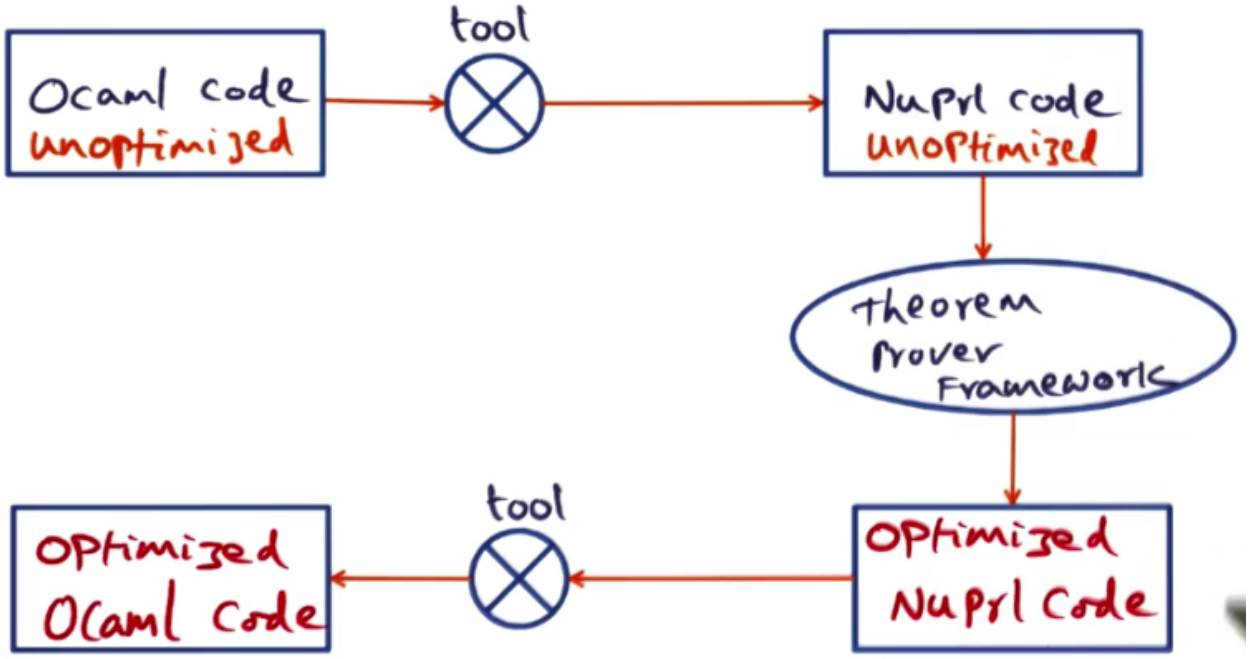
        retrieve the code from its soft-store and execute it
        + If node hasn't seen capsules of the type, it sends a request to
        the previous node to acquire the code
        + When code arrives, the node computes the hash and compares
        + Save in soft store for future use
        + Typically send many packets, so even though the first packet might
        be slow, subsequent packets are likely to be much faster (locality)
        + If the previous node doesn't have the code, the current node
        simply drops the packet so the source knows to retransmit
        + Capsule may need to evict code from the soft-store (like a cache)
* Potential Applications
    - Protocol independent multicast
    - Reliable multicast
    - Congestion notification
    - Private IP (PIP)
    - Anycasting
    - Useful for building applications that are difficult to deploy on the
    Internet
        + Network flow may require a different setup from the physical setup
        + Example of sending card to multiple people
    - Active network paradigm should provide
        + Accessibility
        + Speed
        + Compact
        + Doesn't rely on all nodes being active
    - Provide network layer functionality, not end application functionality
* Pros and Cons of Active Networks
    - Pros:
        1. Flexibility from application perspective
    - Cons:
        1. Protection threads (prevent interference with other network flows)
            + ANTS runtime safety -> Java sandboxing
            + Code spoofing -> robust fingerprint (hash)
            + Soft state integrity -> restricted API
        2. Resource management threads (prevent executing code from going wild)
            + At each node -> restricted API
            + Flooding the network -> Internet already susceptible
* Roadblocks
    - What can be roadblocks to the active networks vision?
        + Need router vendors to commit to the idea (Cisco)
        + ANTS software routing cannot match throughput needed in Internet
        core (routers typically only operate in hardware, so running
        software on top can be slow)
* Feasibility
    - Router makers are loath to opening up the network
        + Only feasible at the edge of the network
    - Software routing cannot match hardware routing
        + Only feasible at the edge of the network
    - Social and psychological reasons
        + Hard for user community to accept arbitrary code executing in the
        public routing fabric
* Active Networks Conclusion
    - Active networks were ahead of their time (solution looking for a problem)
    - No killer app to popularize the idea

- Software-defined networking: Modern incarnation of active networks
- Cloud computing promotes a model of utility computing where multiple tenants can host corporate networks on the same computational resources in the data center
- Need perfect isolation of network traffic of different businesses
    + Solution: Virtualize the network

Systems from Components

* Systems from Components Introduction
    - VLSI design for CPUs uses a component-based approach so individual components can be tested and optimized
    - Applying this to OS design can provide the same benefits
        + This is orthogonal to monolithic vs microkernel approach
    - Does this cause inefficiencies in performance? Loss of locality? Unnecessary redundancies in terms of copies?
* The Big Picture
    - Design cycle
        1. Specify
            + IOA (IO Automata): Provides C-like syntax
            + Composition operator
        2. Code
            + OCaml: Object Oriented Categorical Abstract Machine Language
            + Object oriented, functional programming language
            + Efficient code similar to C
            + Complement to IOA
        3. Optimize
            + NuPRL: Theoretical framework for optimizing OCaml code
            + Output verified to be functionally equivalent, but optimized
            + Uses theorem proving framework
        4. Deploy
* Digging Deeper from Specification to Implementation
    - Abstract behavioral spec using IOA
        + High level, logical specification of components
        + Specify properties, not how it will be implemented (every packet should have an ack, for example)
        + IOA isn't executable code, but can prove that the behavioral spec meets the properties
    - Concrete behavioral spec
        + Refine the abstract behavioral spec
        + For a queue with FCFS behavior desired, how do we guarantee that the abstract behavioral spec conforms to the desired execution
    - Implementation
        + Scheduling of operations within the concrete behavioral spec
        + Executable code (OCaml)
        + Compact code with automatic garbage collection, memory allocation, and marshaling/unmarshaling built in
        + This implementation is unoptimized; there is no way to easily show OCaml implementation is the same as the IOA specification
        + Don Knuth: "Beware of bugs in the above code. I've only proved it correct, not tried it."
    - Optimization
        + NuPRL is a theorem proving framework is a vehicle where you can make assertions and prove theorems about the equality of the

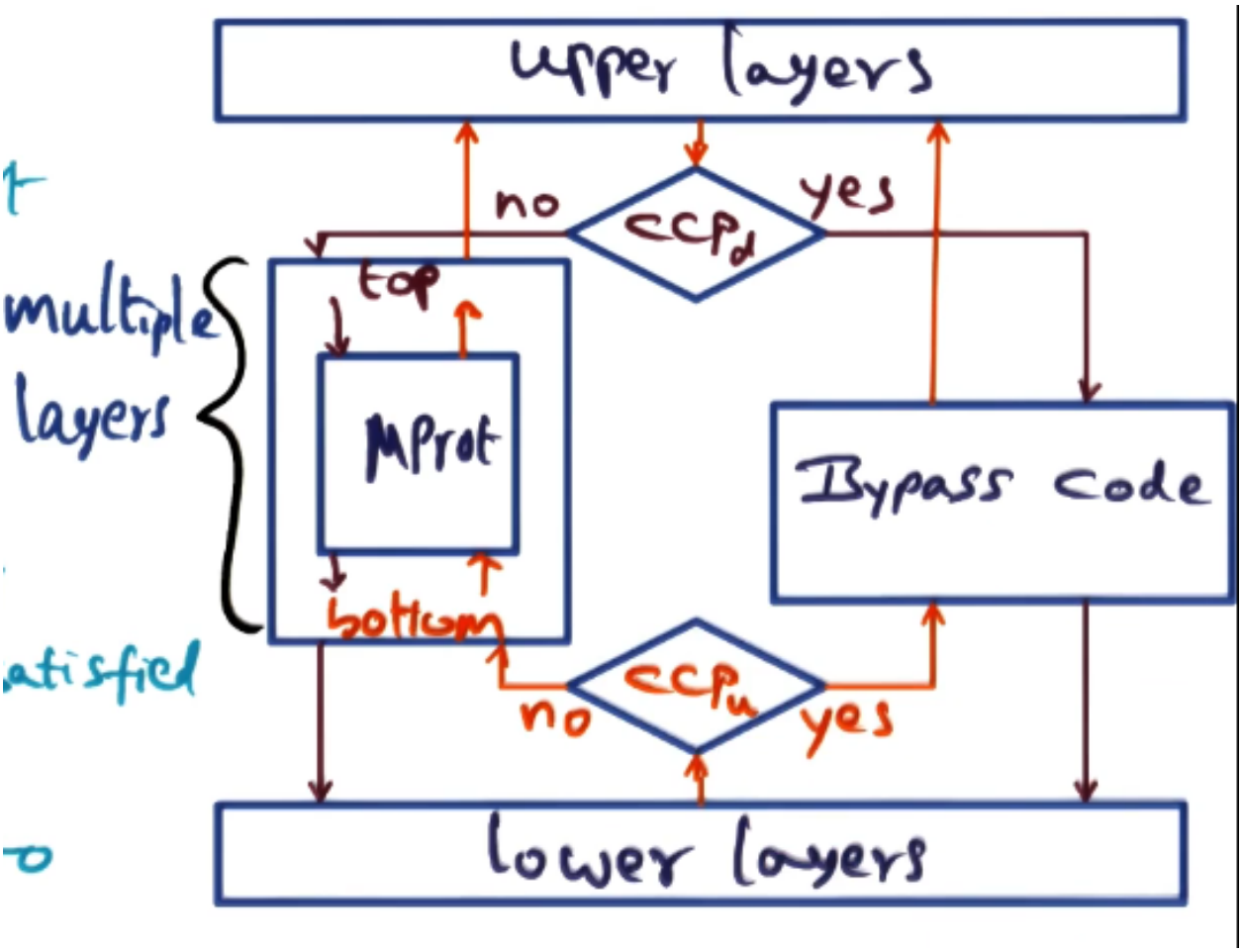
optimized and unoptimized code
+ Theorem prover framework proves that unoptimized and optimized implementations are functionally equivalent
+ Tool to convert OCaml <-> NuPRL



NuPRL Workflow

* Putting the Methodology to Work
    - Start with IOA spec
        + Abstract spec
        + Concrete spec
    - How to synthesize the stack from concrete spec?
    - Getting to an unoptimized OCaml implementation
        + Ensemble suite of microprotocols
        + Flow control, sliding window, encryption, scatter/gather, etc.
        + Each of these are necessary in a complex transport protocol like TCP/IP (one size fits all is never a good approach)
        + Microprotocols have well defined interfaces allowing composition
        + Facilitates component based design
        + Mimic VLSI design
* How to Optimize the Protocol Stack
    - Assembling stack from microprotocols leads to layering, which can lead to inefficiencies
        + Analogy to VLSI component-based design breaks down
        + In VLSI, there is no inefficiency going between components
        + In software, there are well-defined interfaces between components, which may require copying parameters (inefficiency)
    - Several possible source of optimization
        + Explicit memory management instread of implicit garbage collection
        + Avoid marshaling/unmarshaling across layers
        + Buffering in parallel with transmission
        + Header compression

        + Locality enhancement for common code sequences
    - NuPRL can be used to automate the optimization process
* NuPrl to the Rescue
    - Can translate from OCaml to NuPRL
    - Static optimization
        + Requires NuPRL and OCaml expert go layer by layer to identify what transformation code can be applied to each of the layers
        + Function inlining, for example
        + Optimization can be down with NuPRL, but requires expert analysis
        + Many layers in the protocol, would like to collapse to reduce the number of interfaces
    - Dynamic optimization
        + NuPRL does this automatically
        + Identify commonalities between layers
        + Generate bypass code if common case predicate is satisfied
        + If the common case predicate is satisfied, traverse through the bypass code instead of the microprotocol. This is much faster
        + Can collapse all layers into a single predicate
    - Final step
        + Convert back to OCaml
        + NuPRL is only optimizing, not verifying that the OCaml code is adhering to the behavioral spec of IOA
    - NuPRL framework proves equivalence of bypass code to the layers of protocol it is replacing



NuPRL Workflow

```
* Conclusion
    - Do we lose out on performance by following this workflow?
    - Cornell shows that this methodology produces a performance-competitive
    implementation compared to the monolithic implementation
```