Design of High-Performance Software

C. M. Woodside

GOALS:

How to describe, discover, and understand the performance aspects of a system design

Software resource architecture

Solutions to performance problems through principles, performance models, and patterns

Concentration on distributed systems (web, client-server architectures, cluster systems)

Topics

Part A: Introduction on describing software and performance,
- Sequential software on flat physical resources,
- Queue network models,
- Design principles to reduce demands and avoid device (physical) bottlenecks.

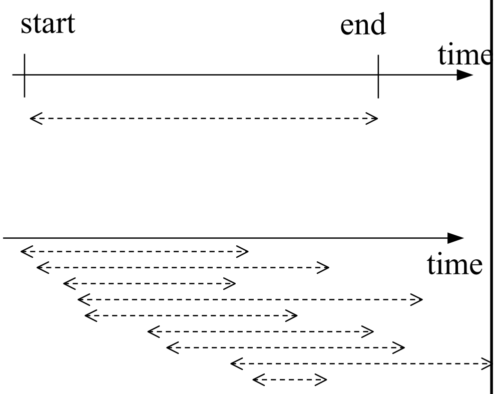
Part B: Layered software and resources,
- Logical resources, threads and limited parallel operation,
- Resource architecture. Layered queue models,
- Design principles to avoid logical bottlenecks.

Part C: Performance descriptions in UML
- Performance profile, examples

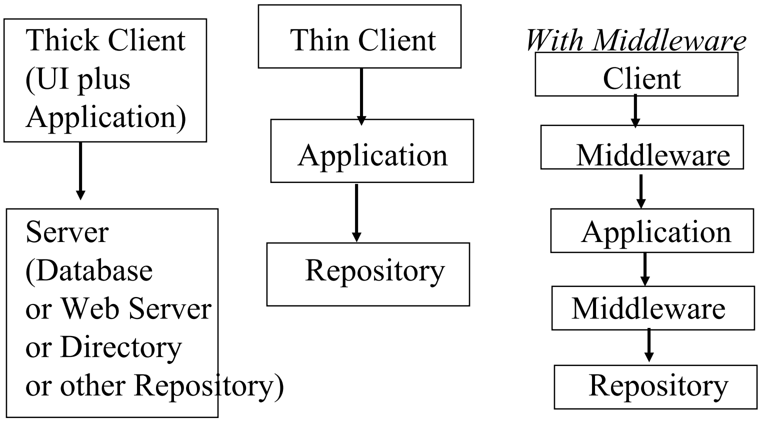
Part D: Path behaviour that includes parallelism,
- and more design principles (e.g. optimism, replication/partitioning).

What is performance?... *Measures*

- Response time:
  - from start event (such as sending a request)
  - to end event (such as getting an output)
  - (web browser)
- Throughput capacity
  - max allowed frequency of responses
  - may overlap in time
  - (many web server clients)
- Utilization of a resource



Distributed Systems: ...Many have a *Client-Server* Architecture



Thick Client (UI plus Application)

Server (Database or Web Server or Directory or other Repository)

Thin Client

Application

Repository

With Middleware

Client

Middleware

Application

Middleware

Repository

- *Generalization:* a program can be a client

Language and Notation

- Performance measures must be defined in a ***context of software structure and behaviour***

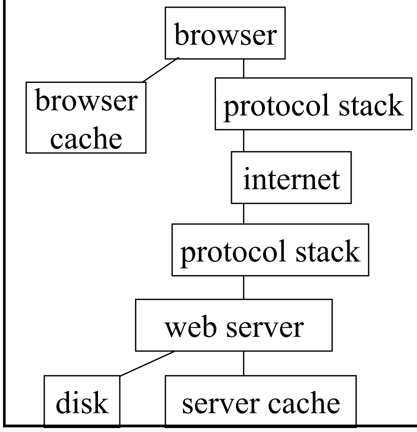
- ***Behaviour:*** what is carried out during a response
  - defined as a ***scenario***
  - a sequence of operations, possibly with alternatives, loops etc.
  - often specified as a ***use case*** for the software
  - we will use **UML** and **Use Case Maps**
- ***Structure:*** software components and relationships; **use UML**
  - architecture or design
  - calling or service relationships
  - containment, inheritance etc are less important for performance

---

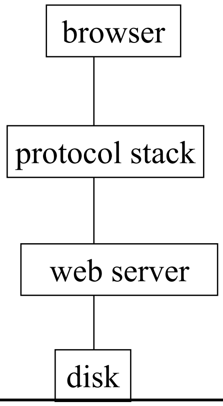
Web Server

- Client is the browser in your PC
- Server is Apache or similar

Components involved


browser
browser cache
protocol stack
internet
protocol stack
web server
disk
server cache

Simplified

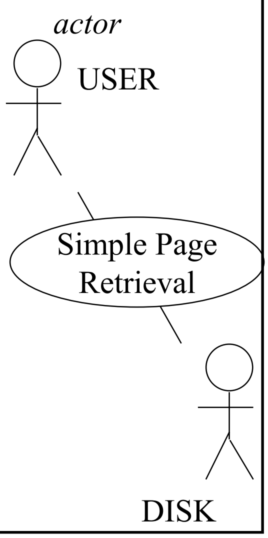

browser
protocol stack
web server
disk

Paths as defined by Use Cases

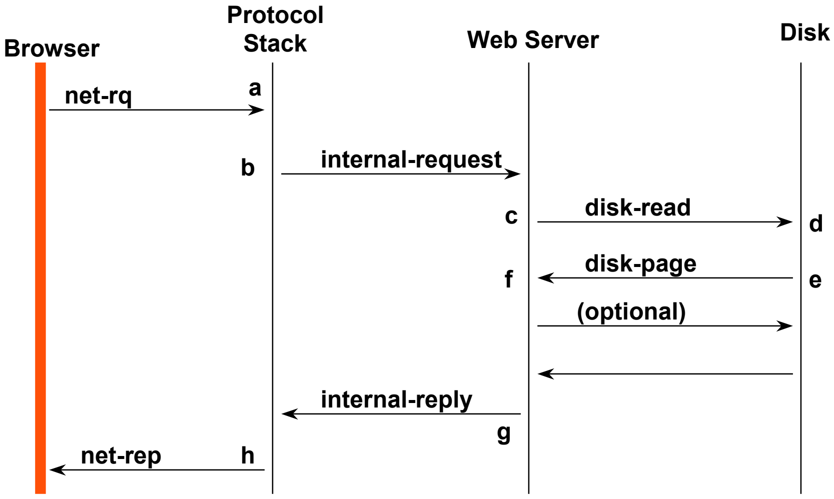
Use case “Simple Page Retrieval” for a web server:

- ...USER generates a page request
- ...USER sends request to SERVER
- ...SERVER reads request and checks its validity
- ...SERVER decodes request to identify location of data
- ...SERVER issues read to DISK for page
- ...DISK finds and returns page
- ...SERVER buffers page
- ...SERVER sends response to user.

..... now develop the use case into one or more scenario, and attach performance parameters... “activity graph”....



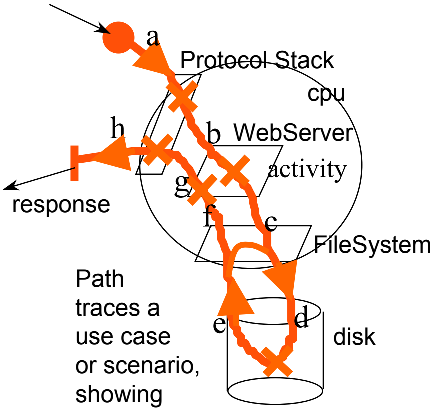
Path in the Web Server... MSC and *events*



Use Case Maps for scenarios over components... Web Server

Web Server

request, request rate



Path traces a use case or scenario, showing locations of execution

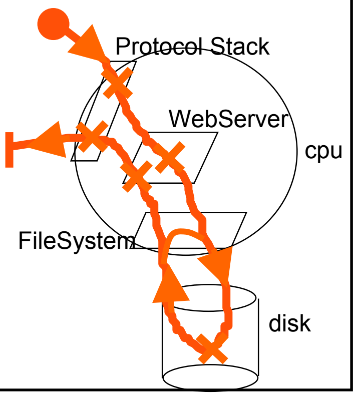
- the file system component was ignored on the sequence diagram... where does it fit?
  - this points to the need for flexible abstractions for operations, so detail can be added

---

Where do performance limits come from?

- long processing times per operation
- fragmentation of operations
  - small procedures
  - small messages
- indirection (and the overhead to resolve it)
- insufficient memory in the server
- ineffective caching
- insufficient THREADS at the server
- or too many threads (overhead)
- slow CPU or DISK
- long network delays

Web Server



Sources and cures for performance limits

Categories of sources.... patterns for cures:

- long CPU time for an operation: shorten it
    - hot-spot seek and remove (80/20 rule "code bottlenecks")
    - fast path processing for special cases
    - low-level code optimizations
    - parallelism
    - optimism
- indirection: remove it
    - early binding
- fragmentation of operations or data: improve "locality"
    - aggregate components, operations, data, messages
- resource bottlenecks: replicate resources or partition operations

---

Examples: Code changes that reduce CPU time

- ***a repository*** returns a record based on some key information
    - faster search for the record
    - storage using a hash table for each key
- ***a matrix calculation*** depends on storage mapping for data tables
    - store on doubleword boundaries
    - operate on data rows that load together into cache (a kind of locality)
- ***in layered systems*** copying of data between layers is common
    - engineer the layers to share data storage
- ***in network management***, data can be accessed singly or in groups
    - batch up transfers
- ***signal processing*** has short fixed-length loops: unroll them (program all the steps, without counters or pointers) (early binding!)

Ex2...Generic code optimization tricks

- overhead to resolve late binding due to indirection:
  - pointers (cost of calculating pointer values)
- code "straightening" can improve cache performance
  - and other compiler optimizations to unroll loops, eliminate +0,...
- flatten inheritance hierarchies (!!)
- code changes to improve fine-grained locality...
  - avoid data copying, batch up storage accesses, cache data or connections,....

Ex3...Examples that involve locality and overhead

- ***open distributed processing*** uses run-time resolution of entities and operations
  - directory look-ups, e.g., use of CORBA to resolve a server
    - cache the server reference to avoid frequent accesses
  - code loaded across a network
  - mobile agents to perform operations
- ***open distributed processing*** also uses interpreted languages ... these have substantial costs... could they be compiled instead
  - early binding....tradeoff flexibility vs performance

Ex4...Examples requiring architecture changes

- ***web servers*** must serve many simultaneous clients. A single server is inadequate; concurrency must be increased.
  - threads for multiple active instances of a service
  - requires concurrency control (e.g. locking) for shared data
- ***web servers, directories and database servers*** need to be scaled up, beyond the concurrency from threads sharing a multiprocessor node
  - replicated servers on separate nodes
    - partitioned data, or fully replicated
  - partitioned operations into specialized versions
- ***signal processors*** must have high throughput which may require parallel processing
- ***IP telephony systems*** may need to mask network latency
  - proxy server can support local operations

Ex5.... Generic architectural or design-level support for performance

- concurrency: threads, pipelining, layers, parallel tasks.
- pre-fetching and caching of data
  - pre-fetching and caching of other bindings (connections, pointers, operation code)
- agents and proxies to avoid network latencies
- optimistic operations
  - quick versions based on assumptions that can only be checked on completion
- layer bypass or cut-through

Ex6...Examples of problems due to dynamics

- ***protocols*** have timeouts.
  - a layer had a fixed timeout that was premature, causing many repeated retransmissions and wasted work.
  - needed to be found and fixed
- ***some network clients*** may time out and retry (e.g. web users on an overloaded server)
  - effort spent on their connections to now is wasted
  - can be tackled by prioritizing the service to penalize long responses, server short ones faster (not software design, but operation strategy)
- ***telephone systems*** also have clients that retry if they cont get a response
  - also resolved by priorities (last-come-first-served!!!)

Ex7.... Examples involving fairness and QoS guarantees

Operational policies such as scheduling may be key here

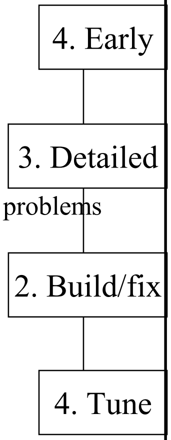
- e.g. ***DiffServ in networks***, similar ideas in end-to-end services
  - identify the responses by group
  - monitor performance of each group, user or response
  - adjust priorities, modify the service or shed load

- ***Priorities*** are coarse but powerful in discriminating
  - new disciplines are more subtle (e.g. fair-share queueing)
- ***Modified service*** is application specific (e.g. reduced fidelity in audio or video)
- ***Shedding load*** can be done at admission of requests
  - or during service... allowing the user to retry

How do we deal with all these problems?

Four approaches:

1. ***tune*** after installation
    - priorities, buffers, configuration parameters
    - YES... design to be tuned. But limited capability to adjust.
2. build now, ***fix it later***
    - concentrate on function first
    - address performance at integration testing, or in response to field problems
    - tighten code, modify design, modify architecture
    - OFTEN TOO LATE
3. ***model the detailed design*** before building
    - simulate the detailed operation;
    - TOO HARD (except perhaps for simulations from a design tool)
4. ***early, high-level modeling*** (THIS COURSE)
    - model the architecture and high-level design
    - part of the analysis process



Goals

Motivations for considering performance during design

- *value:* we want to maximize value: capacity is money
- *risk:* performance failures are a serious risk
    - minimum performance is required to function
- *time:* development time: performance fixes take a lot of time
    - they usually require changes at deep levels of structure
- *entropy:* in evolving a product, creeping features degrade performance

Necessary capabilities

1. Capture software factors and performance parameters
2. Extract predictive models
3. Identify problem spots (diagnosis)
4. Apply patterns or principles for improvement

Examined in 3 stages of increasing complexity

Part A ... Sequential software with flat resources
Part B ... Layered software and resources
Part C ... Systems with internal parallelism

NEXT: Capturing Sequential Behaviour

Use ***Scenarios*** defined for Use Cases.

- define performance measures of interest relative to the scenario:
  - identify ***start and end events*** for responses, with delay requirements or measures
  - identify ***event type***, to define the throughput or jitter requirements/measures for a repetitive stream of events

- model the operations within the scenario:
  - identify operations
  - estimate their workloads (CPU seconds, calls for other operations)
  - identify their components (to localize the operations in the software)

There are many notations for scenarios.....

- to define the sequence or precedence of operations or events, including
  - sequence; OR-fork/joins (or CASE constructs),
  - AND forks and joins (parallel sub-paths)
- for **paths without components:**
  - our *activity graphs,* also “task graphs”, flow charts, Smith’s execution graphs
  - Petri nets
  - use cases, and Use Case Maps without components
  - regular expressions (for fully nested behaviours)
  - process algebras such as LOTOS, CCS, CSP give a structured and combinational view (a process here is an object that generates a set of paths)
  - data flow diagrams (deMarco, Yourdon) sort of qualify
- for **paths going through components**:
  - Use Case Maps with components
  - MSCs (message sequence charts) and UML Sequence Diagrams
  - communicating state machines (many kinds, including StateCharts, ROOM, SDL)

UCM notation for a scenario

Time is consumed in five ways:

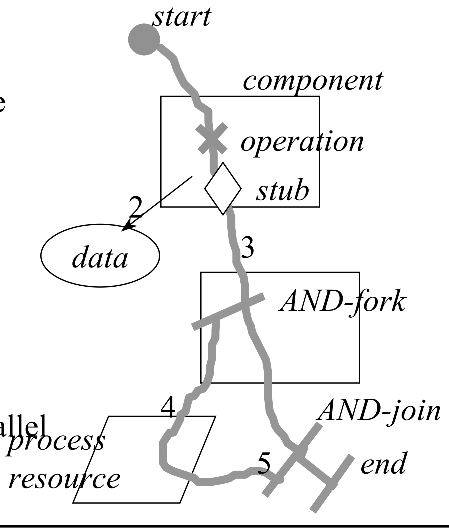
1.. time to do operations (this is the payload)

2.. data access

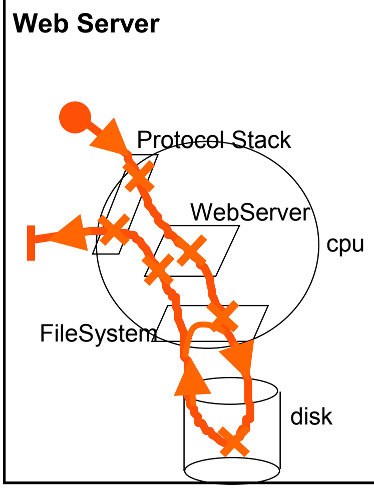
3.. context-switching and communications operations: overhead

4.. resource contention delays

5.. synchronization delays (for parallel operations)



UCM for Web Server Scenario



Performance parameters:

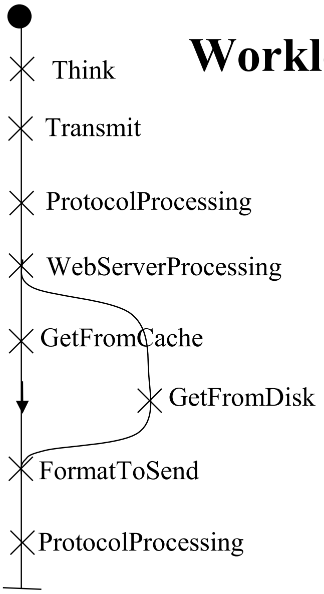
- Workload ***type*** and ***intensity*** parameters can be associated with the start point
  - open, a stream of given rate f
  - closed, a population with a given size N and delay Z between entries
- operation ***demand*** parameters can be associated with a responsibility ✕
  - host (CPU), services
- ***branching and looping*** parameters can be attached to branch or loop points
  - probabilities, loop counts

Analyzing a scenario

- Sequence is given by the scenario model
- Workload parameters must be estimated for the operations
  - maybe the operation needs to be broken down
- a ***workload table*** is useful to capture the makeup of each operation in terms of constituent sub-operations:
  - a device-operation is a basic physical step in execution (CPU-op, disk-op, printer op)
  - a resource-operation is requested from some logical server
  - a functional operation is just a subdivision, perhaps for one procedure call.
- the columns are:

operation [repetition-count] sub-op1 sub-op2 .....

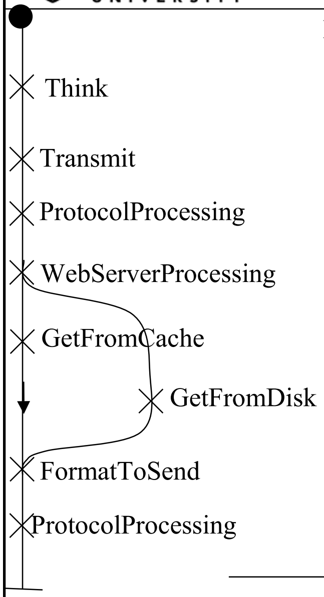
Workload Table Display



- for ease of reference, it is helpful to lay the scenario out down the side of the page, in order from top to bottom
  - with a line for each operation
  - the line is used for parameter data (next)
- if the UCM display is used, the components usually cannot be retained in this display
- A complicated part can be hidden inside a stub

Workload Table with UCM

for the Web Server
(only device-operations in this table)



	RepCount	CPU	DISK	USERS	NET
Think	1			2.5 sec	
Transmit	1				0.150
ProtocolProcessing	1	0.001			
WebServerProcessing	1	0.005			
GetFromCache	0.6	0.002			
GetFromDisk	0.4	0.004	.010		
FormatToSend	1	0.004			
ProtocolProcessing	1	0.001			
	Wtd sum	.0138	0.004	2.5	0.150

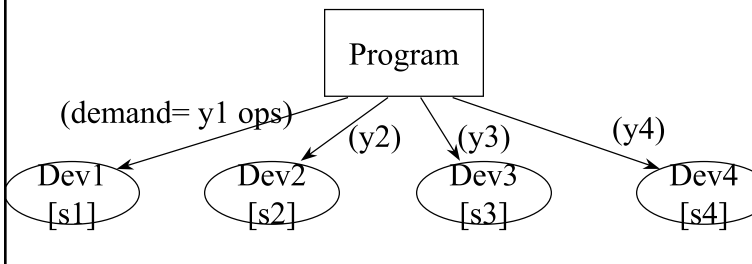
(Demands in sec = Sum of RepCount times OpCount)

NEXT: Basic model-based analysis for "linear" software

- ***sequential programs*** using one device at a time
  - one, or more than one program sharing the system
- no logical resources
- analysis is based on knowing the demands on each device as a series of "service times"
  - seconds for each "service"
  - each service is a single use of the device by the program
- a lot of analysis can be done from the ***total demand per response*** for each device (simpler to record)
  - bottleneck analysis for limits,
  - queueing analysis for delays
- Find the ***total demand per response***
  - by reducing the scenario data (via the workload table)
  - or by reducing layered call-graph data (via a module model)

Demands made by "Linear" software, on flat (device) resources

- a sequential program (one or more copies) running on a system with one or more processors and storage devices
- A program executes and gives a response
- Demands for device operations are per response (AKA visits)



Program

(demand= y1 ops)

(y2)

(y3)

(y4)

Dev1 [s1]

Dev2 [s2]

Dev3 [s3]

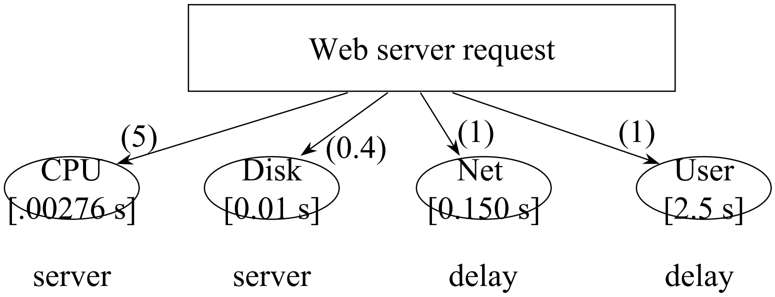
Dev4 [s4]

Note: CPU operations are really small! We may use CPU-sec or M-ops millions of ops)

➡ Demand for device *execution time* Di = yi*si

Flat resource model for the Web Server system

- A response here === a request
- We suppose that the CPU time is divided into five slices of .0138/5 = 0.00276 sec on average

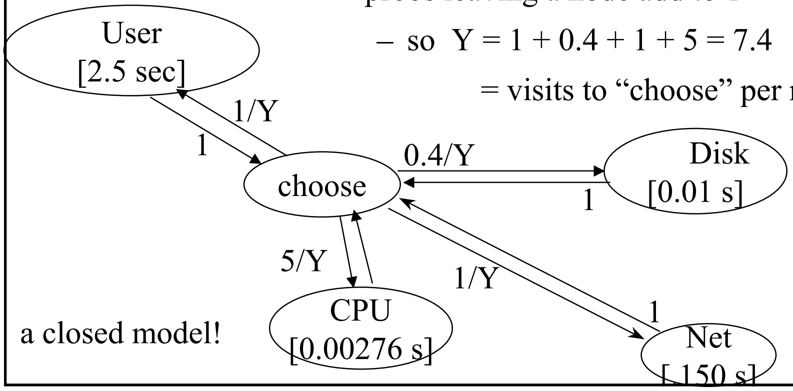


➔Execution demands = 0.0138, 0.004, 0.150, 2.5 sec, respectively
➔bottleneck resource is the disk (biggest demand at a single server)

Queueing network model for the Web Server system .... constructed from the workload demand values

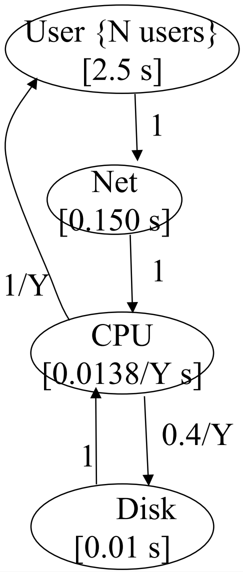
- with probabilities of which resource is used next, on routing arrows
- prob from “choose” is proportional to “y”
- probs leaving a node add to 1
  - so Y = 1 + 0.4 + 1 + 5 = 7.4

    = visits to “choose” per response



a closed model!

Another topology closer to actual server transitions



- customer “token” in turn represents
  - a user thinking,
  - a message passing through the net,
  - an active process
  - a disk operation request
- there are one or two visits to the CPU (say, Y visits on average)

Probs give 1 = (1 + 0.4) / Y --> Y = 1.4

Thus, cpu service time = 0.00986 s

- this is a “closed” model with a fixed number of N users

Queueing model analysis: Bottlenecks

- Bottlenecks: the bottleneck device in a plain queueing model is the device with the largest demand Dmax.
  - further, the system throughput is limited, by the bottleneck, to fmax = 1/Dmax
  - easily calculated from the workload table
  - using profiling measurements also
- familiar example is I/O bound vs CPU-bound operations.
- design strategy outline: reduce the operations at the bottleneck, or divide them among more devices
  - effort to reduce work away from the bottleneck will be wasted, as it does not change Dmax.

About bottlenecks

A bottleneck resource “bn” limits throughput as follows:

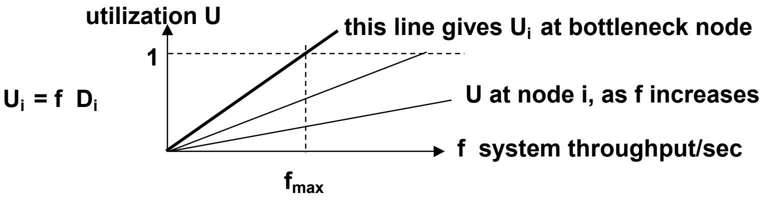
1. The bottleneck resource is the one with the largest demand, say Dmax = Dbn seconds per response
   - Dbn = Vbn * Sbn, where
     - Vbn = resource requests per system response
     - Sbn = resource holding time (service time) per request
2. suppose f = system throughput in responses/sec
   - then utilization Ubn = f * Dbn, Ubn < 1.0
   - at saturation, Ubn = 1, f = fmax = 1/Dbn
3. relieve the bottleneck by reducing Dbn, meaning Vbn or Sbn

---

Some notes on Bounds/Bottlenecks: open system

- Arrivals at a fixed rate f to a set of devices i= 1, 2, 3...
- Node i will saturate at

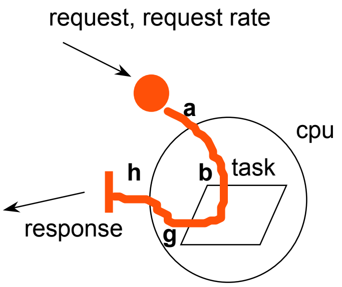
  $f = 1 / D_i$
- The node with the biggest demand $D_i = D_{max}$ saturates first, is the bottleneck node
- Saturation throughput is $f_{max} = 1 / D_{max}$, this is the system saturation capacity
  - practical capacity is much less, determined also by delay



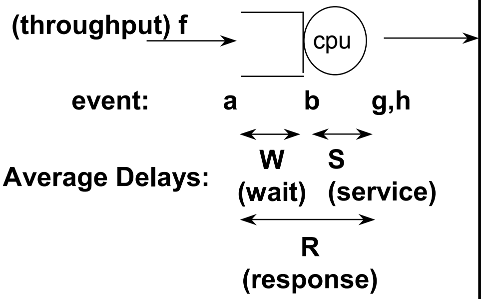
- As f approaches f_{max} then the bottleneck U_i approaches unity, the queue at node i becomes very long, and the mean network delay approaches infinity, dominated by the saturated node.

A queueing model can also estimate resource contention delays

- Consider a name server operating completely out of main memory for speed. This is a single open server. Delay notation…..



Name Server B (from Main Memory)

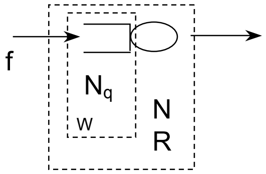


CPU Queue Model

The famous Little's Law

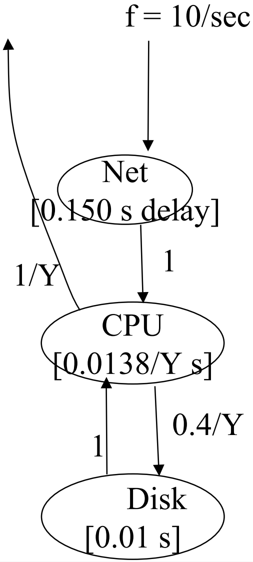
- Relates
  - the mean number N of tokens in any subsystem or other identified state
  - the mean rate f at which they enter and leave that subsystem or state
  - the mean time R they stay in it, on each visit
  - then:

$$N = f\ R$$



- or, if the subsystem is the queue alone, with $N_q =$ the mean number waiting but not in service, then $N_q = f\ W$
- if it is the server, the $U = f\ S$
- this is exactly true for averages over any period if the subsystem is empty at the beginning and end. It requires no assumptions.
- if a steady state exists it is also true for theoretical mean values

Open Web Server Model



- Dmax = CPU demand at 0.0138 sec/response
  - only considers the servers CPU and Disk
  - delay at Net cannot limit throughput
- a flow of arrivals at a stated rate f such as 10 per sec.
  - the user think time does not enter into the system performance
- the system may be unstable if arrivals are too fast (here, if f > (1/0.0138)=72.46 per sec)
- analyze to find the response time from entry to exit
  - R = sum of mean delays at nodes

Closed systems

- A finite pool of users, as in the closed Web Server
  - periods between finishing one response and requesting the next one are called “think times” or user latency
- longer delays slow down the users and reduce the throughput… so throughput is not a given, but is determined by the system
  - productivity
  - data entry workers going as fast as they can
- few users or long user latencies can starve the system, which bounds the throughput at the source
  - “path length” bound

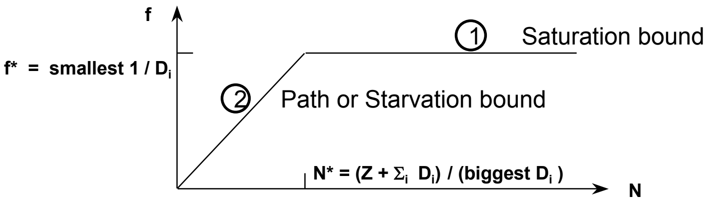
A second kind of limit for closed systems… path length bounds for input starvation

- in a closed system with just a few users there may not be enough load to saturate any resource.
    - consider N users in one class...
- the response delay with small contention is dominated by the operations along the path, so **R = Σi Di + a bit of waiting**
- ignoring the waiting, we can say that **R > Σi Di**
- We can also identify a user *cycle* as **Cycle = (user thinking Z) + R**
- Then Little’s result says that **f = N / Cycle** always,
    - so...... **f < N / [Z + Σi Di]**
    - Z is defined as (user latency) or, more generally, as the sum of any pure delays in the system
- so, small N or large Z constrains f... “starvation” by path bound

---

Throughput bounds, one-class closed system

- Service center demand D_i per response (exclude infinite servers)
- Total infinite-server pure delay of Z / response
- A saturation-based bound at each resource (except infinite servers) is:
    - at resource i, utilization < 1.0 means: $f_i < 1 / S_i$
    - This constrains f: $f < 1 / (V_i S_i) = 1 / D_i$ ① $< 1 / D_{max}$
    - This is the same bound as we had for open systems
- A path-length bound for the system as a whole is:
    - the response cycle must have: $C > Z + \Sigma_i D_i$
    - thus by Little's result: $f = N / C < N / (Z + \Sigma_i D_i)$ ②
    - this part is only for closed systems
- The classic throughput bound as a function of N has this shape:



Closed Web Server Bounds

- Bound is a function of number of users if this number is small (starvation)
- Z = sum of delays at delay servers = 2.5 + 0.15 = 2.65
- D = 0.0138 at CPU

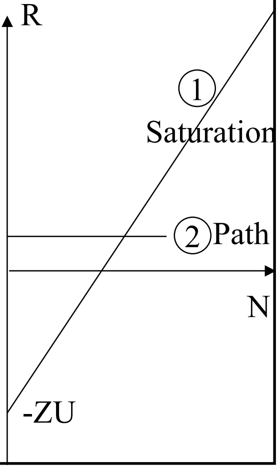
  = 0.004 at Disk

  So Dmax = 0.0138, Saturation bound is fmax = 72.46 requests/sec
  - For the mean response time R seen by the users,

    Little gives N = f*(2.5 + R) --> R > (N/72.46) - 2.5
- Starvation bound is f < N/(Z + Σ D) = N/(2.6678) = 0.3748 N
  - For R, N = 0.3748 N (2.5 + R) --> R > .0178 (= Σ D)

- Break at N* = 2.6678 * 72.46 = 193.3 users

Response time bounds, closed system



- C = mean time for a cycle around the closed system

  C = N/f sec (by Little)
- R = mean response time relative to a “user” state with mean think time ZU

  R = (C - ZU) sec
- Upper bound on f gives a lower bound on C:

  Cmin = N / fmax
- So, saturation bound fmax = 1/Dmax gives

  Cmin = N * Dmax

  Rmin = N * Dmax - ZU ①
- And starvation bound fmax = N/(Z + Σ D) gives Cmin = Z + Σ D

  Rmin = (Z - ZU) + Σ D ②

Sensitivity

- Performance is mostly sensitive to demands at the bottleneck resource
- D = V * S = visits time service time

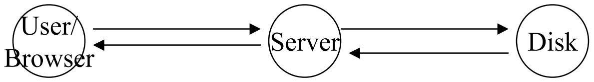
- To improve performance, reduce V or S at the bottleneck resource
  - reason about how this resource gets loaded!!!
  - reduce V by doing fewer operations at this device (e.g. more successful caching at web server... if disk were bottleneck)
  - reduce S by tightening code, or calling procedures less often

- Sensitivity to other demands is ***small*** in starvation (underloaded)
  - and ***zero in saturation***

Adding resources at the bottleneck

- Is good
- Multiple servers (M): saturation becomes fmax = Dmax/M
- Sharing load among M servers equally, same
- Faster server reduces Dmax

Bottleneck Sensitivity: Another Web Server



N users, (Z = 5 sec) Ds = 8 ms Vd = 2.5, Sd = 5 ms

- Web server runs a script to create dynamic pages, based on data retrieved from tables stored on disk
- Bottleneck for large N is the disk (Dd = 12.5 ms, fmax = 80/sec at Users)
  - small N: $f < N/(Z + \Sigma D) = N/5.0205$ per sec (UNITS)
- Crossover is at N/(5.0205) = 80, or N = 402 approx

- Response Cycle (user, round trip): C > 5.025 and C > .0125 N sec

- Response time, excluding Z : R > 0.025 and R > (0.0125 N) – 5 sec

---

Sensitivity (cont’d)...

- Consider N = 500.... f = 80/sec, mean response time is
      - R = (500/80) – 5.00 = 1.25 sec.
  - Non-bottleneck: 10% decrease in Ds gives no change
  - Bottleneck:10% decrease in Dd gives f = 88.8/sec, R = 0.68 sec.
  - **Throughputs at various N:**

N	100	200	300	400	500
Base f	19.9	39.7	59.3	76.7	80.0 /sec
if Ds = 7.2 ms	unchanged.......................................				
if Dd = 11.25	19.9	39.7	59.4	**78.3**	**88.8** /sec

Web Server: Changes to the CGI Script

1. “hot spot”... suppose there is a frequently-accessed index page which changes only very slowly: create it and keep it ready

2. binding/fixing-point: design the dynamic page so it has a fixed format, and just fill the fields

3. locality: ...make the tables memory resident, or
   ...rearrange the tables to reduce the number of disk accesses

4. total cost: compare the preformat in 2, to the memory resident structure in 3

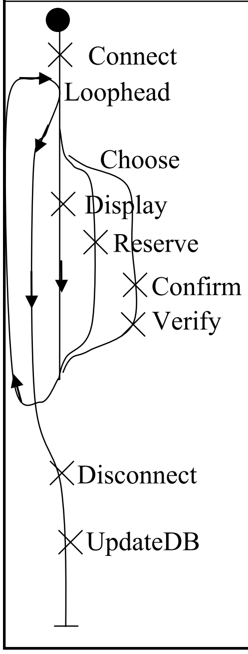
Changes which impact the disk (Dd) will be more effective!!

NEXT: Workload parameters for a more complex case: Ticket Reservations

- A web server as before, but with more complex software design at the server
  - different kinds of pages
  - CGI operations, dynamic pages, database interactions

- This time we will consider an ***unbound UCM***, a scenario with operations defined but no software components.
  - the operations have demands directly on the devices
  - this approach was introduced by Smith in her 1990 book, she called her scenarios “execution graphs”.

An *Unbound* UCM for Ticket Reservations: a pure scenario

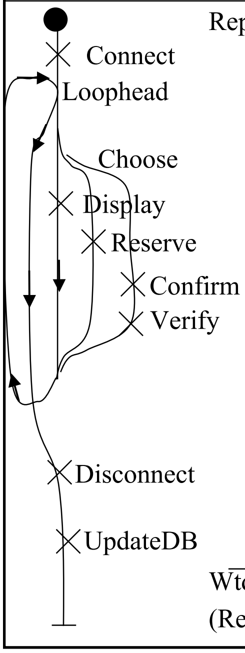


- Connect: client connects to server
- Loophead: loop overhead for repeated interactions
- Choose: decide on the type of interaction
- Display: display the shows and seats
- Reserve: choose tickets and signal to buy
- Confirm: take a confirmation and credit card data
- Verify: verify credit card with the bank to clinch the sale, update the ticket database to show the tickets as sold.
- Disconnect: process a connection shutdown (explicit or timeout... here assumed to be explicit)
- UpdateDB: update data base with sales and marketing data for later analysis

---

Workload Table with UCM

for an e-commerce Server (for Ticket Reservations)



	RepCount	cpu-op	db-op	com-op	CCreq-op	
Connect	1	0.01		1		
Loophead	6	0.001				
Choose	6	0.001				
Display	4.5	0.005	1	1		
Reserve	0.9	0.015	2	1		
Confirm	0.6	0.002		1		
Verify	0.6	0.004	1		1	
Disconnect	1	0.001		1		
UpdateDB	1	0.007	1			
	Wtd sum	0.0696	7.9	8	0.6	= Entire demands

(RepCount times OpCount

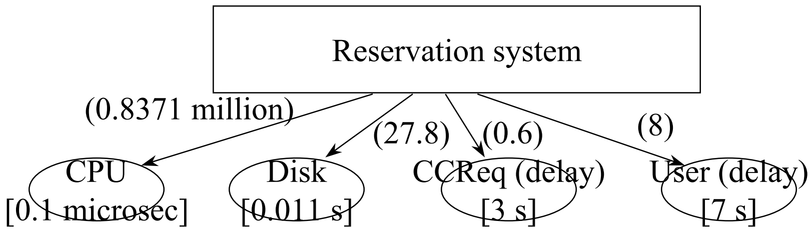
Workload Table Reduction to Total Demands on Devices

(demand in seconds per session)

	cpu	disk	db	comms	CCreq	User	
Wtd sum (repeated)	0.0696		7.9	8	0.6		
Logical Services to be eliminated							
db op	0.085	2					
comms	0.012	1.5				1	
External demands							
in wtd sum	0.0696				0.6		
for db	0.6715	15.8					
for comms	0.096	12				8	
sum	0. 8371	27.8			0.6	8	
Operation time	0.1	0.011			3	7	sec/operation
Demands D	0.08371	0.306			1.8	56	sec/session

Flat resource model for the reservation system

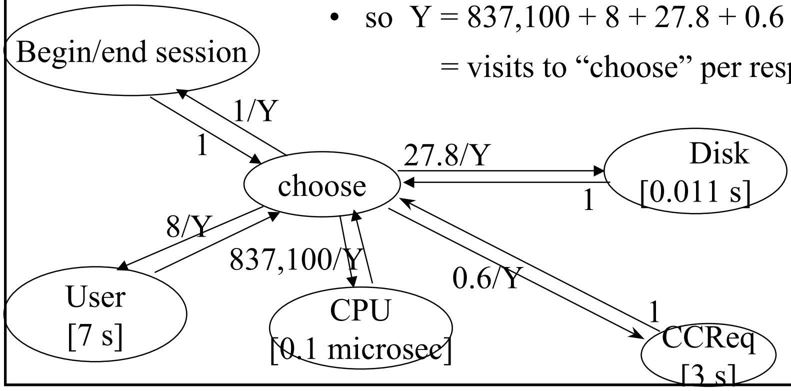
- A response here === an entire session, including connect and disconnect
- Database is on the same CPU as the web server
- CCReq and User are infinite servers
- CPU operation is one processor operation, at 10 million per second... One way (as here) is to show the demand in operations, consistent with other devices.



➔Execution demands = 0.08371, 0.306, 1.8, 56 sec, respectively
➔bottleneck resource is the disk (biggest demand at a single server)

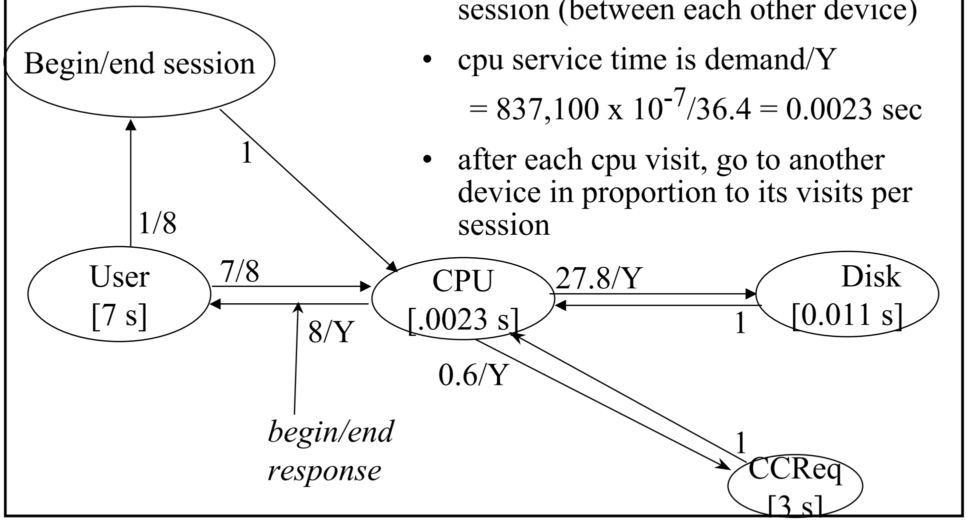
Queueing network model for the reservation system

- with probabilities of which resource is used next, on routing arrows
- CPU is modeled as having one visit per op’n
- probs leaving a node add to 1
- so Y = 837,100 + 8 + 27.8 + 0.6 + 1
  = visits to “choose” per response



More conventional queueing network model for the reservation system

- Y = 27.8 + 0.6 + 8 cpu visits per session (between each other device)
- cpu service time is demand/Y
  = 837,100 x 10^{-7}/36.4 = 0.0023 sec
- after each cpu visit, go to another device in proportion to its visits per session



Asymptotic bounds for the Reservations System

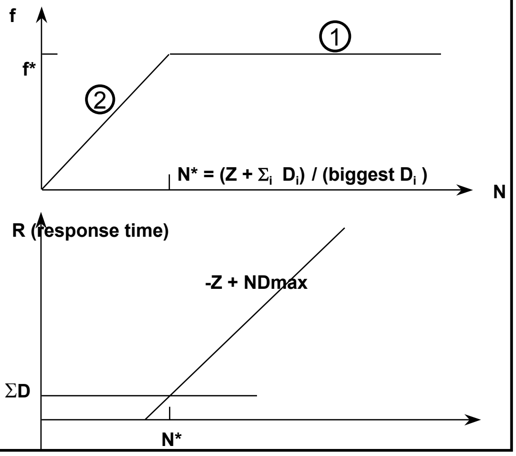
- Throughputs and responses are referenced to the “end of session” node...
- Dmax = 0.306
- f* = 1/0.306

  = 3.27 responses/sec
- Z = 56 + 1.8 = 57.8
- ΣD = 0.08371 + 0.306

  = 0.39331

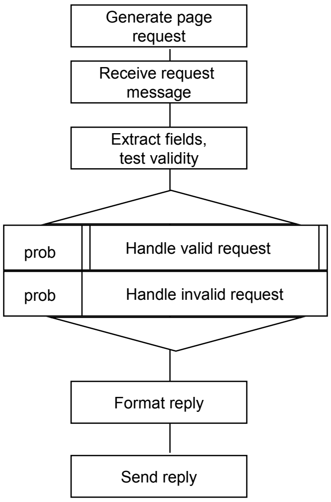
(sum of finite-server demands)

- N* = (57.8 + 0.39331)/0.306



Scenario definition also by Activity graphs:

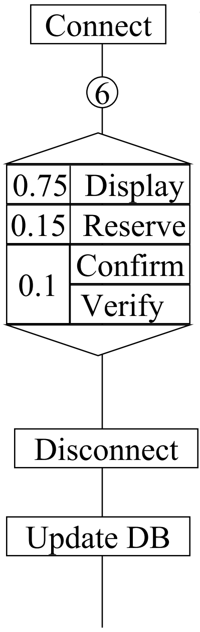
Web server “Simple Page Retrieval”.....



Workload parameter Comments	**Graph Notation Comments**
•workstation	
•comm I/F, CPU	•standard activity
• CPU	
• Case test Overhead	• “Case” structure, showing overheads and the activities included
• Parameters derived from subgraph	• each activity has a probability
• CPU	• “Handle valid request” is a high-level activity which has a subgraph to define it (see next slide)
• CPU	• a Case structure can have probabilities that add up to less than 1.0, meaning there is some probabiity of doing nothing
• CPU, comm I/F	

Activity Graph for Ticket Reservations

- Web browser interacting with a ticket info CGI program on the web server
- CGI program accesses a ticket status database
- Confirmation of sale uses an interaction with the credi-card company to validate the card information.
- end-to-end scenario
- a notation designed for reducing data on demands


Connect
⑥
0.75 Display
0.15 Reserve
0.1 Confirm
Verify
Disconnect
Update DB

Comments on activities
•connect to client

•loophead: loop for repeated interactions
•selection of type of interaction
•display shows/seats
•reserve seats
•confirm the sale of the seats
•get and verify credit card data to clinch the sale

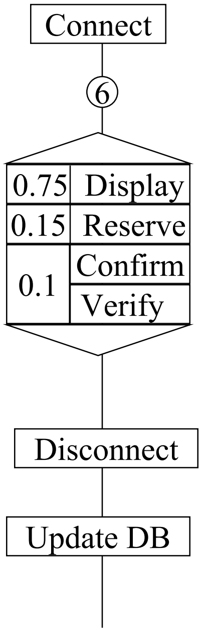
•client has chosen to disconnect

•update data base with sales and marketing data for later analysis

---

Workload Table with Activity Graph

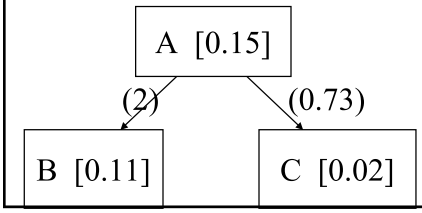
for the same e-commerce Server (for Ticket Reservations)

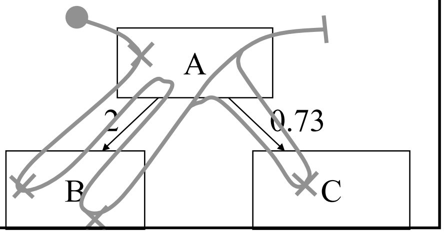

Connect
⑥
0.75 Display
0.15 Reserve
0.1 Confirm
Verify
Disconnect
Update DB

MeanTimes	cpu	db	comms	CCreq	
1	0.01		1		
6	0.001				
6	0.001				
4.5	0.005	1	1		
0.9	0.015	2	1		
0.6	0.002		1		
0.6	0.004	1		1	
1	0.001		1		
1	0.007	1			
Wtd sum	0.0696	7.9	8	0.6	“Entire demands”

NEXT: Scenarios defined by Component Interactions

- For sequential systems in which modules call each other, a scenario may be defined by a call
  - each call has an implicit response time from call to return
  - the call includes an amount of CPU demand (call it ***host demand***)
  - the scenario of the call includes nested calls
    - use a call graph, with arcs (arrows) for calls
    - we can describe multiple calls by putting a parameter on the call arrow.





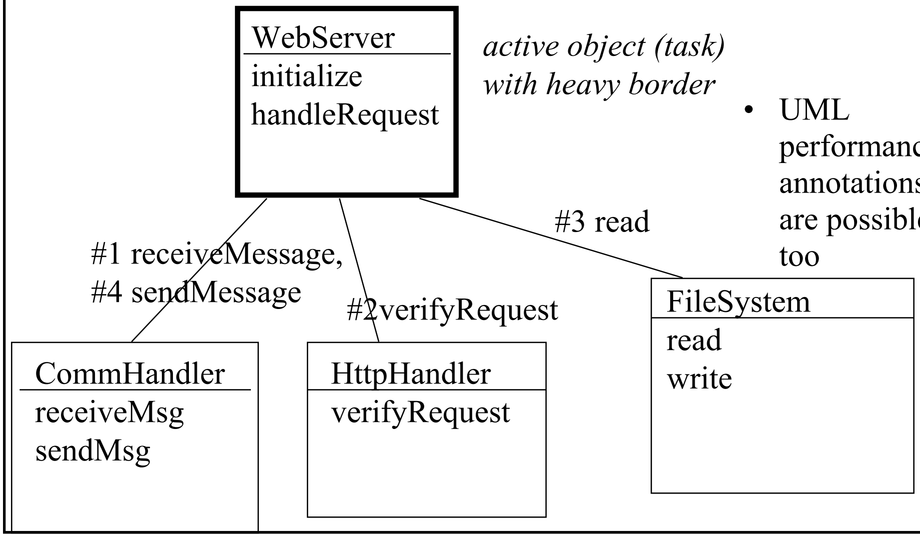
Workload Table for Component Interactions

component	CPU (sec)	Disk (ops)	Printer (ops)	i/o (ops)	print (ops)	CallB (ops)	CallC (ops)
A	0.15					2	0.73
B	0.11			3.0			
C	0.02				1		
i/o (suppose)	0.007	1.3					
print (“)	0.01		1				
Reduction:							
Btotal	0.121	3.9	0				
Ctotal	0.03		1				
Atotal	0.4066	7.8	0.73				
service times	1	0.011	0.0825				
Demands (sec)	0.4066	0.0858	0.060225	(= product of 2 lines above)			

Note...It is the same as the table for a scenario description, with abstract operations for the calls

Performance and component structure: Coordination diagram in UML

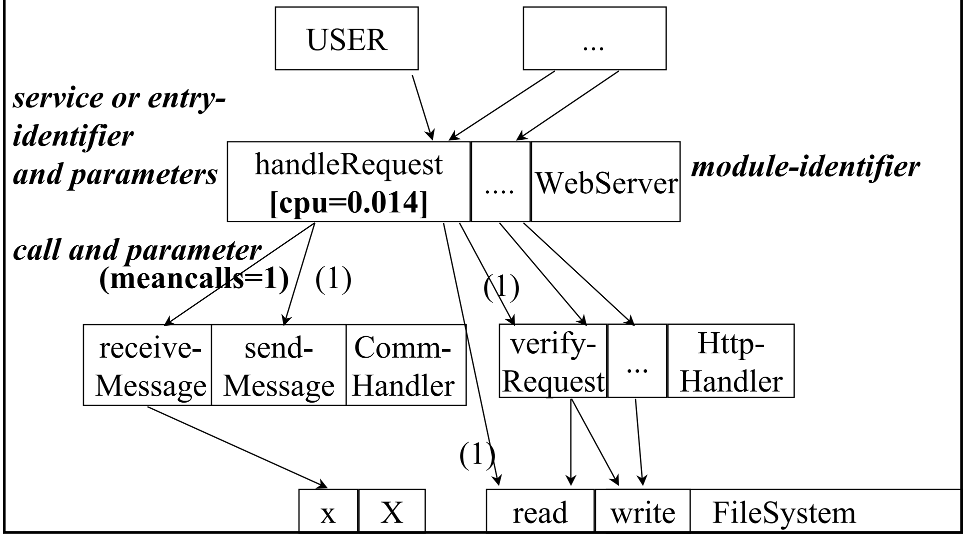
(scenario fragment indicated over objects)



active object (task) with heavy border

- UML performance annotations are possible too

Layered model notation shows the calls to *services*, and the performance parameters



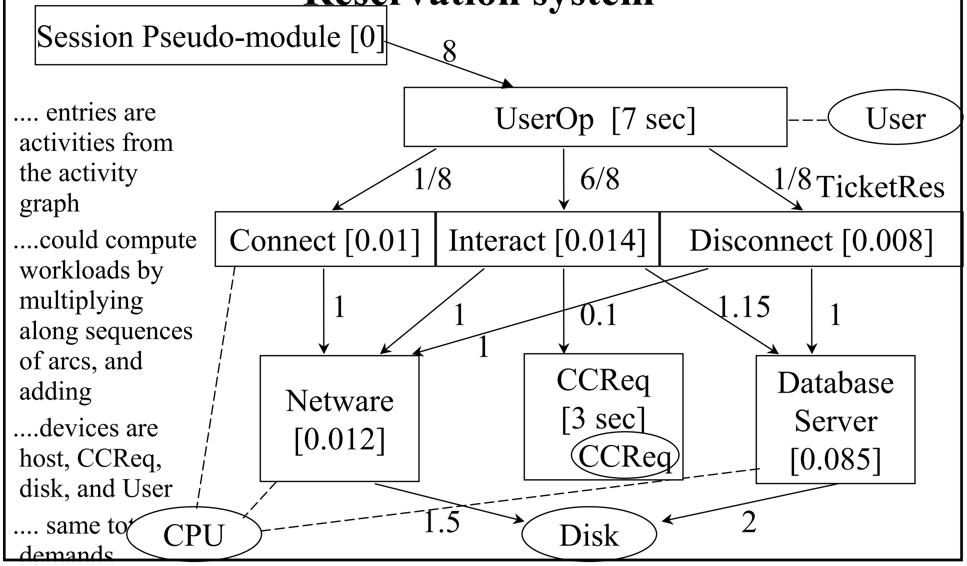
Component interaction diagrams express the software Architecture

There are several different ways to express architecture

- Architecture Description Languages (ADLs)
- many use some kind of box-connector diagrams
  - UML stereotypes are developed for this, in “Applied Software Architecture” by Hofmeister, Soni and Nord
    - using a “protocol” across the connector to define the message types exchanged
- architecture languages, including structured description of inclusion
- some, like ROOM, include behaviour definition by state machines

---

Demand analysis from a layered module model: Reservation system



.... entries are activities from the activity graph

....could compute workloads by multiplying along sequences of arcs, and adding

....devices are host, CCReq, disk, and User

.... same to demands

Summary of the Modeling Framework so far

- ✓ Model
- ✓ Approach to parameters through scenarios
- ✓ first diagnostic idea: reduce the demands at bottleneck resources

Still to consider

- choosing the parameter values
- basing the framework on design notations in UML, SDL, etc
- modeling multiple programs competing for a system
- software structure effects:
  - ***logical resources and parallelism***

Getting the Parameter Values

Three approaches:

- measure the operations, preferably at a high level
  - benchmark them
  - explore the key parameters, eg dependence of “sort” on size of list
  - especially suitable for re-use of components in a new system
- estimate values, based on experience
  - Smith suggests a group consensus approach
  - estimate best case, worst case
- budget an allowance for each operation, that will be used as a guide by the developer
  - can exploit experience
  - also suitable for new software (better to budget than not to think about it at all)

Getting the Parameter Values (2)

- This is a serious difficulty for some groups... no one likes to admit ignorance, no one likes to guess
- a budgeting approach is rational however...
  - money budgets are normally estimates of unknown expenditures
  - budgets can be adjusted.
- When in grave doubt, study the effect of a range of possible values (***sensitivity analysis***)
  - if the value doesn't matter then forget it, use any value
  - find the parameters that do matter and study them more.

Getting the Parameter Values (3)

- As a student... pick a number and focus on studying the techniques. These are realistic but rough values:
  - 100 microsec for a process switch
  - 300 microsec to send a packet
  - 1 ms for a small operation,
  - 2 -5 ms for a disk operation (retrieve an 8K disk block)
  - 10 ms for a moderate operation, or for a small Java operation

- For the course... don't hesitate over parameter values, it isnt the focus of the study
- Later, for any real program, you can try both cautious and bold approaches. Always try some measurements if you can.

Getting the Parameter Values (4)

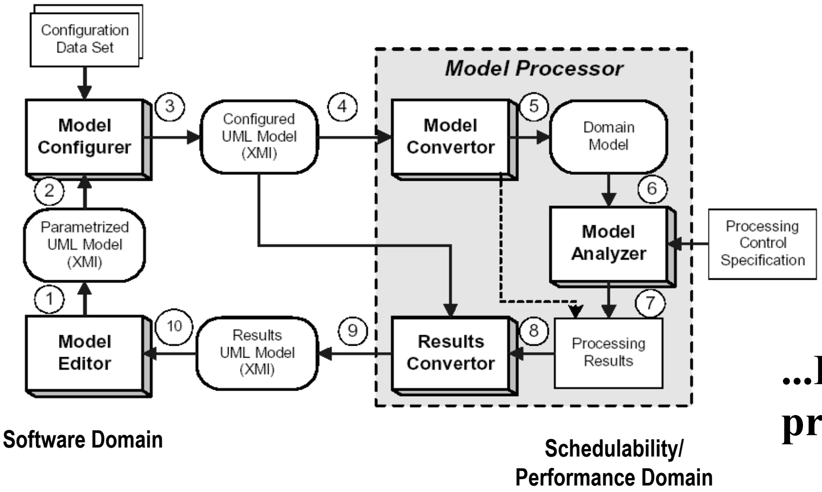
Measurement Techniques

- profiling (gprof in unix) (java profilers)
  - gives time in each procedure
- end-to-end time (time command in unix, or repetitive running with wall-clock time)
  - benchmarks are of this type
- event-to-event delay from probes in the code (record time of event, store in trace file, postprocess for delay)
  - watch out for the coarse granularity of the clock e.g. in Unix)

- run the code on a simulator of the processor, and monitor the instructions it executes.... elaborate.

Basing Analysis on Standard Design Notations

- leverage design tools, and design documents produced in early stages
  - limit the extra work to create the performance model



...PUMA project

UML with Performance Profile

(details later)

- Performance profile is on the course web site
- It adds performance data as ***annotations*** to behaviour diagrams in UML, which describe scenarios
  - performance requirements and placeholders for results
  - workload intensity and type
  - choice probabilities
  - operation demands
- ***Stereotypes and tagged values*** for time, resources, and parameters in the Sequence, Activity, Collaboration and Deployment diagrams
- Two sub-profiles, for schedulability of deterministic systems, and for performance of non-deterministic systems

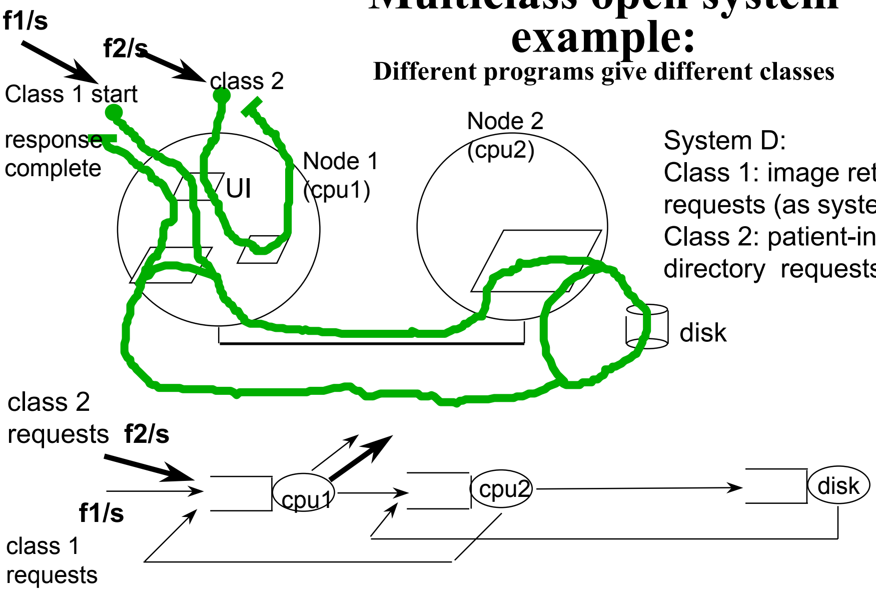
---

NEXT: Systems with Multiple sequential applications

- separate demand analysis for each application
- queueing analysis calls each application a "***class***" c of work
- ***Class c*** has its own:
  - **throughput f(c)** in responses/sec
  - **demand D(i,c)** at resource i, in sec/response
    - **D(i,c) = V(i,c) S(i,c)**
      - where V is visits/response, found for each class (V or y equally)
      - S is seconds/visit
  - waiting W(i,c) at node i
  - **utilization U(i,c)** at node i
- each class may have a separate ***reference node*** for throughputs and responses
  - (a response is a cycle from the reference node, back to it again)
- Bottlenecks: sometimes ***more than one resource can be bottlenecked***

Multiclass open system example:

Different programs give different classes



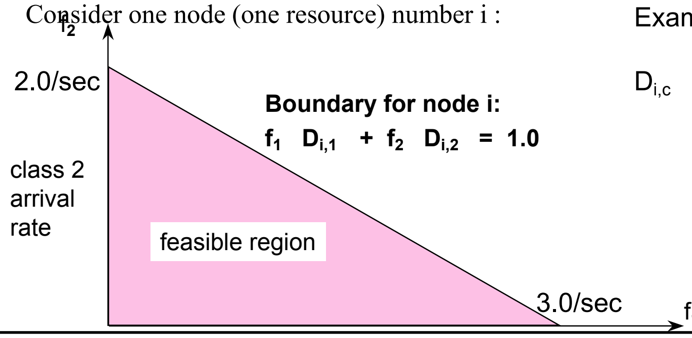
System D:
Class 1: image retrieval requests (as system C)
Class 2: patient-info directory requests

Visualising open two-class system saturation

- saturation at node i requires: $\Sigma_c \; f_c \; D_{i,c} \; < \; 1.0$ at service center i
- For two classes a plot of the feasible (stable) throughputs (f_1 , f_2) can be made. Each center i gives a *boundary* which is a straight line on the plot
- Consider one node (one resource) number i :

Example values:

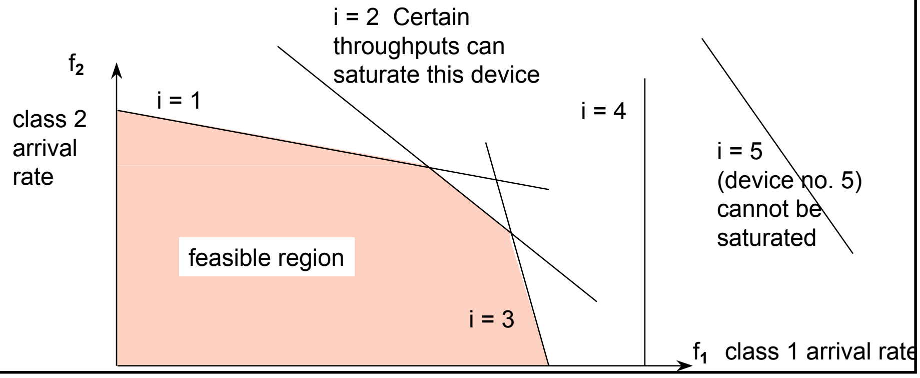
	class 1	class 2
$D_{i,c}$	0.333	0.5



Visualising open two-class system saturation: more nodes

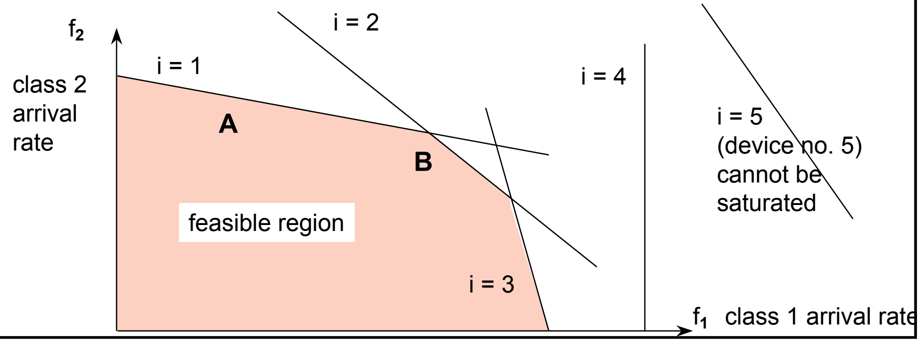
Each service center gives one saturation condition; all boundaries must be satisfied for the system to be stable

For two classes and many centers a plot of the feasible (stable) throughputs (f_1 , f_2) is shown:

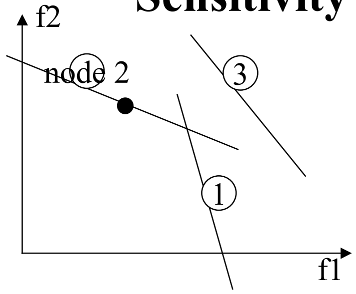


Bottleneck location in open systems

- if the joint throughputs approach any boundary, that node is becoming saturated, and it will have a long queue!
- near A, device 1 is becoming a bottleneck; near B, there are two bottlenecks (devices 1 and 2). Device 4 or 5 *cannot* be a bottleneck

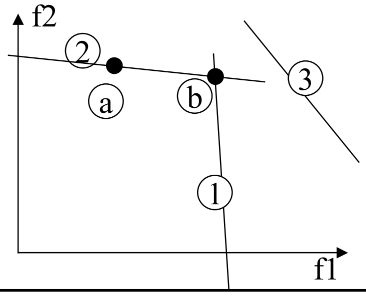


Sensitivity in two-class systems



Upper:

- High sensitivity to demands at node 2, low or none at nodes 1 and 3,
- Away from bounds, all parameters are slightly sensitive



Lower:

- At a, class 2 is sensitive to node 2, but class 1 is not (very).
- At b, class 1 is sensitive to node 1, class 2 to node 2

More than two classes

- Visualization doesn’t work on a two-dimensional page!
- Relationships are still true...
  - solve the system, find the saturated resources
  - a class may visit a saturated node, yet not be bottlenecked there, if its demands on that node are quite small
    - this can be seen when the partial utilization Uic is very small

Exercise on visualizing saturation with two open classes

- A system has four devices and two classes, with demand values ($D_{i,1}$ $D_{i,2}$) being: (0.1, 0.5) (0.4, 0.4) (0.2, 0.1) (0.6, 0.001) in seconds.
  - sketch the feasible space of throughputs of the two classes
  - which devices can be saturated?
  - is a throughput of 3 reponses/sec for each class at once feasible?
- In Example D suppose the service times are:

	cpu1	cpu2	disk
class 1	32 ms	55ms	17 ms
class 2	40 ms	0	0

and the probability of a class-1 response being completed on leaving cpu 1 is 0.1 (while class 2 always completes)

  - what are the visit ratios of class 1? the demands?

Full Notation for Multiclass open network models

- Jobs are tokens of different classes, numbered c = 1,2,3...
- Let node 1 be the entry node, so class c has throughput rate $f_c = f_{1c}$ at node 1 and rate f_{ic} at node i
- A token of class c visits resource i an average of $V_{i,c}$ times, (its *visit ratio*). The visit ratio for the entry node is 1.0. Note that $f_{i,c} = V_{i,c}\ f_c$
- The *service demand* at resource i per class-c response is
  - $D_{i,c} = V_{i,c}\ S_{i,c}$.
- The *routing probability* of a token going to device (node) i after j is Θ_{ji}^c . Then we can find the V's from the Θ's by solving
  - $V_{1,c} = 1$; $V_{i,c} = \Sigma_j \Theta_{ji}^c\ V_{j,c}$ i = 2,... .
- The utilization of service center i is the sum of the partial utilizations of the classes,
  - $U_i = \Sigma_c\ f_{i,c}\ S_{i,c} = \Sigma_c\ f_c\ D_{i,c}$ which must be less than 1.0

Full Notation for closed Multi-class systems
(similar to the open case)

- Again, the tokens belong to different classes, numbered c = 1,2,3... Class c has arrival rate $f_c = f_{1c}$ at the reference center node 1 and rate f_{ic} at node i
- The routing probability from station i to j is again $\Theta_{i,j}^c$ which depends on the class.
- A token of class c visits center i an average of $V_{i,c}$ times in each response (visit ratio). The visit ratio for the reference center is 1.0, and $X_{i,c} = V_{i,c}\ f_c$
- The *service demand* at resource i , per class-c response is
  - $D_{i,c} = V_{i,c}\ S_{i,c}$.
- As before, we find the V's from the Θ's by solving
  - $V_{1,c} = 1$; $V_{i,c} = \Sigma_j\ \Theta_{ji}^c\ V_{j,c}$ $i = 1, 2, ...$

NEXT: Guidance for improvement of “linear” software

- Demand and bottleneck analysis:
  - address the bottleneck
- Queueing Model (based on the demand parameters):
  - same, but delays are more realistic (not just bounds)
  - gives utilization information for many classes (visualization breaks down)
- Back to the Activity model:
  - choose *strategic changes*
    - frequently executed operations are visible
    - expensive operations, especially those using the bottleneck device
    - chains of causality that create *unexpectedly* frequent operations
    - use frequency times demand to choose activities to address
  - try to enhance cache effectiveness
  - may also push work off the sequential path, into parallel operations

Software Optimization: Making activities cheaper

- the most basic optimization is to reduce the cost of an operation. Consider:
- assembly-language coding, “code straightening” for locality
- EARLY BINDING
  - macros and in-line procedures, loop unrolling, inheritance flattening, eliminate pointers
- DATA STRUCTURE effects
  - reduced structure traversals, data alignment
- ALGORITHMS
  - search, hashing.....
- Ref... Programming Pearls, by Jon Bentley
- WHEN IS IT WORTHWHILE?: use the activity graph

Load/frequency Improvements: “Do it less”

- Find operations that are done a LOT
- Reduce them by:
  - batching up data on many operations into a single group and doing the operation once
  - similarly, working on larger units of data (such as, a whole message insted of just one character)
  - elimination: do the operation only when necessary (the basis of lazy evaluation)

Early binding (fixing-point)

- many many ways to do this
  - avoid run-time decisions... compile them in
  - fixed memory allocation, pre-allocated record and field sizes
  - compiled code instead of interpreted
  - unrolled loops
  - flat inheritance hierarchy

Locality

- avoid boundaries
  - put data with operations
  - put operations into the same thread or process, on the same node
  - batch operations together
  - increase the granule size for operations
  - physical storage in consecutive memory
  - fewer processes, less communications

Hot spots

- A hot spot is a strategically important set of activities
  - they hit the bottleneck
  - they have tight performance requirements
  - frequent operation
- hand optimize
  - design the data to favour these activities
  - assembly code or code straightening (improve locality for cache)
  - flatten object heirarchy here
  - pre-compute or pre-fetch some data
  - batch operations
- try a fast path or optimistic structure

Fast Path pattern (cheap special cases)

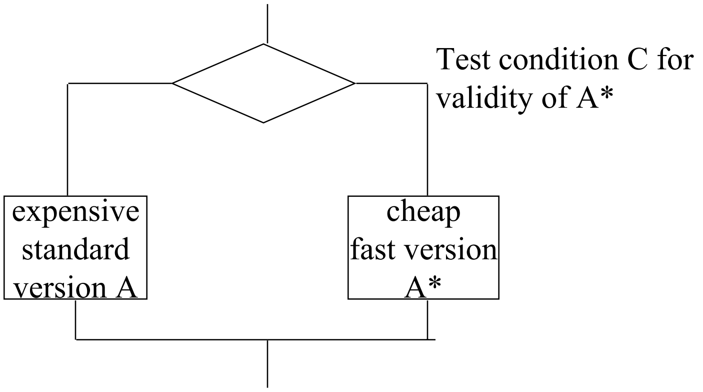
- suppose there is
  - an expensive activity A (with demand $Y_{host}(A)$ to device *host*)
  - a condition C with fairly high probability p*, in which cheap version A* can be used (with demand $Y_{host}(A^*)$)
  - demand of testing C is $Y_{host}(C)$
- for A substitute:
  ```
  SwitchBegin...Test C {HOST Yhost (C) }
  CaseBegin (If C , prob p*)        do A* {HOST Yhost (A*)}
  CaseBegin (Else, prob (1 - p*))      do A {HOST Yhost (A) }
  Switchend
  ```
- replace cost $Y_{host}(A)$ with

  $$Y_{host}(C) + p^* Y_{host}(A^*) + (1\text{-}p^*)\, Y_{host}(A)$$
- better if

  $$Ratio = [Y_{host}(C) + p^* Y_{host}(A^*) + (1\text{-}p^*)\, Y_{host}(A)]/\, Y_{host}(A) < 1$$

  or if $$Y_{host}(C) + p^* Y_{host}(A^*) < p^* Y_{host}(A)$$

Fast path pattern: substitute for any activity A


Test condition C for validity of A*

expensive standard version A

cheap fast version A*

Fast Paths

- examples:
  - avoid protocol encoding for a message on a socket connection to a process on the same computer
  - in remote procedure calls, avoid data encoding when the data representation is identical at both client and server
  - fast access to usual next operation
  - pre-compute frequently-used results so they are always up to date
  - compile some frequently-used operations in an interpreted language
  - special storage of frequently accessed data pages

Fast Path: Protocol Header Prediction

- optimizes handling of long messages.
  - Originally developed by Van Jacobson for paging transfers from a network file system (IEEE Communications Mag about June 91)
- assume the next packet is the next in the same message, calculate the header expected.
  - use the paging memory space as a data buffer, copy the message dirrectly to it
  - check the header against prediction. If it matches, data is already in place.
- greatly reduced processing on the fast path
  - one connection was able to run near 10 Mbit/sec.
- If no match, process it for whatever connection, socket etc. it is directed to, and copy it again as needed.