HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

Internet Content Distribution

Chapter 3: Client-Side and Proxy Caching

Jussi Kangasharju

Chapter Outline

- Client-side techniques
- Caching basics
  - Replacement policies
- Browser caching
- Freshness
- Proxies
  - Proxy discovery
  - Proxy operation
- Proxy caches
  - Replacement policies

Client-Side Techniques

- Client-side techniques are implemented on the client's side as opposed to the server
- Two possibilities:
  - Directly in the user's computer (e.g., browser)
  - Implemented by the user's ISP
- All currently implemented techniques are based on caching of content
- Difference is where caching takes place
  - One of the above two possibilities
- Remember: *Caching is only a performance-enhancing technique. It's never required for correctness*
  - But sometimes caching can cause correctness problems :-(

Basic Idea of Caching

- Cache is: (Wikipedia)
  A collection of data duplicating original values stored elsewhere or computed earlier, where the original data is expensive (usually in terms of access time) to fetch or compute relative to reading the cache
- Once data is cached, it can be fetched from the cache
- Benefit: Faster average access time to data
- Caching is widely used in computing
  - CPU caches
  - Virtual memory
  - Hard disks
  - Web browsers

Building Blocks

- Two parts to a caching system:
  - Permanent storage
  - Cache
- All data is always available in permanent storage
  - For us: origin server of content provider
- Some data is available in cache
  - Data in cache can be accessed very fast (relatively speaking)
- Client wishing to access data in permanent storage first sends its request to the cache
- If requested data is in cache, it is delivered from cache
  - Called cache hit
- If data is not in cache, it is fetched from permanent storage
  - Called cache miss

Hits and Misses

- Ratio of cache hits to total number of requests is called the hit rate (hit ratio) of the cache
- Corresponding definition of miss rate (also 1 - hit rate)
- When a miss happens, data is fetched from permanent storage and *usually* put into the cache
  - Note: Some caching schemes do not admit all data into cache
- If new data is inserted into a full cache, some cached data must be thrown out (replaced)
  - Decided by cache replacement policy
- If data in permanent storage changes, cached copy becomes stale
- If data in cache is modified, it must be written back to permanent storage
  - Not a concern in web caching, big issue in CPU or hard disk

Replacement Policies

- Also called replacement algorithms
- Tons of research in this area, both theoretical and practical and in many topics
- Problem statement: *Cache is full and we want to bring in a new item. Which item gets evicted?*
- What is the metric we use for comparing algorithms?
- Traditionally, hit rate is used as a metric
  - Traditionally = virtual memory page replacement, CPU cache, hard disk caches, …
- Web caching also uses:
  - Byte hit rate
  - Access time

Caching Metrics

1. Hit rate
   - Ratio of cache hits to total number of requests
2. Byte hit rate
   - Ratio of data delivered from cache to total data requested
   - In other words, hits are weighted with the data sizes
3. Access time
   - Average access time to data

- Why "traditional" caching only considers hit rate?

- In "traditional" caching, all objects are same size
  - For example, memory pages
- Hence, byte hit rate = hit rate
- Cost of a miss is the same
  - Get page from main memory
- Neither is true in web caching
  - Objects have different sizes
  - Cost of a miss (= get page from origin server) is different
- Most work on replacement policies in web caching focuses on hit rates

Optimal Replacement Policy

- What is the *optimal replacement policy*?
- Simple: It evicts the item that will be needed last
  - Each item has a value that tells when it will be needed next
  - Throw out item with largest such value
- "Minor" problem: This is impossible to implement
- How can we know when items will be needed next?
  - For memory caching, we can run program once and see the sequence of page references
  - For second run of same program, we know the sequence
- Still, not very practical
- Main use of optimal policy is to determine how good a practical algorithm is
  - If within 1% of optimal, then at most 1% improvement possible

Practical Replacement Policies

- Let’s look at some practical replacement policies
- First in the context of virtual memory systems
  - All of them can be used for web caching (and 1 is used)
  - Web caching -specific policies come later
- Policies:
  - Not-Recently-Used (NRU)
  - First-In-First-Out (FIFO)
  - Second Chance
  - Clock
  - Least Recently Used (LRU)
  - Not Frequently Used (NFU)
- All are well-known and well-researched

Not Recently Used

- Assume that each item has two information bits:
  - R = was referenced since time X (e.g., last clock interrupt)
  - M = was modified in cache (virtual memory system)
- At start, both bits are 0
- R is set to zero on e.g., every clock interrupt
- Pages classified in 4 classes:
  - Class 0: Not referenced, not modified
  - Class 1: Not referenced, modified
  - Class 2: Referenced, not modified
  - Class 3: Referenced, modified
- When we need to replace item, pick random item from the lowest category
- Logic: If item is recently referenced, it will be useful in near future too

FIFO

- First-In-First-Out policy is very simple
- We maintain a list of all items in cache
- Item at head of list is the oldest item
- New items are inserted at the tail and item at head is removed from cache
- If we get a cache hit, the list is not modified, i.e., the item keeps its place
- FIFO generally has poor performance because it may throw out an item which is heavily used
  - Example: Item gets put in cache, used heavily, but slowly other, less popular items get into cache. Eventually the heavily used item will be kicked out

Second Chance

- Second Chance remedies FIFO’s problem
- Policy is FIFO, with the modifications:

1. *If item at head has R = 0, it gets evicted (as per FIFO)*
2. *If item at the head of the list has R bit set, then clear R bit and move item to the tail of the list. Continue with next item in list*

- Idea: Look for an item that has not been referenced
- If all items have been referenced (R = 1), then Second Chance becomes FIFO
  - Has to go through the whole list

Clock

- Second Chance is inefficient to implement because it keeps items on a linked list
- Note: Inefficient for virtual memory systems, acceptable for web caching
- Clock algorithm is a different implementation of Second Chance; no difference in replacement

Least Recently Used

- Approximation of optimal algorithm:
  Items that have recently been heavily used will also be heavily used in the near future, and vice versa
- Idea: Throw out item that has been unused the longest
  - Least Recently Used replacement policy (LRU)
- For virtual memory, LRU is not cheap
  - In contrast: Easy to do in web caching
- Reason is that it requires a linked list of all items in cache, sorted by reference time
- On *every* access, list has to be modified
- Expensive for virtual memory, acceptable for web cache
- We need either special hardware or simulate in software

LRU with Hardware

- Solution 1: Hardware has counter that gets incremented on every access. Each item has field for counter.
- Counter value is stored on access, smallest counter value is the least recently used item
- Solution 2: Maintain matrix of items and references
- Assume system with *n* memory pages
- We need a matrix of *n* x *n* bits
- When page *k* is referenced, then
  - Set all bits in row *k* to 1
  - Set all bits in column *k* to 0
- Page to replace is the one with lowest value on its row
- See example below

LRU with Matrices

- 4 pages, references: 0 1 2 3 2 1 0 3 2 3

Page 0

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 \end{array}$$

Page 1

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 2 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 \end{array}$$

Page 2

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 2 & 1 & 1 & 0 & 1 \\ 3 & 0 & 0 & 0 & 0 \end{array}$$

Page 3

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 2 & 1 & 1 & 0 & 0 \\ 3 & 1 & 1 & 1 & 0 \end{array}$$

Page 2

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 2 & 1 & 1 & 0 & 1 \\ 3 & 1 & 1 & 0 & 0 \end{array}$$

Page 1

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 2 & 1 & 0 & 0 & 1 \\ 3 & 1 & 0 & 0 & 0 \end{array}$$

Page 0

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 0 \end{array}$$

Page 3

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 3 & 1 & 1 & 1 & 0 \end{array}$$

Page 2

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 1 & 0 & 1 \\ 3 & 1 & 1 & 0 & 0 \end{array}$$

Page 3

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 1 & 0 & 0 \\ 3 & 1 & 1 & 1 & 0 \end{array}$$

Not Frequently Used

- Previous LRU algorithms require special hardware
- LRU can be simulated in software
  - Not Frequently Used policy
- Each item (page) has a counter initialized to zero
- On every clock interrupt, R bit (0 or 1) is added to counter
  - Counter keeps track of how often page is referenced
- Replace item/page with lowest count
- Problem: *NFU never forgets anything*
- If an item was heavily accessed in the past, but not anymore, NFU will still want to keep it

Solution: Aging

- Simple modification:
  - Shift counters to right before adding R bits
  - Bring R bit in as the new left-most bit
- Example: 6 pages, 5 clock interrupts time

R bits	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0
Page 0	10000000	11000000	11100000	11110000	01111000
Page 1	00000000	10000000	11000000	01100000	10110000
Page 2	10000000	01000000	00100000	00010000	10001000
Page 3	00000000	00000000	10000000	01000000	00100000
Page 4	10000000	11000000	01100000	10110000	01011000
Page 5	10000000	01000000	10100000	01010000	00101000

- Remove page with lowest counter
- If a page has not been referenced recently, counter starts with 0’s

Difference of NFU and LRU

- Two differences between NFU and LRU

1. NFU is not really LRU
   - In example, pages 3 and 5 are least recently used (time 3)
   - But which was *really* first used in that time interval?
   - Not possible to know, so pick page 3, because 5 was referenced already earlier (time 1)
2. Counters are finite
   - Suppose 8 bit counters and two pages with all 0's
   - We pick either at random
   - But: Maybe one of the pages was referenced 9 ticks ago, and the other 1000 ticks ago
   - Not a problem in practice, interval between clock interrupts long enough to make this problem irrelevant

LRU in Web Caching

- LRU is expensive for virtual memory systems because it needs to maintain linked list
  - Not feasible for high performance systems
  - Hence approximated with NFU
- In web caching, speed is not as crucial
  - Page downloads take a couple of seconds
  - Maintaining a linked list of items in cache is fast (relatively)
- LRU is widely used in web caching
- Main reason: Good performance
- *Tradeoff:* Get bigger cache or develop better algorithm?
- Answer is not always clear
- More on web caching replacement algorithms later

Belady’s Anomaly

- A bigger cache will always have a better hit rate, right?
- WRONG! :-)
- References to objects: 0 1 2 3 0 1 4 0 1 2 3 4
- Space for 3 items or 4 items, FIFO replacement

	0	1	2	3	0	1	4	0	1	2	3	4
	0	1	2	3	0	1	4	4	4	2	3	3
		0	1	2	3	0	1	1	1	4	2	2
			0	1	2	3	0	0	0	1	4	4
	M	M	M	M	M	M	M			M	M	

= 9 misses

	0	1	2	3	0	1	4	0	1	2	3	4
	0	1	2	3	3	3	4	0	1	2	3	4
		0	1	2	2	2	3	4	0	1	2	3
			0	1	1	1	2	3	4	0	1	2
				0	0	0	1	2	3	4	0	1
	M	M	M	M			M	M	M	M	M	M

= 10 misses!

Stack Algorithms

- How is Belady's Anomaly possible!?!
- A cache can be characterized with:
  - Reference string to items (*r*)
  - Replacement algorithm (policy)
  - Number of items we can store in cache (*m*)
- Express state of cache as *M(m, r)*
  - State tells which objects are cached, given *m* and *r*
- Replacement algorithm is called a stack algorithm if

$$M(m, r) \subseteq M(m+1, r)$$

- In other words, any item cached in a cache with space for *m* items is also cached with *m+1* items of space
- Stack algorithms are preferred
  - LRU is stack algorithm, FIFO is not

Browser Caching

- Every browser since mid-90’s has a built-in cache for web content
- When a user views a page, it gets stored in the cache
- Since cache is stored on local disk, if user visits the same page again, it can be shown very fast
- Most of the browsers allow the user to decide how much space to use for caching
- Two problems:
  - What happens if the page has been modified?
  - What happens when we run out of space in cache?
- Look at first problem now, second a bit later

Freshness of Cached Data

- Freshness of data in web caching is a huge problem
  - Also known as cache consistency problem
  - Note: Applies to browser and proxy caching
- Problem definition:

  User accesses a URL, which gets stored in a client-side cache. Then the contents of the URL are modified. User accesses the same URL again and gets the stale content from the client-side cache.
- Three basic solutions:
  1. Check freshness of cached copy before sending it to the user
  2. Origin server explicitly tells how long a URL can be cached
  3. Heuristic for “guessing” when to check/refresh a URL

Check for Freshness

- Idea: Before sending cached copy to user, check with origin server if file has been modified
- How to check?
- Download new copy and compare?!?
  - No benefit from caching
- HTTP defines mechanisms for checking freshness
- So-called IMS GET request
  - IMS = If-Modified-Since header of HTTP

IMS GET

- When object was first retrieved, origin server includes (always) Last-Modified-header
  - Tells when object was last modified on server
- Cache sends GET request with If-Modified-Since-header
  - Value is the one given by server in Last-Modified
- Server replies either:
  - 304 Not modified = means cached copy is valid
  - 200 Ok = object modified and new version is in the reply
- In both cases, we have fresh object
- Cost: 1 extra HTTP request (+ TCP) in the case where cached copy was fresh
  - Note: If object has been modified, we need to get it anyway, so no extra cost in this case

Explicit Content Expiry

- If origin server knows when an object will be modified, it can explicitly let all clients know about this
- HTTP defines Expires-header which says when the object will become invalid
- Client can simply check the time and Expires-header to know if it should refresh the object from origin server
- Not much used in its original purpose
  - Hard to guess expiry times
- Widely used to disallow caching of objects
  - If Expires-header has current time (or time in past!), object cannot be cached at all
  - Also known as cache busting

Cache Busting

- Why would a content provider disallow caching of its own objects?
- Caching is beneficial to content provider, since it reduces load on origin server
- If content provider is concerned with freshness, it can use Expires-header (in the correct way :-)
- Problem in practice: Some client side caches ignore Expires-header and deliver stale content from cache
  - Especially done by ISPs to reduce their bandwidth costs
- Users and content providers unhappy
- Core issue: Client-side caching under the control of the client, not content provider
  - Compare later to content distribution networks

Expiry Heuristics

- If there is no explicit Expires-header, client should check for every request with IMS GET
- Not a problem for browser cache, but a big issue in a busy proxy cache serving many users
  - Need to send HTTP requests often, only to get 304 Not Modified
- Heuristic for determining when object should be refreshed and when to use cached copy
  - Note: Calculated locally by client
- It’s only a heuristic, so it might give wrong answers…
- Note: RFC 2616 (HTTP/1.1) explicitly allows use of heuristics in absence of explicit expiry/age information
  - Must send a Warning-header to client

Freshness Heuristic

- Popular Squid proxy cache uses the following heuristic
- Let:
  - OBJ_DATE = Time object was retrieved (Date-header in reply)
  - OBJ_LASTMOD = Value of Last-Modified-header
  - OBJ_AGE = NOW - OBJ_DATE (How long in cache)
  - LM_AGE = OBJ_DATE - OBJ_LASTMOD (How old it was)
- Heuristic: LM_FACTOR = OBJ_AGE / LM_AGE
- If value of LM_FACTOR is below a configured limit, then object is considered fresh, otherwise stale
- Let's look at it a bit closer

Freshness Heuristic: Logic

- We calculated

$$LM\_FACTOR = \frac{now - date}{date - last\_modified}$$

- Logic behind this equation:
  - If object was already old when we retrieved it, it is likely that it won't be modified anytime soon
  - If object was recently modified, it might be modified again soon
- Only tunable parameter is when do we consider objects to be fresh or stale

Freshness: Summary

- No good solution for freshness problem
- Important to solve this problem
  - In client-side caching, no solution exists
  - Lack of good solution main motivation for CDNs
- Explicit expiry hard to know in advance
- Heuristics for guessing
  - Heuristics typically work well
  - But they are only guesses

Proxies

- Definition of proxy: (Wikipedia)
  A proxy server is a computer that offers a computer network service to allow clients to make indirect network connections to other network services.
- Note that proxy (in its purest form) only takes requests from clients and forwards them onwards
  - Onwards can mean server or another proxy
  - Hence, client can be client or another proxy
- Proxies not just for web caching and content delivery
- Proxies also commonly used to bridge traffic between private networks and Internet
  - Although caching is also commonly used

Types of Proxies

- Pure proxy
  - Just acts as an intermediary
  - Typically used for going through a firewall
- Caching proxy
  - Proxy server with a built-in cache, especially for web content
  - Main focus for us
- Anonymizing proxy
  - Either a pure or a caching proxy
  - Masks client identity to server
  - Onion routing: Using several anonymizing proxies in chain
    - Also known as mix network

Proxy Caching Terminology

- Terminology defined in RFC 3040
  - But often misused in practice
- Proxy = (pure) proxy server
  - Sometimes used to mean any kind of proxy
- Caching proxy = proxy with a cache
  - Also known as “cache”, “proxy cache”, “web cache”
- Intercepting proxy = proxy which does not need any configuration on the client side
  - Intercepts client requests somehow (see below)
  - Often called “transparent proxy”, because client does not see it

Proxy Discovery

- How does client know to use a proxy?
- Three configuration possibilities:

1. Manual
   - User manually configures proxy address
2. "Automatic"
   - User manually configures proxy configuration
3. Request interception
   - Requests intercepted by an L4-switch and forwarded to proxy

- All three have been used

Manual Proxy Configuration

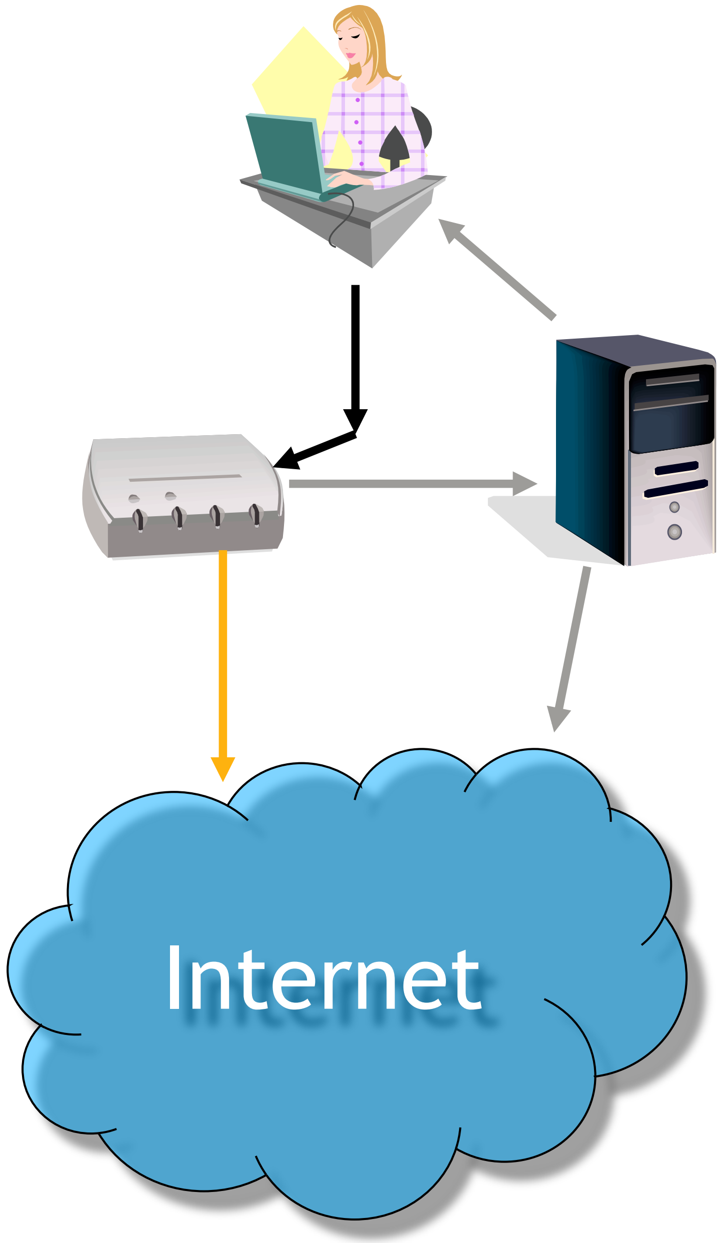
- Every browser (or operating system) offers a manual configuration of proxies
- Note: Proxies can be configured for other things besides web (e.g., FTP proxy)
- User must typically enter:
  - Address (name or IP) or proxy server
  - Port on which proxy is listening
- Can configure different proxies for different protocols
- But any request for given protocol always goes to same proxy
  - Typically can exclude certain domains from going through the proxy

“Automatic” Proxy Configuration

- User manually configures URL of a proxy auto-configuration (PAC) file
- Also supported by all browsers
- PAC file is JavaScript which determines how requests are to be handled
    - Script gets URL as input and returns proxy address and port as output (or lets request go directly)
- Typically configured per-protocol
- More fine-grained configuration is possible
    - JavaScript can look at parts of URL and make intelligent redirections based on URL

Intercepting Proxy

- Intercepting proxies rely on an L4-switch to redirect traffic to them
- No need to configure anything in the client
- L4-switch sits between the user and Internet and sends (web) traffic to the proxy
- Other traffic is unaffected



Caching Proxies

- Regardless of how the traffic gets to the proxy, caching proxies all work the same way
- Basic functionality of a caching proxy:

1. Receive request URL from client
2. Check if URL is cached
3. If URL is cached and fresh, send reply to client
4. If cached copy is not fresh or no copy is in cache, fetch a copy from the origin server
5. Put new object into cache, evicting another object if cache is already full

Key Functionality

- *How to determine if a cached object is fresh?*
  - See above under browser caching
- *How to decide which object to evict?*
  - Determined by the replacement policy of the cache
- Any of the previous replacement policies can be used
  - LRU is commonly used
- Recall three criteria:
  1. Maximize hit-rate
  2. Maximize byte hit-rate
  3. Minimize download time for user
- Reason for these: Heterogeneous objects
- *A given policy usually only optimizes one of the three!*

Web Caching Replacement Policies

- Let’s now look at some replacement policies developed for web caching

1. Least Recently Used (LRU)
2. Size-based policy
3. Download time optimizing policy
4. Least Frequently Used (LFU)
5. Greedy-Dual*
6. Multimedia caching with transcoding

- But first some general points about web caching

General Web Caching

- All policies based on same idea:
  Each cached object has a value (utility) associated with it. Objects sorted according to this value and “least valuable” gets evicted.
- Web caches have “more time” to compute values and sort objects than virtual memory
- But: Web caches typically able to store millions of objects, hence computation is expensive
- Practical caching works as follows

Practical Web Caching

- Squid (popular web cache) does LRU as follows:
  - Objects stored in several hash tables
  - For eviction, sort objects in one hash table and pick LRU object
  - Not true LRU, but close enough
- Replacement also optimized
  - In normal operation cache is full, so every access means replacement
  - In practice, Squid defines high- and low-water marks
  - If cache size over high-water mark, remove objects until size is below low-water mark
  - Replacement run only “every now and then”, more efficient

General Properties

- How to classify replacement policies?
- Mainly done according to how they exploit temporal locality of request stream
- Studies on web request streams show

1. Web request streams show temporal locality on short time scales
   - Good for LRU
2. Popularity of objects over long time intervals also exists
   - Need longer-term measurements of access frequency

Least Recently Used (LRU)

- Same LRU algorithm as we have seen before
- LRU was used because temporal locality in request streams had been observed
  - Recently used objects were used again
- Objects sorted according to last time they were accessed
- List is kept sorted every time an object is accessed
- LRU is widely implemented and used
  - Mainly because it was well-known from other caching
- Performance of other replacement policies often compared against LRU

Weaknesses of LRU

- LRU is widely used and implemented
- Performance for many situations considered “good enough” to merit use
- LRU has three main weaknesses

1. Does not take into account different object sizes
2. Does not take into account different retrieval costs
3. Does not take into account access frequency

- For last point, there exists LRU-K variant
  - Maintains last K access times of object
  - Uses them to calculate access frequency

Size-Based Policy

- Size-based policies use object size to sort objects
  - No official name for these policies
- Can use size in two ways

1. Sort from smallest to largest
   - Improves hit-rate
2. Sort from largest to smallest
   - Improves byte hit-rate

- Purely size-based policies not widely used
- Object size used as one of many parameters for more sophisticated policies
  - Typically divide “metric” by size, gives preference to smaller objects over larger objects

Size vs. LRU

- *Size-based policy which prefers small objects has better hit-rate, but lower byte hit-rate than LRU*
- *Size-based policy which prefers large objects has lower hit-rate, but higher byte hit-rate than LRU*
- Generally no way to get better performance on both

- LRU throws unused objects out of the cache
- Pure size-based policy allows objects to stick around far after they are no longer useful
  - In other words, it does not forget (recall aging)
  - For this reason, pure size-based policy is not used

Download Time Optimization

- Some policies sort objects according to the time it takes to download them from origin server
- Preference given to objects from “slow” servers
- Goal: Minimize average download time for user
- Results show it is very efficient
- Comparison on hit-rate and byte hit-rate not meaningful
  - Typically lower than with LRU
- Not widely implemented
- Used in combination with other policies
  - Just like size-based policies

Frequency-Based Policies

- Which is more important?
- Recency or frequency?
- Recency was found to be good predictor of utility
- As browser caches get better, web caches see less locality in request stream
- Frequency-based policies capture the other kind of temporal locality: Access frequency
- Idea: More frequently accessed objects should remain in cache over less frequently accessed
- Compare with NFU
  - NFU simulates LRU with counters, not true reference count
  - Frequency-based policies work with real reference counts

Least Frequently Used (LFU)

- LFU keeps a reference count for each object
- Objects sorted from lowest count to highest
- Evict object with lowest reference count
- If two objects have same count, use a tiebreaker
  - For example, access time, size, or random
- Pure LFU not a very useful policy, too many gotchas
- Things to consider for implementation:
  - How are reference counts maintained?
  - How to do aging?

LFU Details

- Reference counts for cached objects or for all objects?
- *Perfect-LFU keeps reference counts for all objects*
- *In-Cache-LFU keeps them only for cached objects*
- Perfect-LFU beats LRU on hit-rate and byte hit-rate
- In-Cache-LFU loses to LRU on both counts
- Differences small in both cases
- Note: To store reference count, we must store URL and counter for that URL
  - URL is small compared to objects, can store many
  - Enough to approximate Perfect-LFU?

LFU and Aging

- Variant of LFU with dynamic aging of reference counts
- On a hit, set count to current reference count plus minimum reference count in cache
- Shown to have higher byte hit-rate than LRU
- This algorithm is also called LFU-DA

- One more variant: LRFU
- Combines LRU and LFU with weights
- By adjusting weights, we can get LRU or LFU or something else
- Goal of this policy is to allow dynamic tweaking

Greedy-Dual*

- Let’s look at a more sophisticated replacement policy
- But first some basics
- Effectiveness of caching is based on temporal locality in the request stream
  - Typically modeled with the interarrival time distribution
- Consider two request streams:
  - XAXBXCXDXEXFX…
  - GGHHIIJJKK…
- Both exhibit temporal locality
- If we re-order streams randomly, then:
  - Temporal locality in first stream is preserved
  - Temporal locality in second one is not preserved

Temporal Locality

- From last slide, temporal locality can have two causes

1. Temporal locality due to popularity
   - Preserved under reordering
2. Temporal locality due to correlation in time
   - Not preserved under reordering

- Most of existing research based on assumption of temporal locality due to popularity
  - This is true and the effect is strong, but there is more
- Temporal correlation can also be modeled
  - Take equally popular objects and look at interarrival times
  - Distribution can be quantified with one parameter β, because it's a power-law distribution (typically $0.3 \leq \beta \leq 0.7$)
  - Values of β relatively stable for large ranges

GreedyDual* Replacement

- GD* is an improvement over GDS (GreedyDualSize)
- Let *p* be an object
  - *s(p)* is the size of the object
  - *c(p)* is the cost to fetch it
  - *f(p)* is access frequency (~ reference count)
  - *u(p)* is utility of object
  - β is as above
- GDS algorithm works as follows:
  - Object utility *u(p) = c(p)/s(p)*
  - Inflation value *L*
  - When an object is hit, we set *H(p) = L + u(p)*
  - When an object is evicted, we set *L = H(p)* of evicted object
  - GD* re-defines utility u(p) and aging mechanism

GreedyDual* Details

- Utility
  - GD* defines utility as the normalized expected cost saving if the object stays in the cache
  - $u(p)$ should be proportional to $(f(p) * c(p)) / s(p)$
  - $f(p)$ is approximated with the reference count
- Aging
  - Dynamic aging similar to GDS (values H(p) and L)
  - When L = H(p), then object is candidate for eviction
  - On a hit, H(p) set to base value + L
- Base value
  - Base value should reflect document utility and reference correlation
  - Time to stay in cache proportional to $u(p)^{1/\beta}$
  - Set base value to $u(p)^{1/\beta}$

GreedyDual* Algorithm

- Algorithm works as follows

```
$L = 0$
when object $p$ is requested, do
    if $p$ is in cache
    then
        $H(p) = L + \left(\frac{f(p)c(p)}{s(p)}\right)^{1/\beta}$
    else
        fetch $p$
        while not enough space in cache for $p$, do
            $L = \min(H(q) \mid q \text{ is in the cache})$
            evict minimum $q$
        $H(p) = L + \left(\frac{f(p)c(p)}{s(p)}\right)^{1/\beta}$
```

GreedyDual* Remarks

- GD* is actually a family of replacement policies
- β controls reference correlation
- Small β means weak correlation and slower aging
- When β=1, we get LFU algorithm with dynamic aging (see earlier)
  - Note: β not limited in any way, but experience shows β is small
- When β is close to 0, GD* tends towards an LFU-variant

GreedyDual* Performance

- Compare performance of GD* to LRU, LFU-DA, and GDS
  - See earlier for LRU and LFU-DA, GDS not covered in this course
- Note: Only one of many parameter combinations shown
  - Others give similar results, i.e., ranking is the same, but differences usually smaller

Policy	Hit-rate	Byte hit-rate
LRU	33.3%	31.1%
GDS	36.5%	31.4%
LFU-DA	39%	33.7%
GD*	50.1%	37.6%

How to Tune β?

- Performance better when β is smaller
- Optimal performance usually when β is about 0.5
  - Same value observed for β in real-world request traces
- Confirms that $u(p)^{1/\beta}$ is appropriate base value
- Two more observations:

1. When cache is small, β has larger effect
   - Capturing short-term correlation important for small cache
2. Small β hurts hit-rate less than large β

Multimedia Caching

- Above replacement policies tailored for web content
  - As opposed to virtual memory
- Still, every object is treated the same way
- Some web objects are different from others
- Multimedia objects can suffer data loss without visible effect to the end-user
- Can we somehow take advantage of that?
- Basic idea: Instead of replacing complete objects, replace only parts of them
- Assumes that we can remove data from an object without affecting user-observed quality

Multimedia Objects

- Following techniques apply only to multimedia objects
  - Pictures, photos, audio, video
- Other objects, e.g., HTML-pages, must be kept whole
  - Not possible to throw out half of an HTML-page…

1. Pictures and photos
   - In particular, suitable for JPEG and other photographic images
   - Not well-suited for graphics and line art, but possible
2. Audio
   - Possible, but rarely used. See also transcoding
3. Video
   - Possible with layered encoded video, but also not common
   - Layered encoding not supported by all video formats
   - Transcoding is also possible for video

Progressive JPEG Example

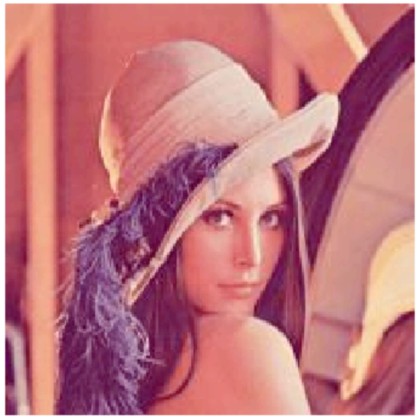
- JPEG standard defines a progressive mode
- Normal JPEG consists of 4 phases:
  1. Image preparation
  2. DCT-transformation
  3. Quantization (information loss)
  4. Entropy coding
- Progressive mode performs steps 1-3 as normal JPEG
- Normal JPEG does entropy coding by 8x8-blocks
  - Zig-zag traversal of AC-coefficients
  - DC-coefficients DPCM-coded
- Progressive JPEG traverses step 4 in a different way

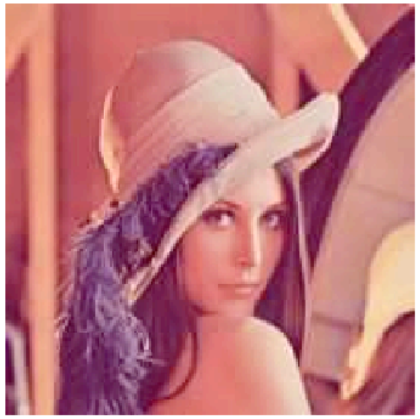
Progressive JPEG Layers

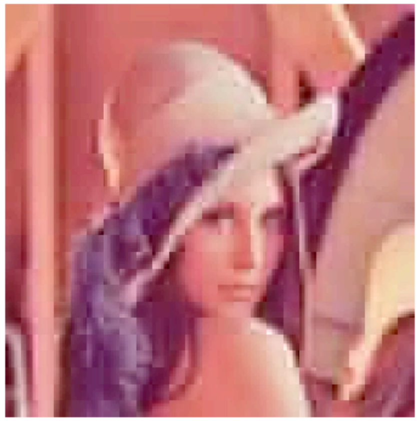
- Number of layers is implementation-defined
- Popular IJG JPEG-library defines 10 layers
  - 6 for black-and-white photos
- Layer 1 has 7 highest bits of DC-coefficients
  - Blocky image
  - Least significant bit of DC-coefficients in layer 7
- Other layers contain different bits of AC-coefficients
  - Lower layers have 6-7 most significant bits
  - Layers 3, 4, 7, and 8 have color information
- Layer 10 has least significant bit of AC-coefficients
  - This layer has about 30-40% of the bits of the file
  - Can be removed without any visible loss of quality

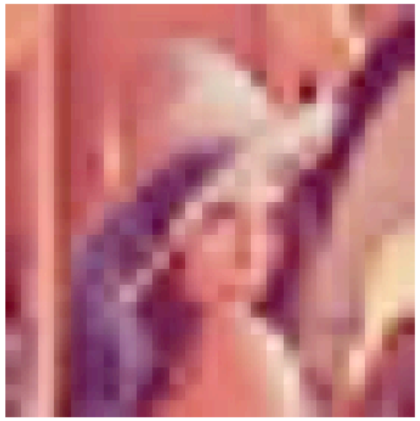
Progressive JPEG Example

- Lena image at different levels
- Typically applies:
  - With 6 levels left, visual quality largely the same
  - Difference visible if you know what to look for :-)
- Means for us:
  - Can throw 50% of the bits in an image away without the user noticing
- Assumes images are created or stored in progressive mode


Full


6 layers


3 layers


1 layer

Recoding

- Recoding means re-encoding an object in cache
  - Sometimes (incorrectly) called transcoding (see below)
- Recoding assumes a progressive-encoded file format
  - Possible with JPEG and MPEG-4
- When an object is to be evicted, we recode it instead of evicting it from the cache
  - Benefit: Object is now smaller, so we have free space in cache
  - Freeing up space is exactly the goal of a replacement policy
- Recoded object remains in cache
  - Gets kind of a second (and third, and fourth, …) chance
- Recoding-based replacement policies have been shown to improve hit-rate and byte hit-rate
  - But see below about byte hit-rate
- Note: Recoding is not a real replacement policy. It needs another policy to decide which object to recode

Recoding Gotchas

- What happens when a recoded object is delivered from cache and user is not happy with quality?
  - Maybe because of object having been recoded too many times
  - How can user force a reload from the origin server?
  - Should such an object count as a hit?
- How do we count byte hit-rate?
  - Weighted by size of recoded object or original size?
  - Second case can be justified, since we saved fetching that many bytes from origin server (= definition of byte hit-rate)
- What happens right after recoding?
  - Object was recoded because it was "least useful"
  - Recoding might not change that, so object gets selected again…
    - For example, if LRU selects an object for recoding, the same object is still the LRU object after recoding…
  - Need to adjust object "utility" somehow (depends on main policy)

Recoding Summary

- Recoding interesting possibility for multimedia objects
- Not a real replacement policy
  - Instead, can be used with any policy
  - Recoding replaces replacement ;-)
- Not widely used in practice
  - Main reason: Recoding mainly aimed at saving space in cache, but hard disks are cheap and recoding objects is “expensive”
  - Hence, small savings in size not worth the CPU-effort
- Above we discussed JPEG images, but same techniques apply for layered encoded video

Transcoding

- Transcoding similar to recoding in the sense that objects are re-encoded in proxy
  - Recoding is a simple form of transcoding
- Goal often to send “lighter” versions to weak clients
  - For example, a PDA cannot show full version video
  - Then, re-encode video at lower resolution and bit-rate
- Transcoding not really related to replacement policies
- Main interest of transcoding is when:
  - Proxy serves many different types of clients
    - For example, desktop, PDA, mobile phone
  - Proxy has sufficient CPU-power to re-encode
    - Transcoding usually requires decoding and new encoding
- Not widely used, but well-researched

Summary of Replacement Policies

- We have seen several replacement policies for web caching
- Any existing policy can be used, but better performance can be obtained when policy is tuned for web objects
- Actual policies:
  - LRU
  - Size
  - Download time
  - LFU
  - GD*
- Multimedia caching:
  - Recoding
  - Transcoding

Cooperative Caching

- Cooperative caching means several web caches collaborating together
- Typically it means the following:
  If caches A and B are cooperating, then when A has a miss, it will ask cache B for the object (and vice versa)
- Cache cooperation common in three cases:

1. Cache clusters
2. Cooperating caches
3. Caching hierarchies

- Last two often used together

Cache Clusters

- Cache clusters have several caches near each other in a cluster
- Main reason to improve performance
  - A single cache can handle only a certain amount of requests
  - Installing several caches in a cluster improves performance
  - Same logic as with server farms
- Problem: How to distribute requests to caches?
- One common solution to use L4-switch
  - Switch knows which caches are heavily loaded
- Problem with L4-switch:
  - Requests for same URL can go to different caches

  --> We get misses instead of hits
- How to make requests for same URL go to same cache?

Cache Array Routing Protocol (CARP)

- Consider a cluster (array) of caches
- All caches are known to client
  - Reasonable assumption, since new caches installed rarely
- When client wants to fetch a URL, client computes:
  1. Hash of the URL, *h(U)*
  2. Hash of each of the caches (e.g., IP-address), *h(C)*
  3. Hash the above two hashes together, $H = h(h(U)+h(C))$
  4. Order caches for each URL according to *H*
  5. Send request to cache with highest *H* value
- Because hash functions are same for all clients, a request for the same URL will be sent to the same cache
  - If new caches join, some requests will go to wrong caches
  - In an array of N caches, a new cache means 1/N wrong requests

CARP in Practice

- CARP specified in an RFC
- Implemented in Squid and some other products
- Not very widely used
  - L4-switch does the job in a different way
  - Disk space not such a critical resource after all…
- Note: CARP can be considered a precursor to distributed hash tables (DHT)
- Same basic idea: Distribute objects on nodes with hash functions
- Difference: CARP requires that all nodes are known
  - Not required in DHTs

Cache Collaboration

- Two or more caches can be set up to collaborate
- Means simply: (recall from above)

  If caches A and B are cooperating, then when A has a miss, it will ask cache B for the object (and vice versa)
- How are caches connected?
- Parent-child?
  - See cache hierarchies
- Siblings?
  - All caches equal

Problem Statement

- Cache A collaborates with N other caches
- If A has a miss, A will send the request to one of the N
- To which cache should A send the request?
- One possibility is to use CARP
- Problem: CARP sort of assumes all caches are under the same administrative entity
  - Not strictly so, but…
- Typical cache collaboration has caches in many different administrative entities
- Goal: Send request to a cache where it will be a hit
- *Idea: Let’s ask the others if they have the object!*

Inter-Cache Protocol (ICP)

- ICP defined for cache collaboration
- Cache A sends an ICP request to other caches
  - Request contains URL
- Other caches tell whether they have the object or not
- Cache A sends request to one of the caches with object
- Problems:
  - ICP messages sent over UDP --> Lost messages
  - Even without loss, (significant) additional delay
- How can we pick the cache without asking first?
- Idea: Let caches tell other caches what they have

Cache Contents

- How can a cache let others know what URLs it has?
- Simple, it sends a list of URLs of all cached objects
- Except that this takes far too much space
- Example:
  - URL’s average length is 60 bytes
  - Cache has 1 million objects
  - Cache collaborates with 10 other caches
  - Then we need ~600 MB of storage for this!
- Need a better solution
- How about compressing the data somehow?
- Solution: Cache digests
  - Summary cache based on same idea, but it’s another solution

Cache Digests

- Idea of cache digests is to compress the information about the contents of the cache
  - Results from Rousskov, Wessels, “Cache Digests”, Computer Communications and ISDN Systems, Nov. 1998
- Works as follows:

1. Cache generates compressed representation
   - Must update this when new objects come in and old ones get evicted from cache
2. Cache sends compressed form to other caches
   - Again, must be updated periodically
3. When a cache has a miss, it:
   - Checks if any other cache has the URL
   - If yes, send the request there

- *So, how do we form the “compressed representation”*

Cache Digests: Compression

- Cache digests use Bloom filters
- Bloom filters are "*a space-efficient probabilistic data structure that is used to test whether or not an element is a member of a set*"
  - False positives are possible
  - False negatives are NOT possible (but see below for us)
- Bloom filter is an array of *k* bits
  - Also need *m* different hash functions, each maps key to a bit
- To insert, calculate all *m* hash functions and set bits to 1
- To check, calculate all *m* hash functions and if all bits are 1, key is "probably" in the set
  - If any bit is 0, then it is definitely not in

How to Use Bloom Filters?

- Cache creates a Bloom filter out of the cached objects
- Send Bloom filter to other caches who can check if the object is cached at the sending cache
- Four possible results:
  1. True hit = Object was predicted to be cached and it was
  2. False hit = Object was predicted to be cached, but was not
  3. True miss = Object was predicted not to be cached and was not
  4. False miss = Object was predicted not to be cached, but it was
- Want to maximize “trues” and minimize “falses”
- Note: False misses not possible with normal Bloom filters, but our filters are not synchronized
  - New object might be cached after sending digest

False Hits

- False hits are a big problem with cache digests
- False hit means cache A thinks cache B has the object cached and sends request there, but B doesn't have it
- What should B do in this case?
- Typically this kind of behavior is not accepted
  - Caches configured as siblings (see later)
- Solution was to add a cache-control header in HTTP requests to siblings
- If Cache-Control header is set to "only-if-cached", receiving cache will reply with "504 Gateway timeout" if the object is not cached

Cache Digests: Performance

- Performance evaluated with NLANR caches
  - NLANR = National Laboratory of Applied Network Research
- NLANR operates several caches in the USA
- Typically used as top level in a caching hierarchy
- Goal of performance evaluation:
  - Speed and efficiency compared to ICP
  - Accuracy of digests
- In terms of performance, cache digests beat ICP
  - Difference quite large, larger than one RTT between caches
  - Typically 30% gain
  - Reason for longer than RTT delay is that with ICP, sending cache has to wait to receive several replies
  - With cache digests, no need to wait to send request

Accuracy of Digests

Cache	True hit	False hit	True miss	False miss	Total true
PB	33.6%	18.1%	48%	0.3%	81.7%
UC	34.7%	15.5%	49.4%	0.3%	84.1%
BO	42.2%	17.3%	40.1%	0.4%	82.3%
SV	25.7%	6.6%	67.7%	0.0%	93.4%

- 4 out of 6 studied caches shown
- False hit rate is far too high in most cases
  - SV-cache was one of the smallest caches in the study
- Also, overall “true”-ratio far from satisfactory
  - Authors estimated 95% “true” needed for digests to be useful
- Digests have been improved since this study was made

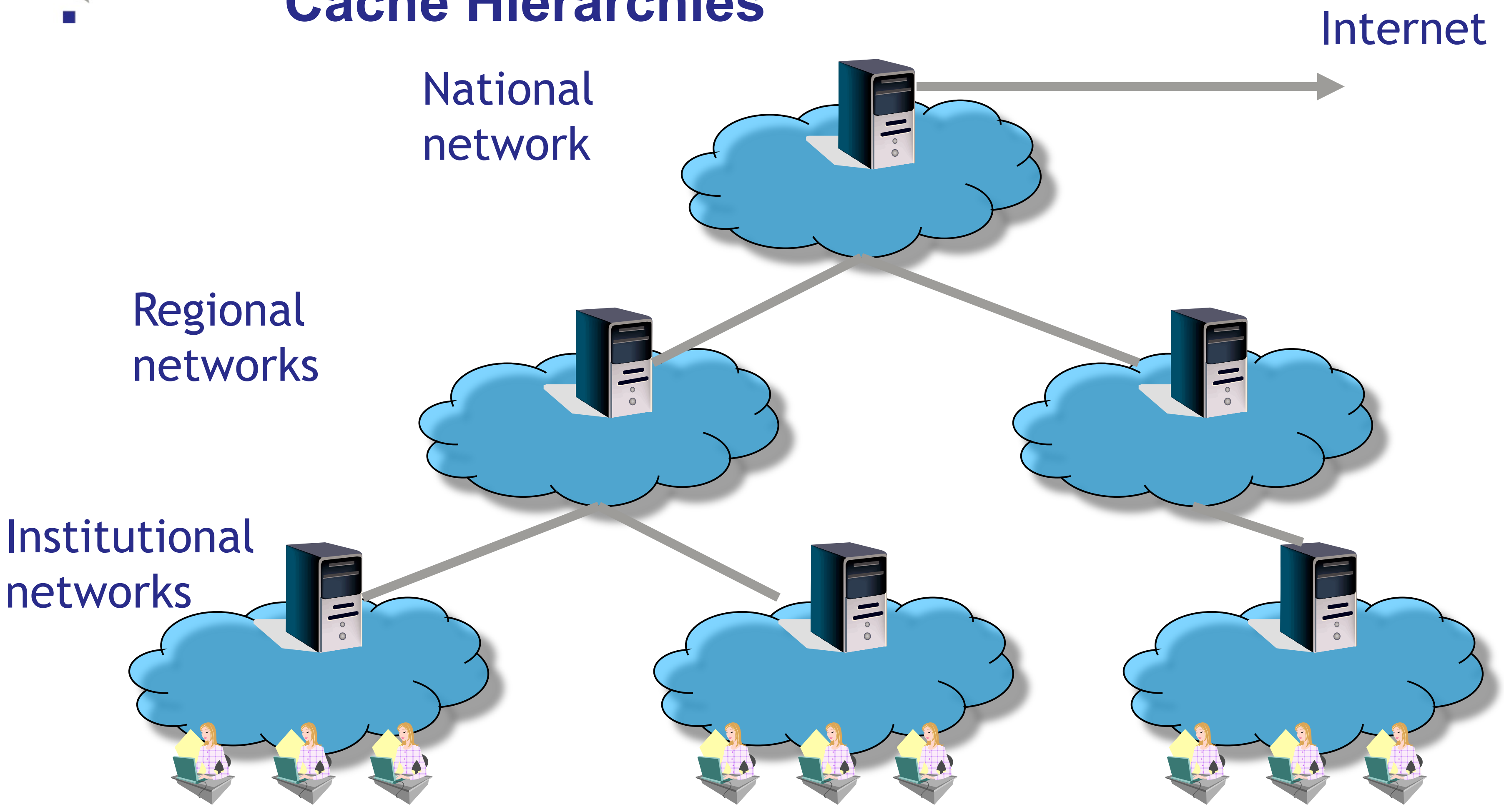
Digests: Other Details

- Compared to ICP, digests cause less network traffic
    - Somewhat less in terms of amount
    - Traffic from ICP is constant, but not much
    - Traffic from digests is rare, but each time transfers lots of data
- Caches need additional memory to store digests
- Updates once per hour seem sufficient
    - No need to do incremental updates, just send new digest

Caching Hierarchies

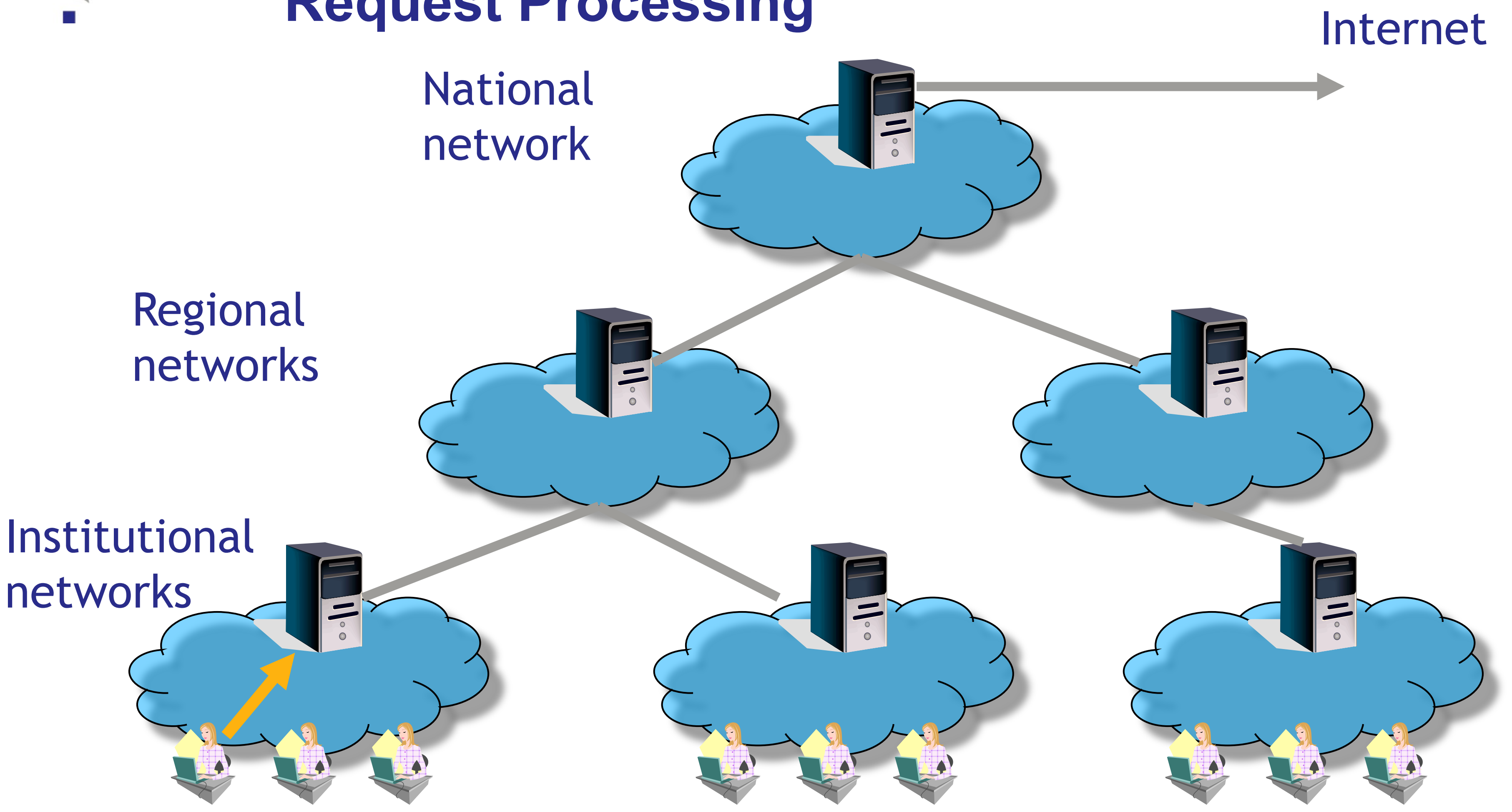
- Caching hierarchies consist of cooperating caches arranged in a hierarchy
- Why hierarchy?
- Because the Internet is also a hierarchy
- Caching hierarchies typically reflect the underlying Internet topology and connectivity
- Basic idea: Instead of using “expensive” bandwidth right away, try a longer path, but with “cheaper” bandwidth

Cache Hierarchies



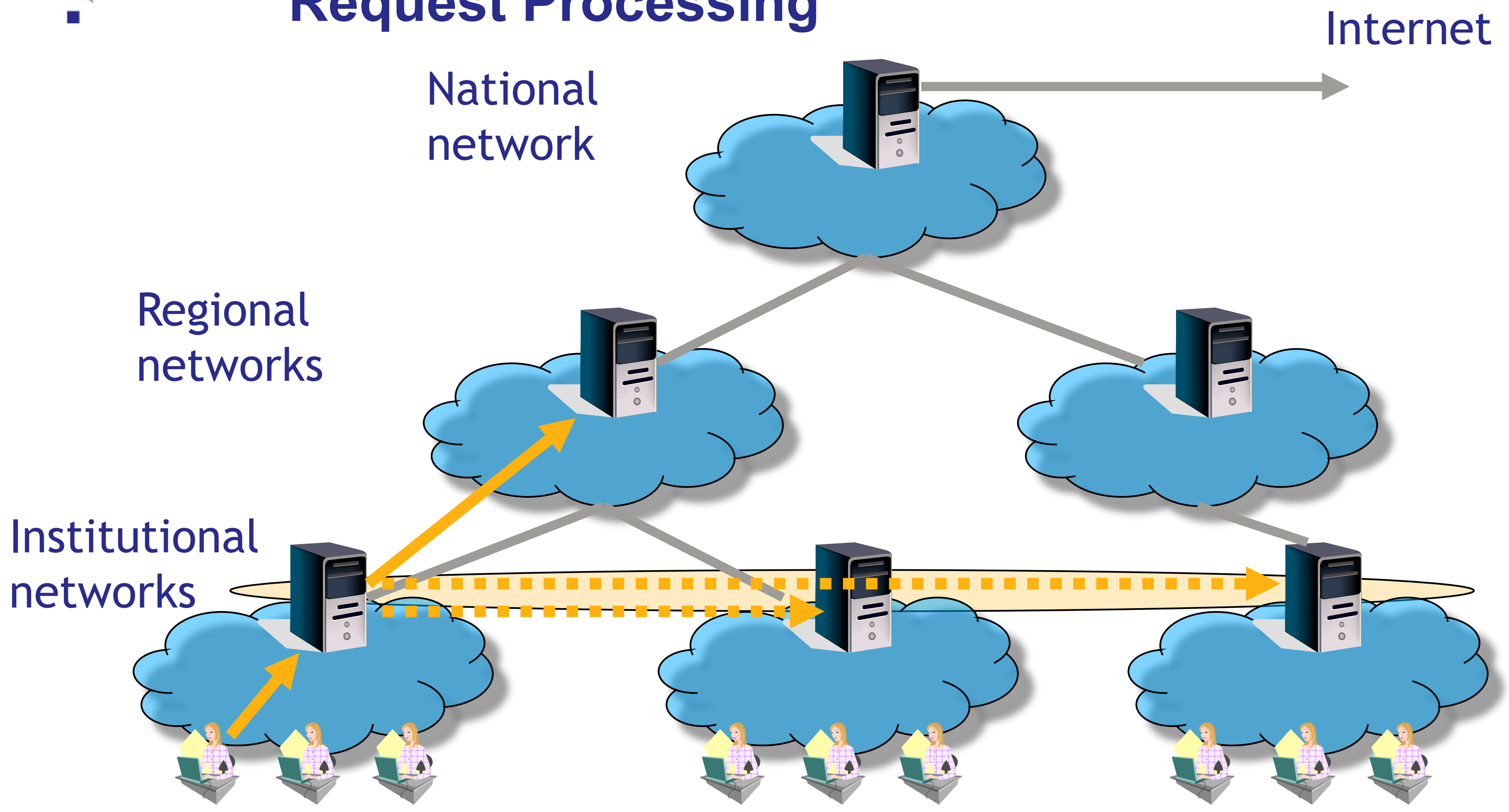
- Levels in hierarchy represent actual network topology

Request Processing



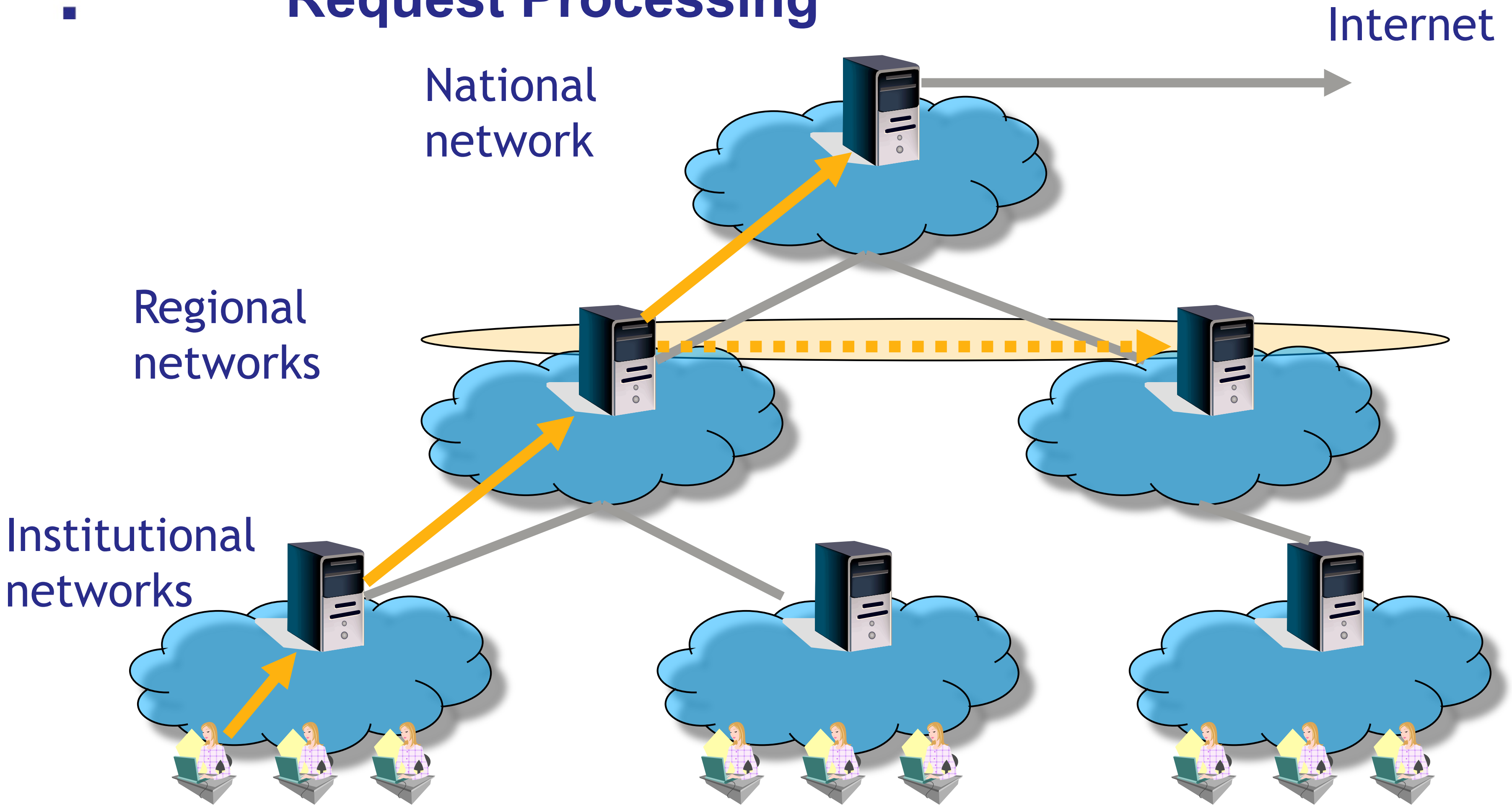
- Users send requests to their institutional (local) caches

Request Processing



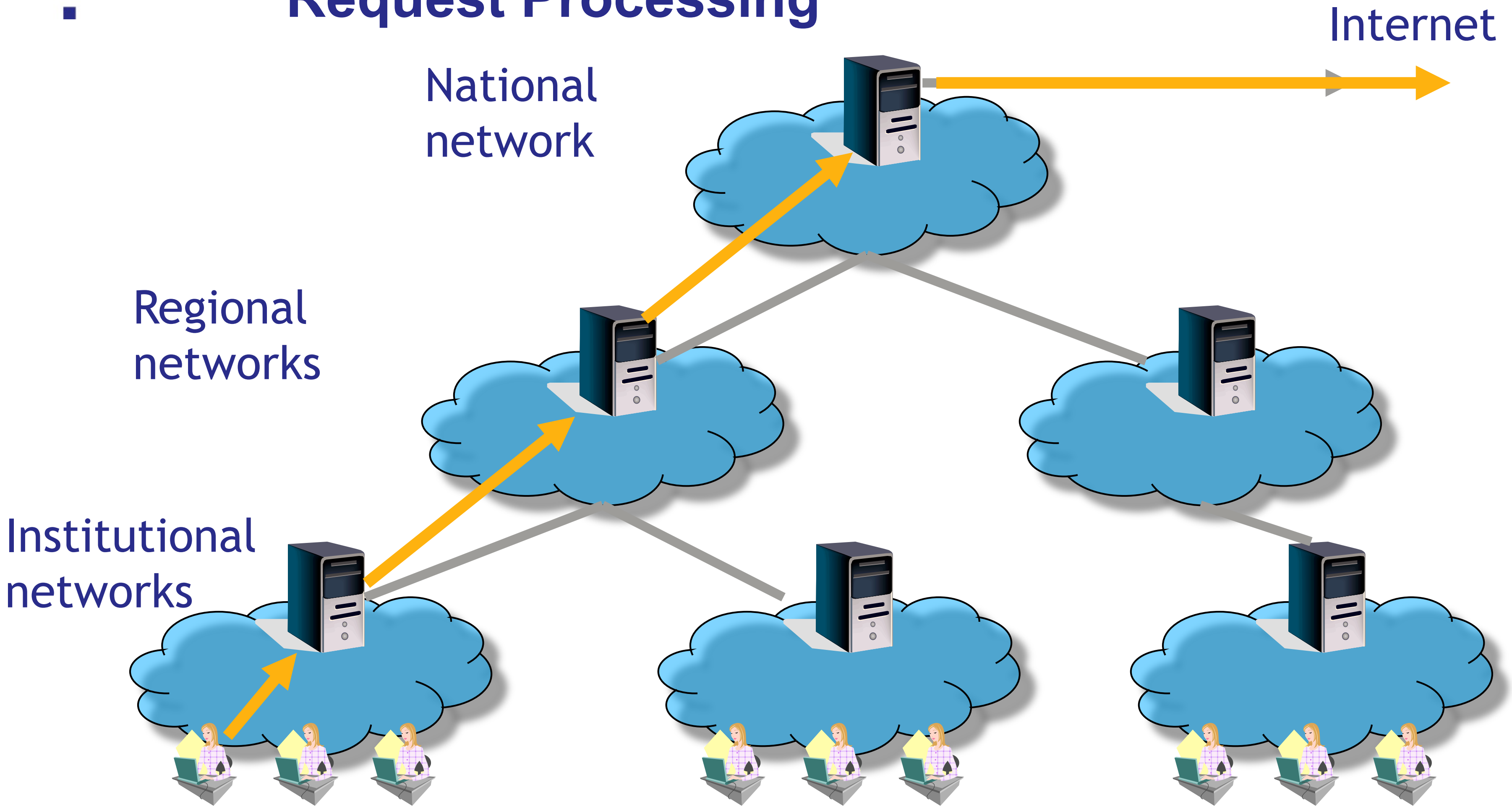
- Institutional caches can cooperate as siblings
- Misses are sent to the parent cache

Request Processing



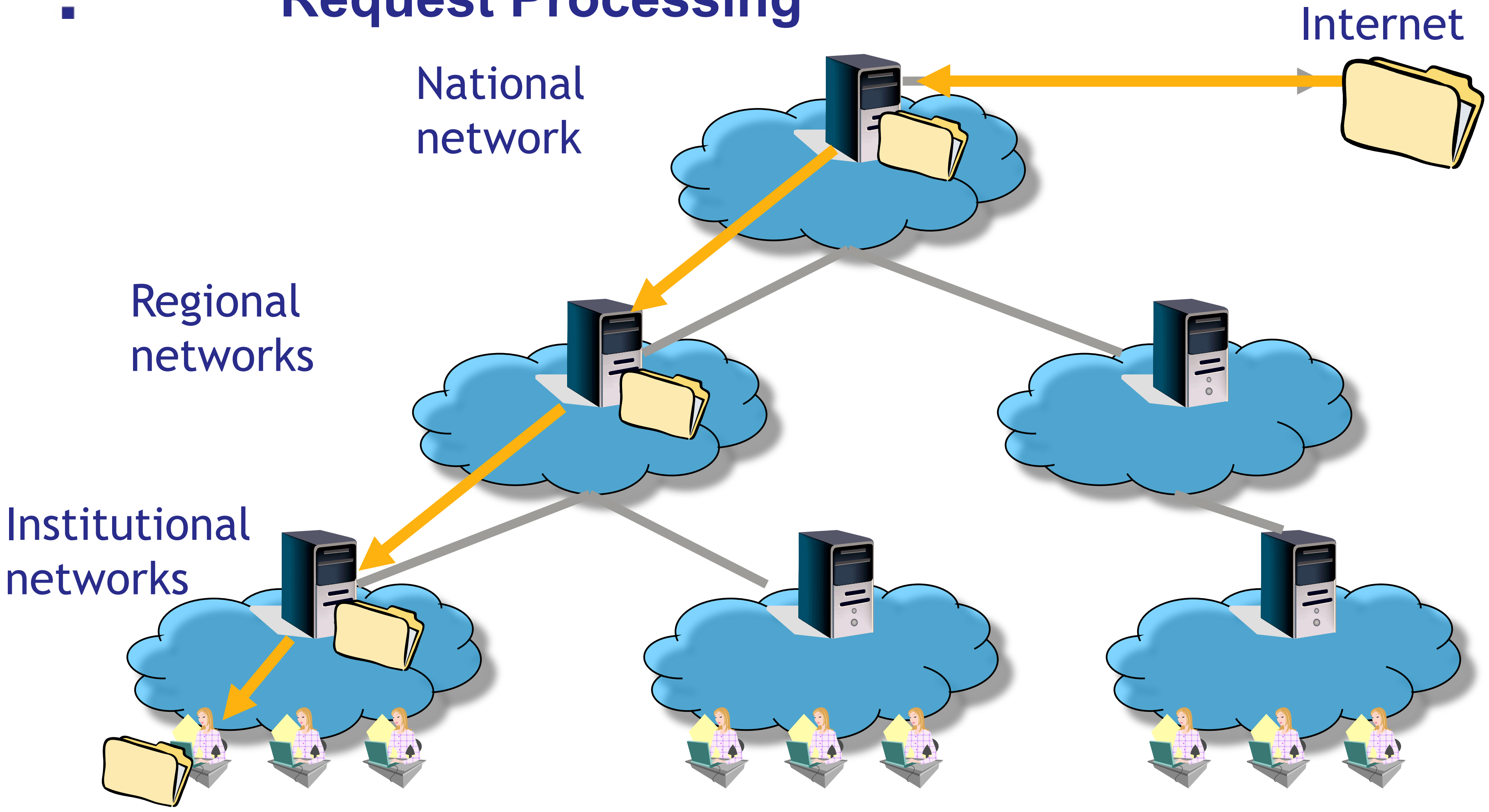
- Regional caches also cooperate as siblings
- Misses are sent to the national (root) cache

Request Processing



- National cache retrieves object from origin server, if miss

Request Processing



- Object retrieved from origin server
- Copy created in all caches on the request path

Cache Hierarchies

- Recall two types of relationships between caches

1. Parent-child
   - Parent typically has several children
   - Creates larger user base, better cache performance
   - Parent will fetch misses for its children
   - Often this corresponds to a provider-customer relationship in the underlying IP-network
2. Siblings
   - Caches on same level cooperating
   - Can only fetch hits from siblings
   - Aggregates some traffic and makes the population larger
   - Not necessarily any actual relationship in IP-network

Why Siblings?

- Why have siblings?
- Parent also aggregates population under it and can more easily benefit from its position
- Siblings only useful for retrieving hits
- So, why siblings?
- Siblings typically close in network
  - Getting objects from them is fast
  - Knowing that it was a miss is fast
- Going through a sibling reduces load on parent
  - Important for caches near root of hierarchy

Cache Hierarchies: Practice

- Caching hierarchies with several levels have been widely used in many countries
- Typical arrangement:
  - Institutional level: Universities and similar
  - Regional level: Group institutions in the same region
  - National level: One root cache per country
- Above arrangement was very typical in Europe
- In USA, national level was NLANR caching network
  - Regional level often non-existent

Chapter Summary

- Client-side techniques
- Caching basics
  - Replacement policies
- Browser caching
- Freshness
- Proxies
  - Proxy discovery
  - Proxy operation
- Proxy caches
  - Replacement policies
- Cooperative caching