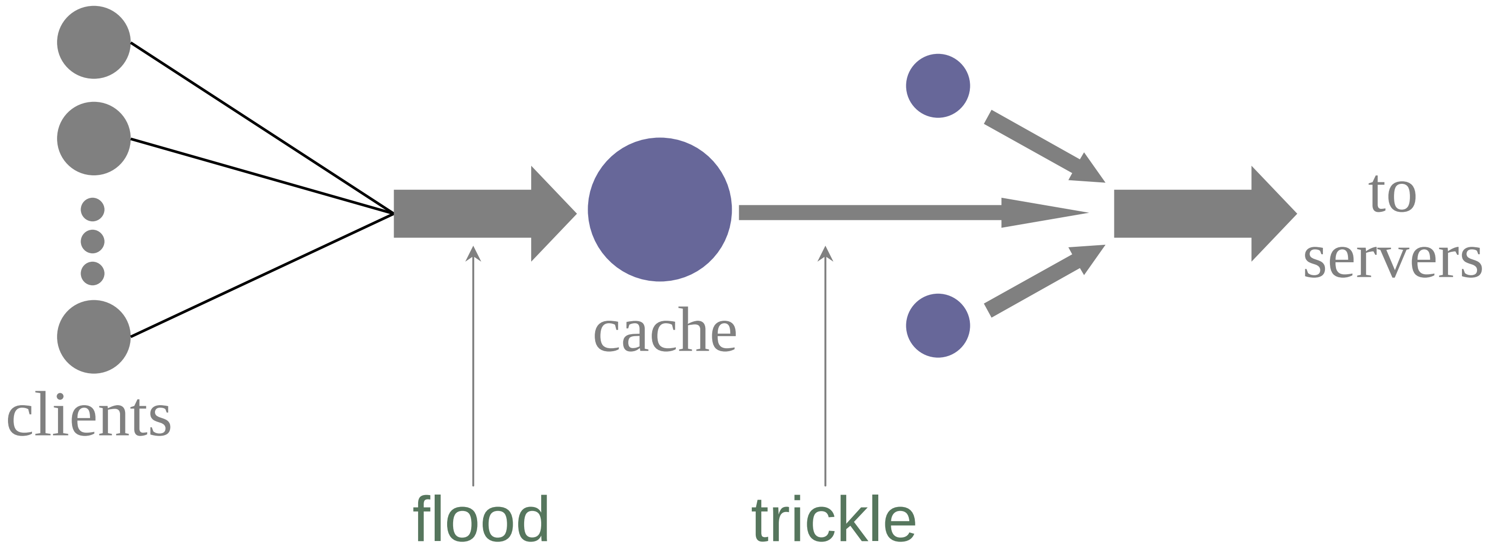
The “Trickle-Down Effect”

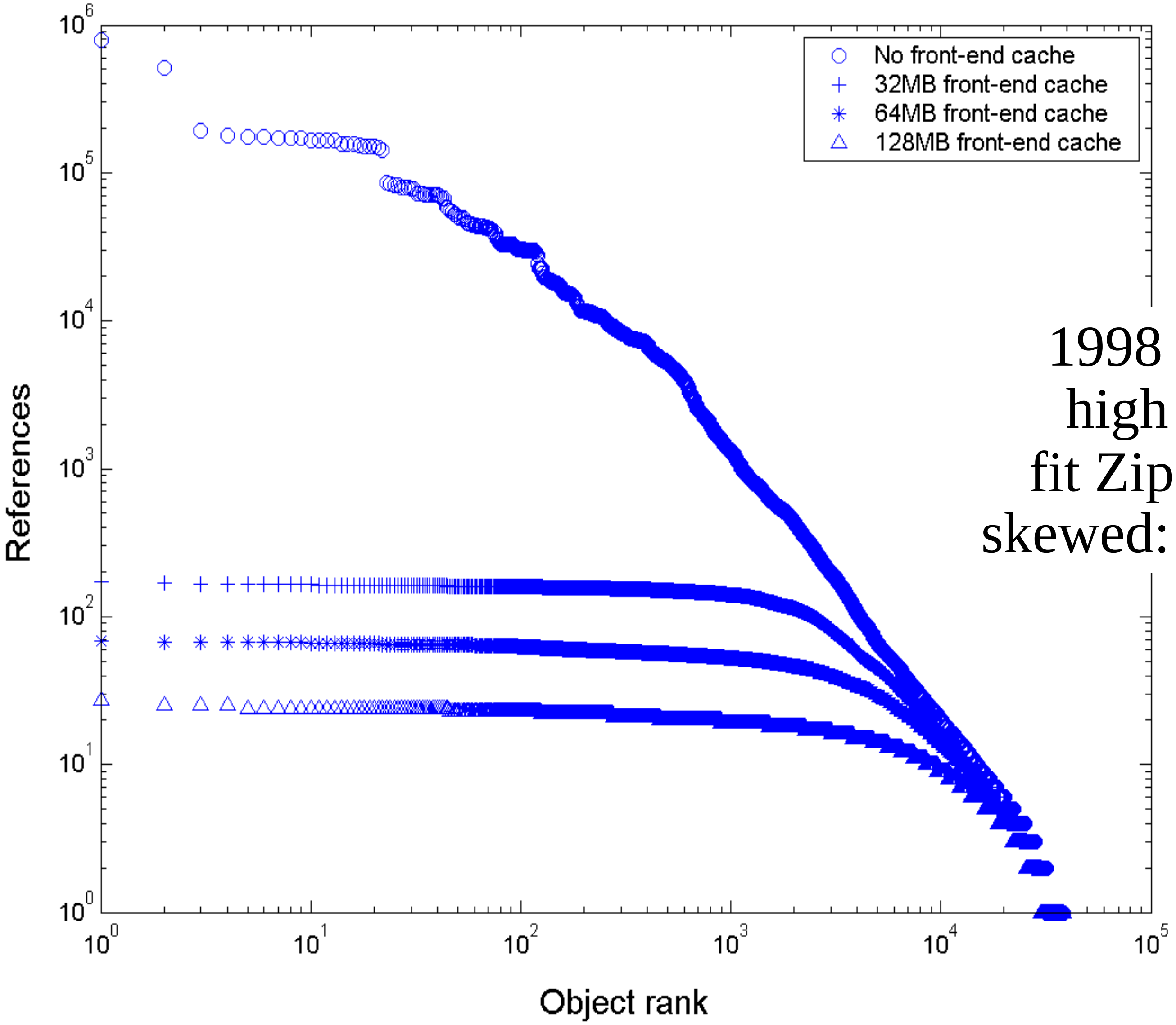


What is the effect on “downstream” traffic?

What is the significance of this effect?

How does it impact design choices for components “behind” the caches?

A Look at the Miss Stream



1998 *ibm.com*
high locality
fit Zipf α= 0.76
skewed: 77 % / 1%

What’s Happening? (LRU)

Suppose the cache fills up in *R* references.

(That’s a property of the trace *and* the cache size.)

Then a cache miss on object with rank *i* occurs only if *i* is referenced….

probability p_i

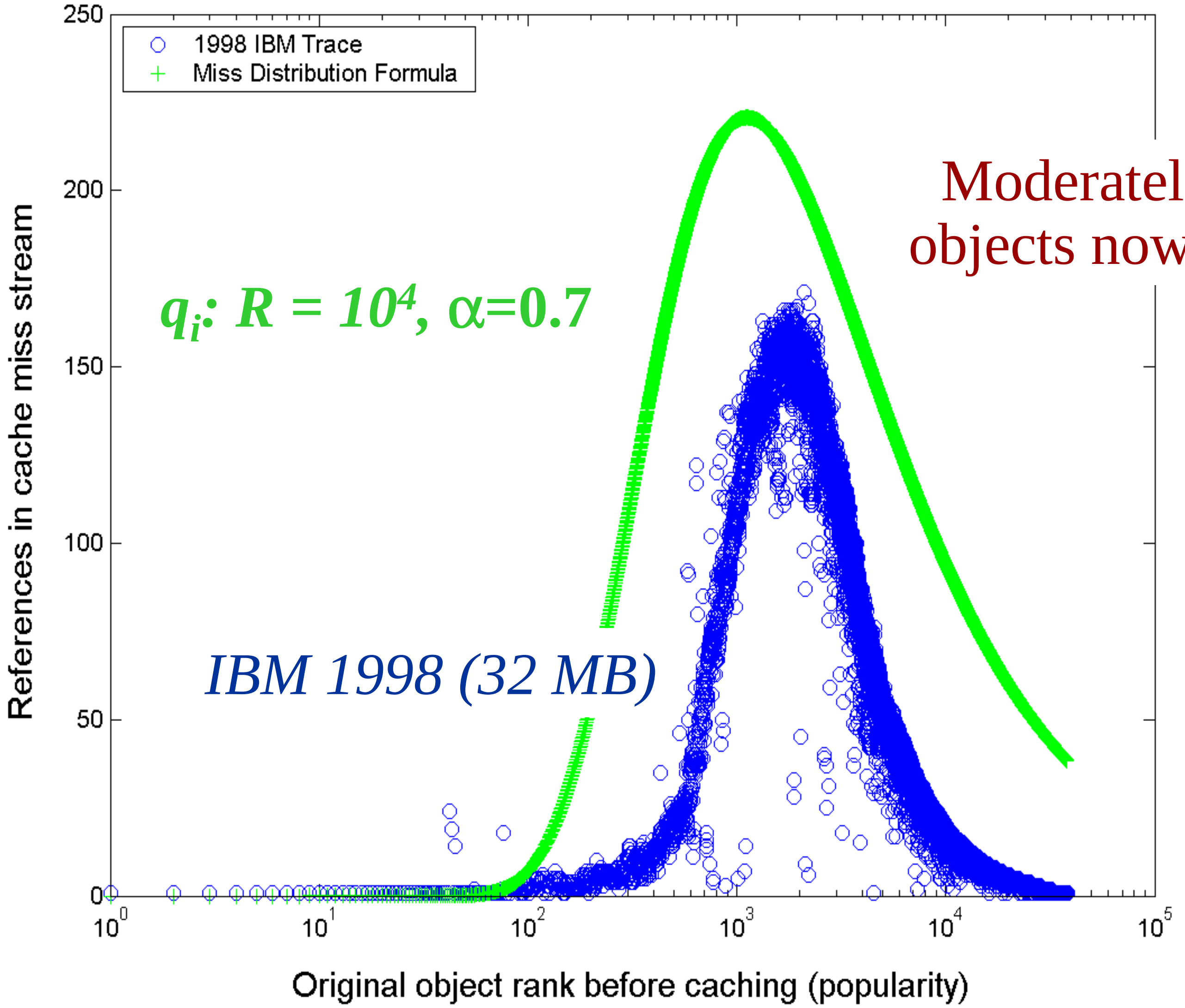
…**and** *i* has not been referenced in the last *R* requests.

probability $(1 - p_i)^R$

Stack distance

P(a miss is to object i) is $q_i = p_i(1 - p_i)^R$

Miss Stream Probability by Popularity

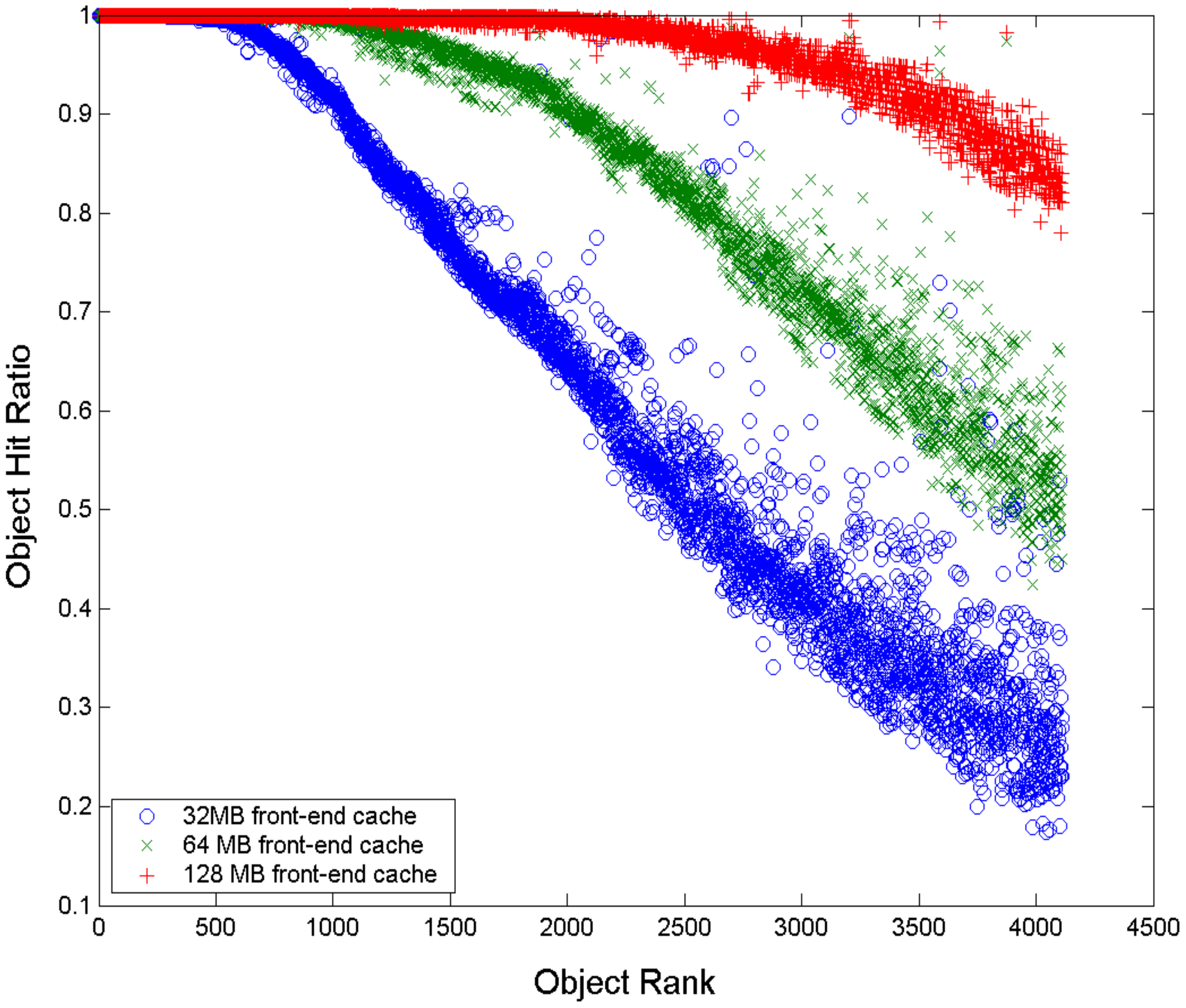


q_i: $R = 10^4$, $\alpha=0.7$

IBM 1998 (32 MB)

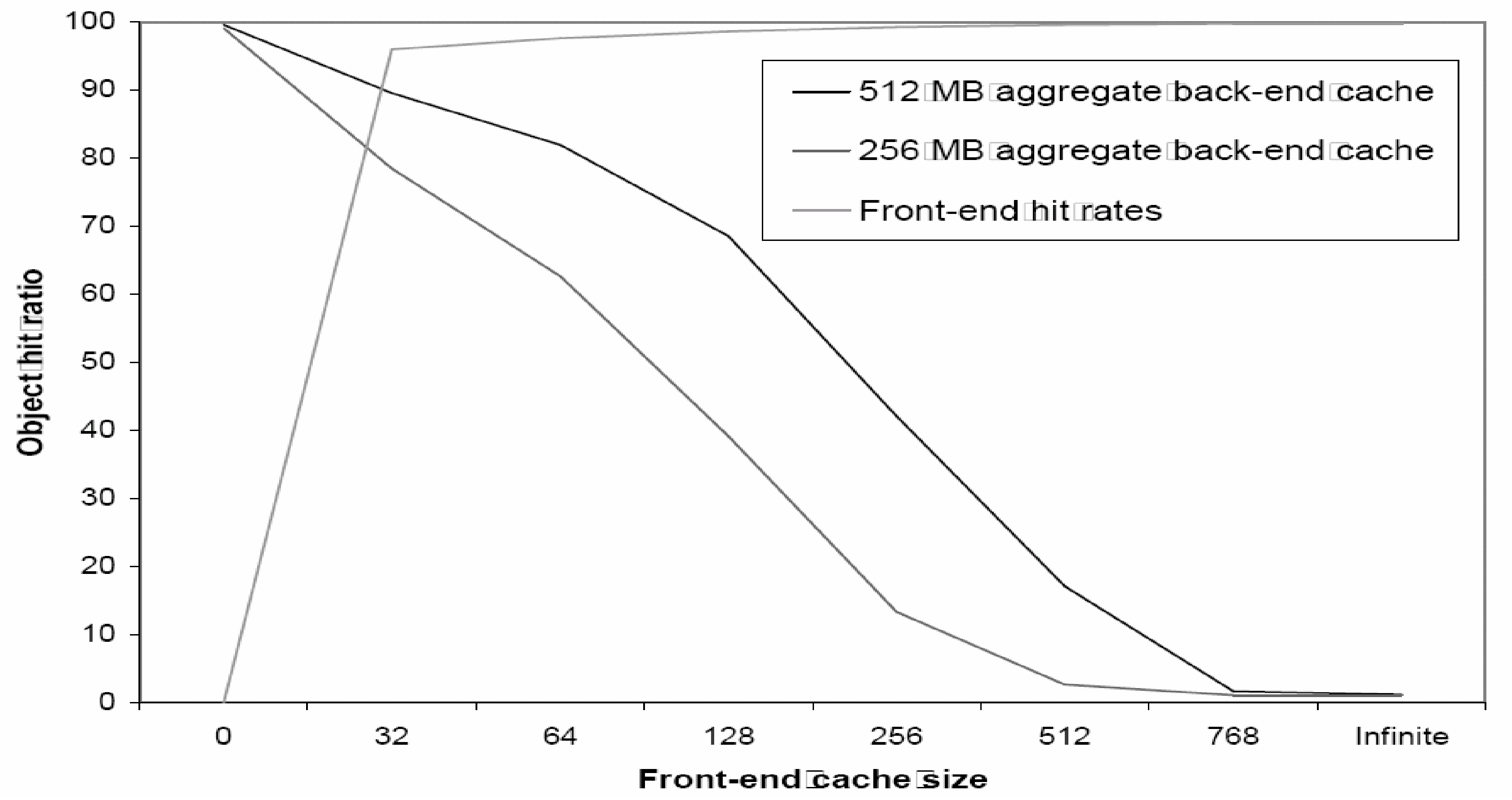
Moderately popular objects now dominate.

Object Hit Ratio by Popularity



IBM
1998

Effects on server locality

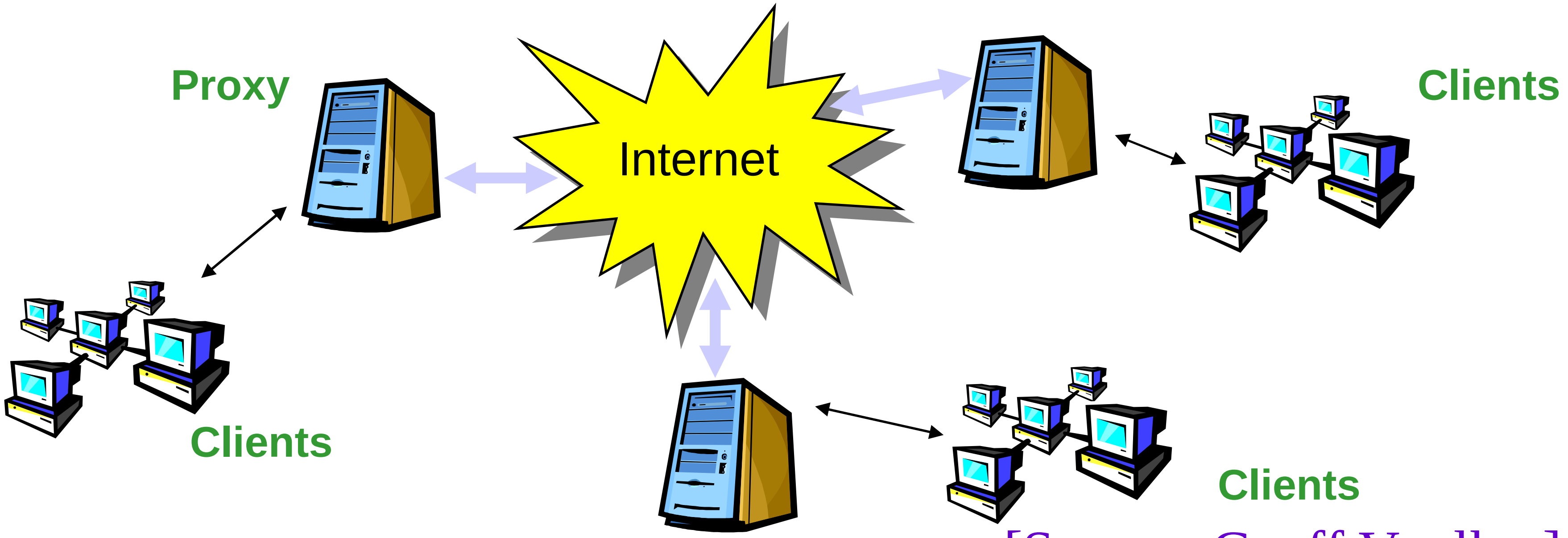


Cache Effectiveness

- Previous work has shown that hit rate increases with population size
- However, single proxy caches have practical limits
  - Load, network topology, organizational constraints
- One technique to scale the client population is to have proxy caches cooperate

Cooperative Web Proxy Caching

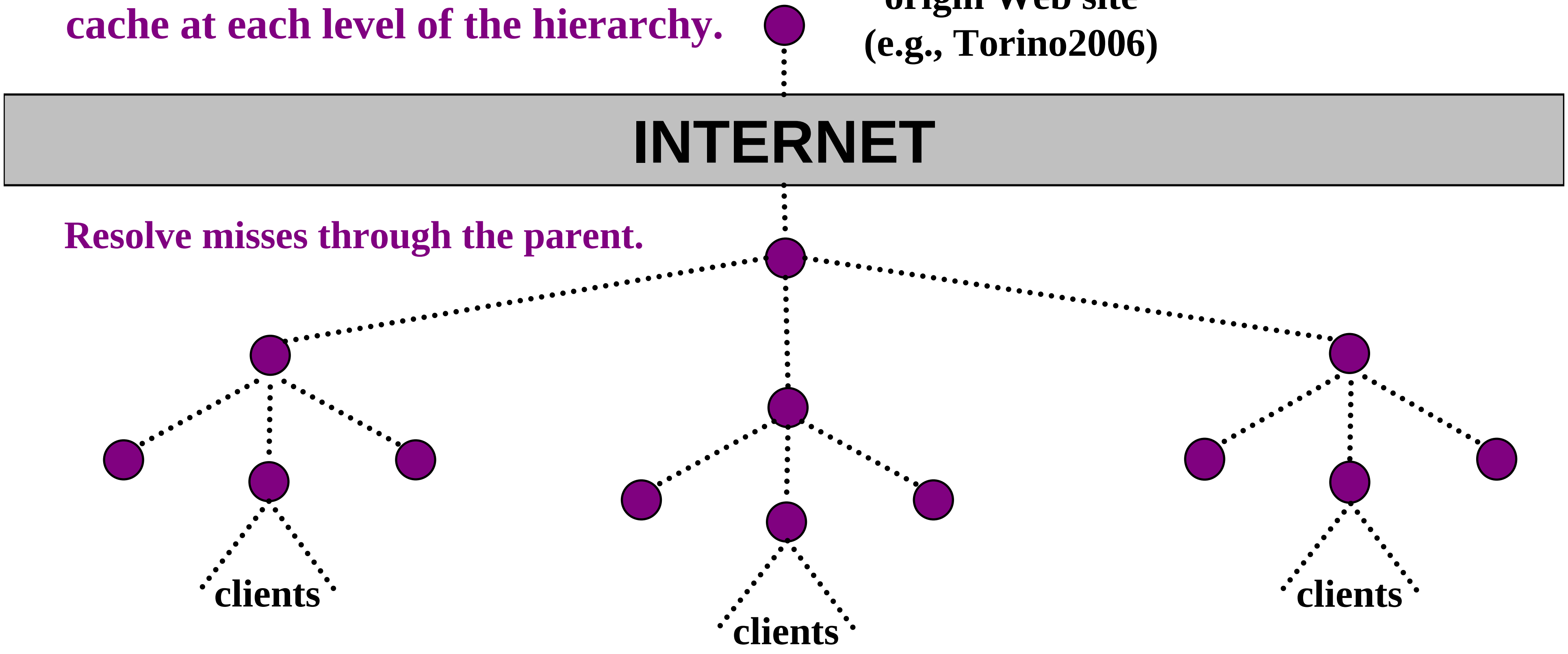
- Sharing and/or coordination of cache state among multiple Web proxy cache nodes
- Effectiveness of proxy cooperation depends on:
  - Inter-proxy communication distance
  - Size of client population served
  - Proxy utilization and load balance



[Source: Geoff Voelker]

Hierarchical Caches

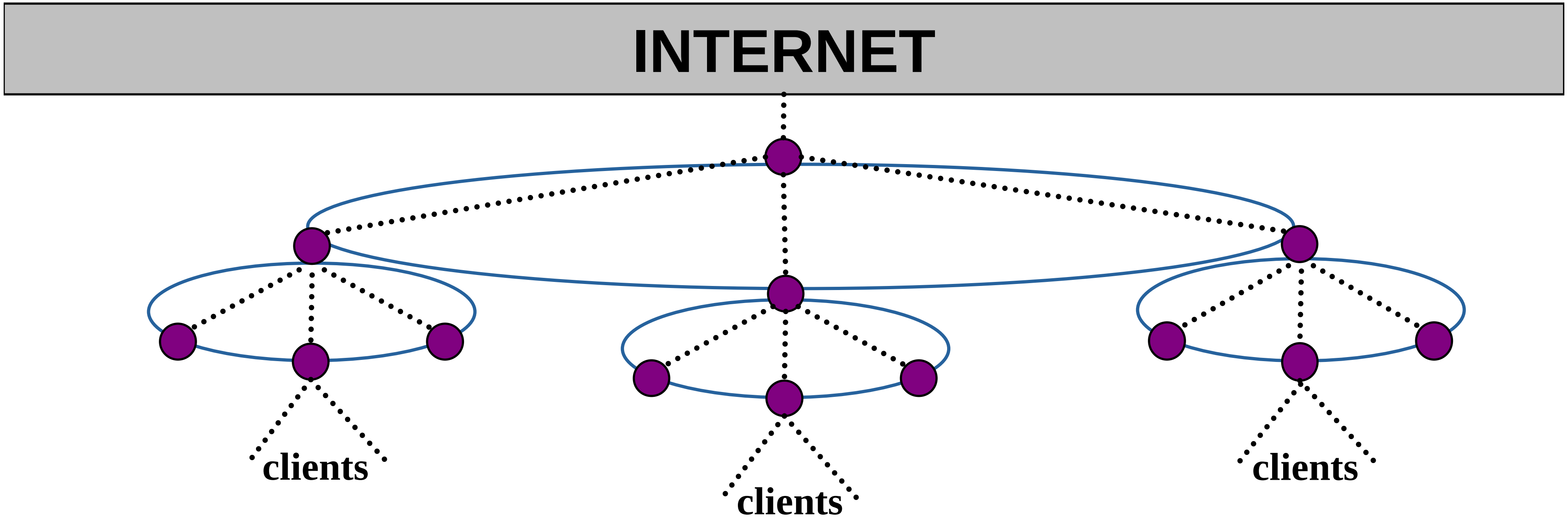
Idea: place caches at exchange or switching points in the network, and cache at each level of the hierarchy.



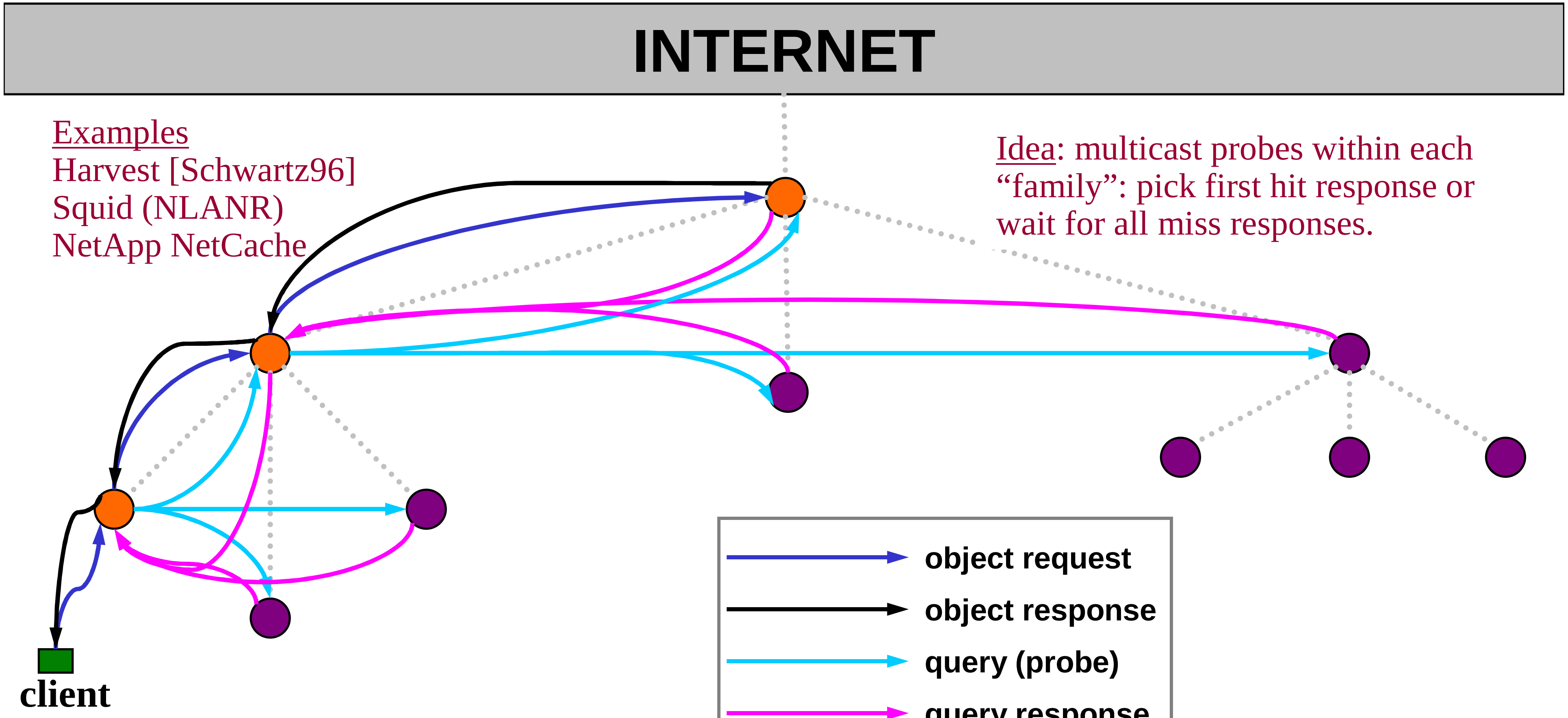
Resolve misses through the parent.

Content-Sharing Among Peers

Idea: **Since siblings are "close" in the network, allow them to share their cache contents directly.**



Harvest-Style ICP Hierarchies



Examples
Harvest [Schwartz96]
Squid (NLANR)
NetApp NetCache

Idea: multicast probes within each "family": pick first hit response or wait for all miss responses.

Issues for Cache Hierarchies

- With ICP: query traffic within “families” (size *n*)
  - Inter-sibling ICP traffic (and aggregate overhead) is quadratic with *n*.
  - Query-handling overhead grows linearly with *n*.
- miss latency
  - Object passes through every cache from origin to client: deeper hierarchies scale better, but impose higher latencies.
- storage
  - A recently-fetched object is replicated at every level of the tree.
- effectiveness
  - Interior cache benefits are limited by capacity if objects are not likely to live there long (e.g., LRU).

A Multi-Organization Trace

- University of Washington (UW) is a large and diverse client population
    - Approximately 50K people
- UW client population contains 200 independent campus organizations
    - Museums of Art and Natural History
    - Schools of Medicine, Dentistry, Nursing
    - Departments of Computer Science, History, and Music
- A trace of UW is effectively a simultaneous trace of 200 diverse client organizations
    - Key: Tagged clients according to their organization in trace

[Source: Geoff Voelker]

Cooperation Across Organizations

- Treat each UW organization as an independent “company”
- Evaluate cooperative caching among these organizations

- How much Web document reuse is there among these organizations?
  - Place a proxy cache in front of each organization.
  - What is the benefit of cooperative caching among these 200 proxies?

[Source: Geoff Voelker]

UW Trace Characteristics

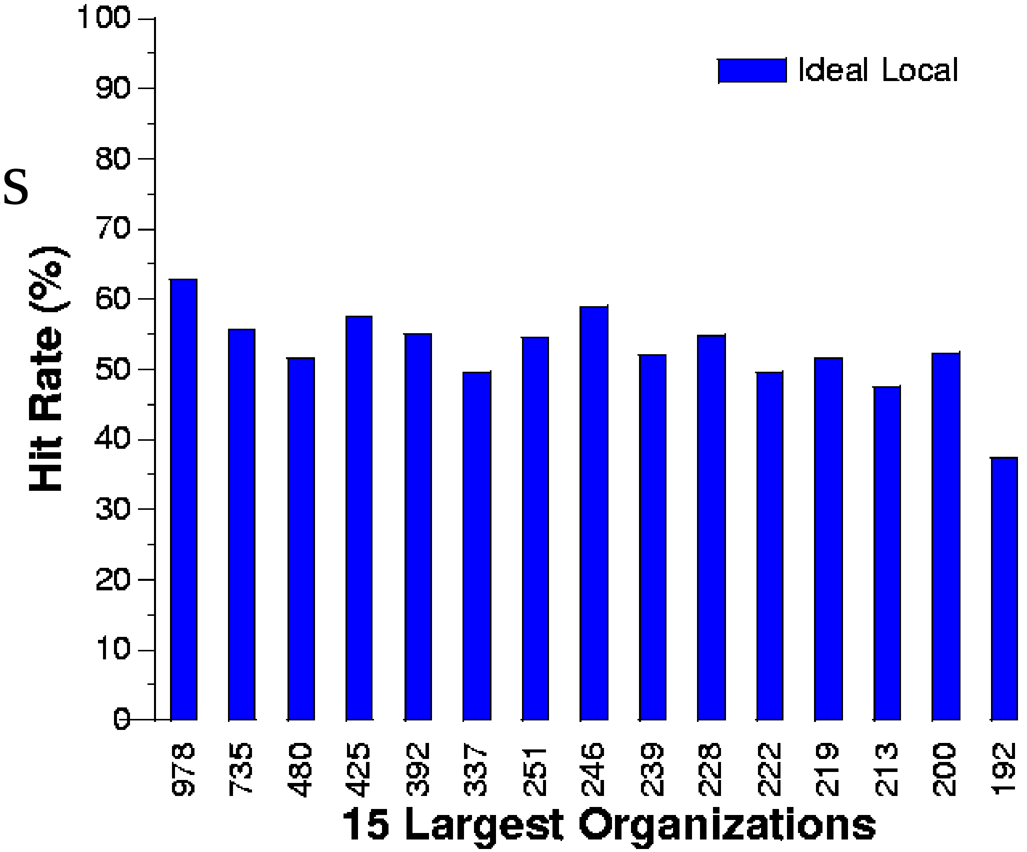
Trace	**UW**
Duration	7 days
HTTP objects	18.4 million
HTTP requests	82.8 million
Avg. requests/sec	137
Total Bytes	677 GB
Servers	244,211
Clients	22,984

[Source: Geoff Voelker]

Ideal Hit Rates for UW proxies

- Ideal hit rate - infinite storage, ignore cacheability, expirations

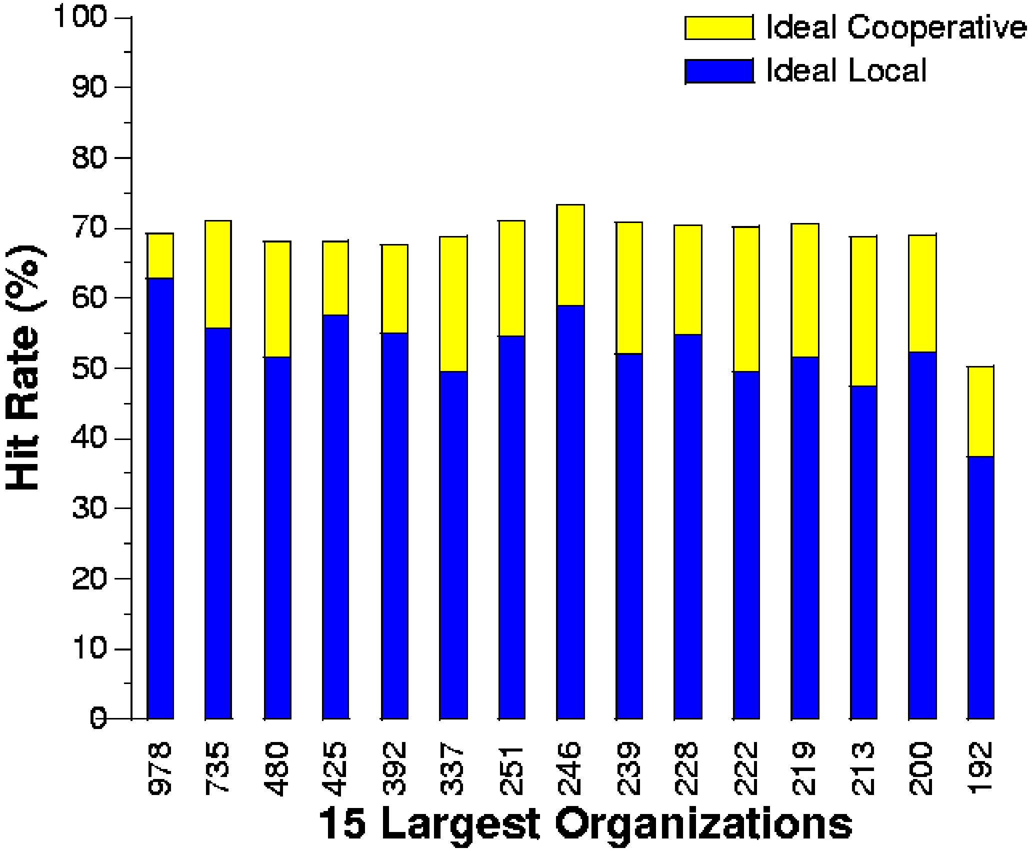
- Average ideal local hit rate: 43%



[Source: Geoff Voelker]

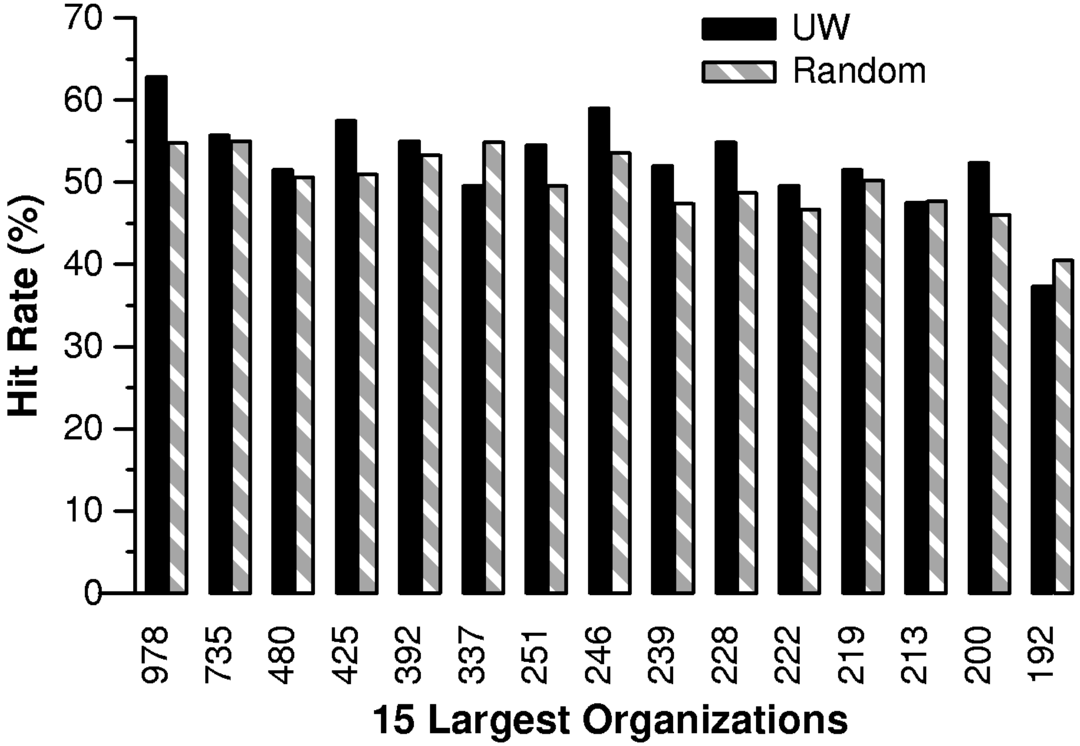
Ideal Hit Rates for UW proxies

- Ideal hit rate - infinite storage, ignore cacheability, expirations
- Average ideal local hit rate: 43%
- Explore benefits of perfect cooperation rather than a particular algorithm
- Average ideal hit rate increases from 43% to 69% with cooperative caching



[Source: Geoff Voelker]

Sharing Due to Affiliation

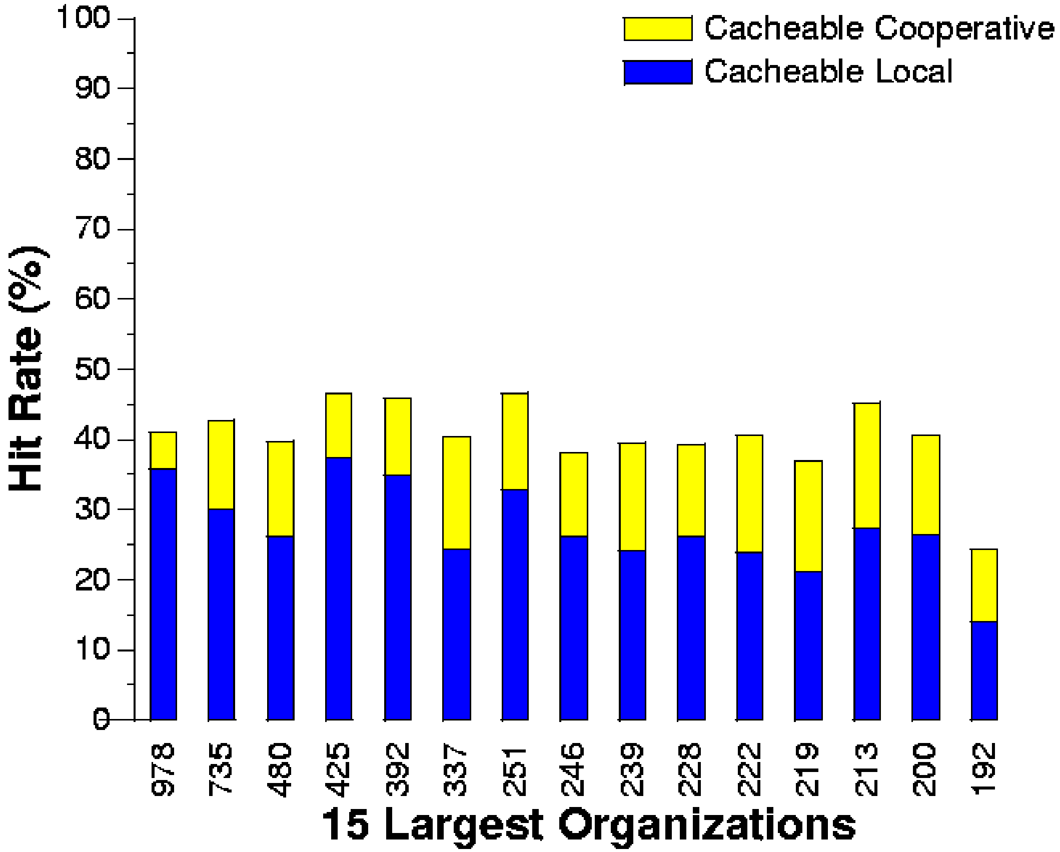


- UW organizational sharing vs. random organizations
- Difference in weighted averages across all orgs is ~5%

[Source: Geoff Voelker]

Cacheable Hit Rates for UW proxies

- Cacheable hit rate - same as ideal, but doesn't ignore cacheability
- Cacheable hit rates are much lower than ideal (average is 20%)
- Average cacheable hit rate increases from 20% to 41% with (perfect) cooperative caching



[Source: Geoff Voelker]

Scaling Cooperative Caching

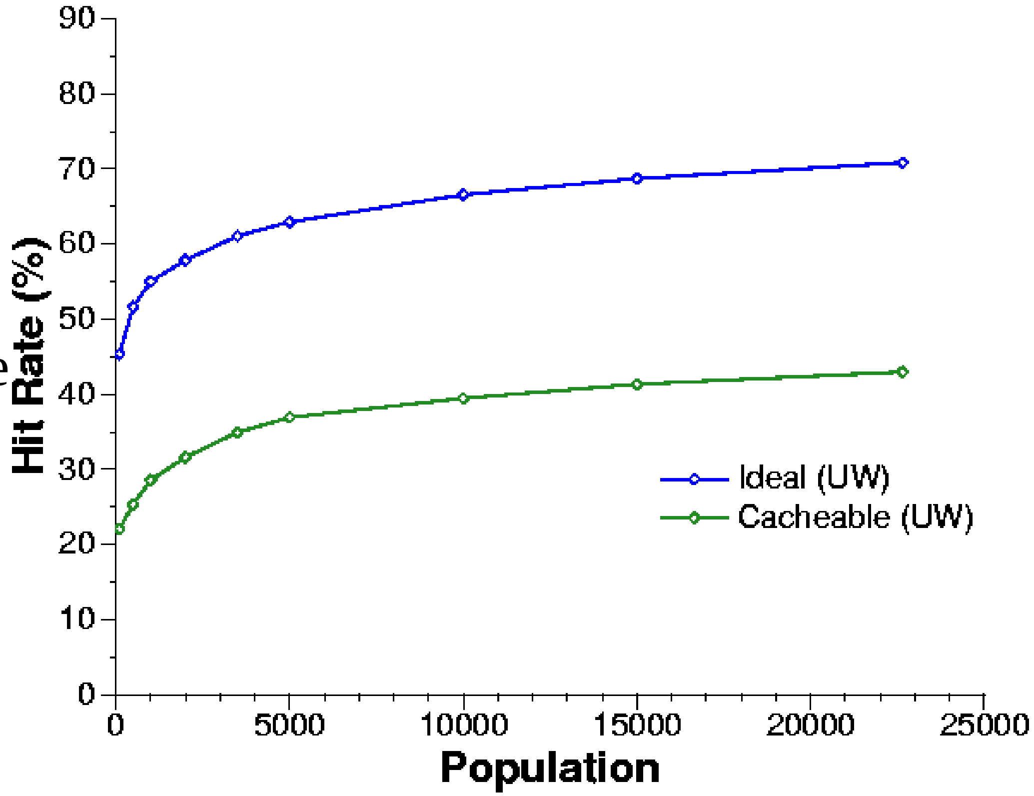
- Organizations of this size can benefit significantly from cooperative caching
- But…we don’t need cooperative caching to handle the entire UW population size
  - A single proxy (or small cluster) can handle this entire population!
  - No technical reason to use cooperative caching for this environment
  - In the real world, decisions of proxy placement are often political or geographical

- How effective is cooperative caching at scales where a single cache cannot be used?

[Source: Geoff Voelker]

Hit Rate vs. Client Population

- Curves similar to other studies in the area
- Small organizations
  - Significant increase in hit rate as client population increases
  - The reason why cooperative caching is effective for UW
- Large organizations
  - Marginal increase in hit rate as client population increases



[Source: Geoff Voelker]

Transactional Data Caching

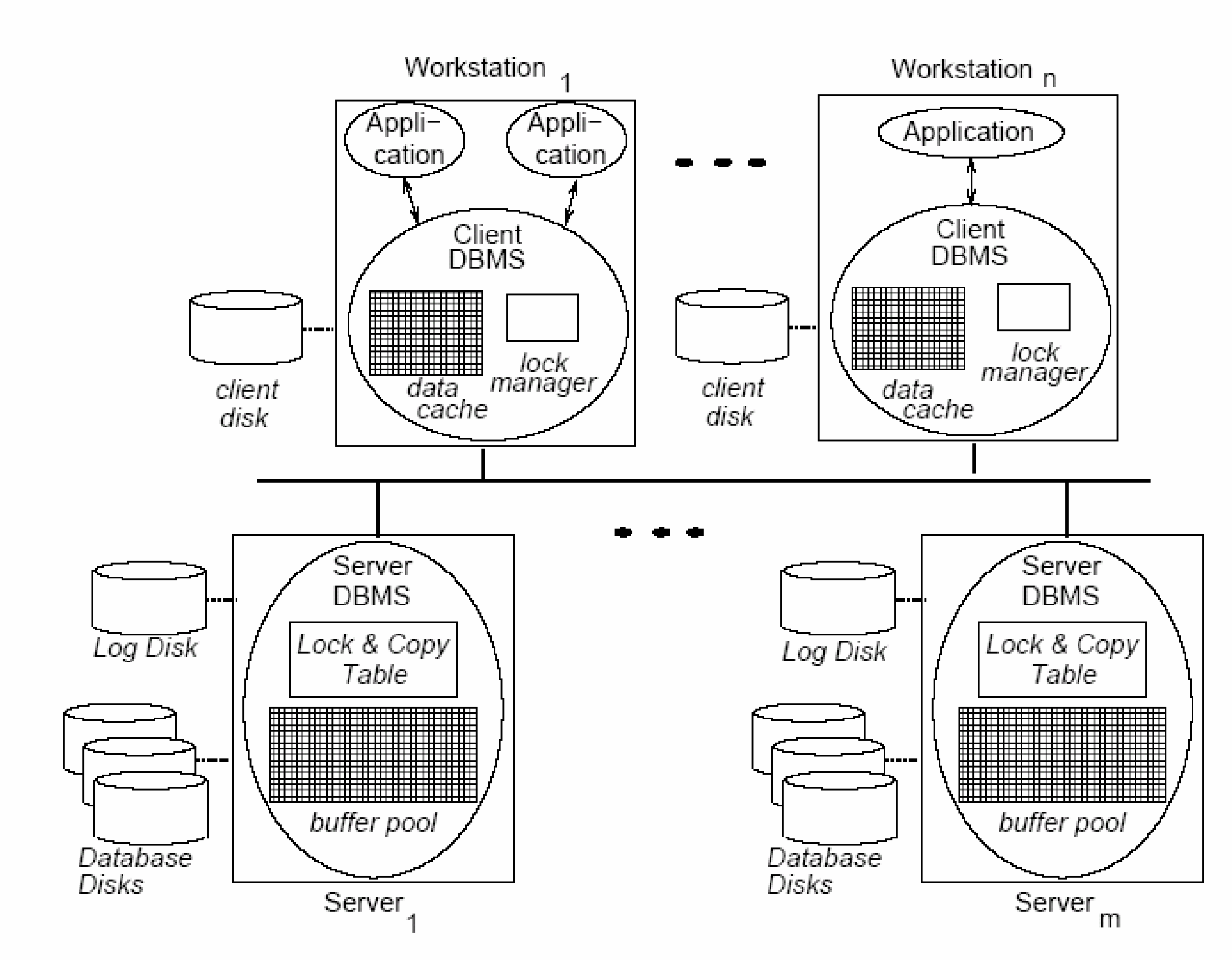
Client-Server Database System Architectures

- query-shipping model
  - – clients send queries (plain SQL text/compiled)
  - – server sends results set
  - \+ simple: lightweight clients, no change to the server DBMS engine
  - \- underutilization of client resources/bottleneck at the server
- data-shipping model
  - – clients request specific data items
  - – query processing takes place at the client side
  - \+ data closer to applications (no need for stored proc.)
  - \+ offload of server DBMS
  - \- higher complexity of client DBMS

inter vs *intra* Transaction Caching

- intra transaction caching
    - – data is retained within the cache only for the duration of the transaction
    - + simple: just manage local page buffer and corresponding locks
    - - requires access to server DBMS at every transaction
- inter transaction caching ←
    - – data is retained within the cache even after termination of the transaction that originally shipped in the data.
    - + load pressure relief at the server DBMS
    - - need for consistency management scheme ensuring serializable view of the database

Reference Architecture



Motivations

- Servers have typically larger capacity than single workstations…
- but clients have more aggregated capacity!
- Avoiding client/server communication:
  - improved latency
  - reduce b/w consumption
  - allow access to data independently of server load: higher performance predictability

Consistency requirements

- Need support for ACID Transactions, including serializability…
- we're in a replicated environment:

“one-copy serializability”

- equivalent to some serial execution on a non-replicated database

Availability

- Strong physical and environmental asymettries between clients and servers:
  - Servers usually have more reliable hw
  - Clients may frequently (explicitly or not) disconnect

- Clients crash or disconnection must not impact availability of data.

Client Caching:
Dynamic Replication + Second Class Ownership

Dynamic replication

- Page copies a created and destroyed based on runtime client demands.
- Finite cache capacity : page eviction policy

Second Class Ownership

- (Consistent) replication can hamper availability in presence of failures
- Client-cache d pages can be destroyed at any time without causing the loss of committed updates:
  - A server can consider a client “crashed” at any time and unilaterally abort any active transaction
  - Servers can’t be hijacked by uncooperative (crashed) clients

Cost Factors

- Consistency enforcing algorithms can be much more complex than those employed for WWW objects caching
- Cost Factors:
  - Overhead for control actions
  - Synchronous vs Asynchronous control actions
  - Transaction blocking vs aborts
  - Effective client cache utilization
- Note that the impact of these factors is workload-dependent:
  - Need for general-purpose solutions

A Taxonomy of Algorithms
Detection- vs Avoidance-based

Detection-based

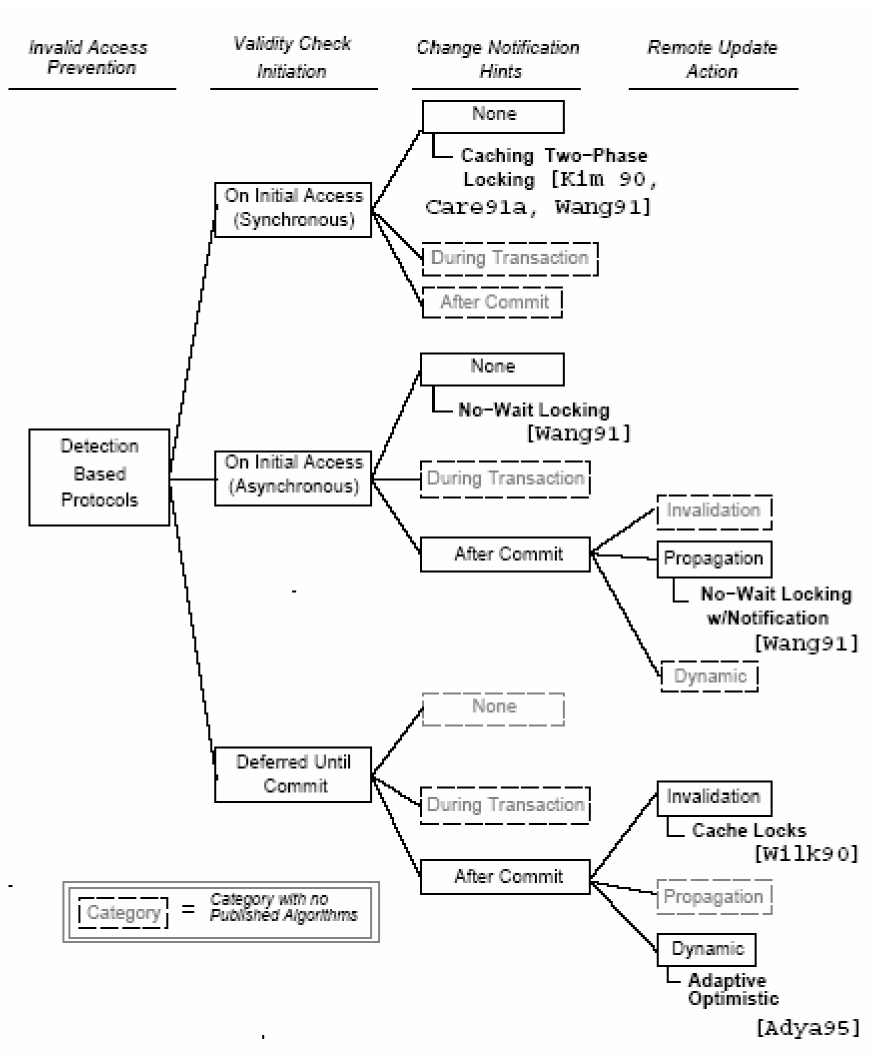
- Stale data is allowed to remain in client caches, but transactions that are allowed to commit have not accessed stale data
- Stale data = older than latest committed value
- LAZY Approach:
  - require transaction validity check
  - asynch update notifications (hints)

Avoidance-based

- All cached data is valid (no staleness)
- EAGER Approach:
  - Invalid data is atomically removed from client caches
  - Read-one / Write-all, just evict any unavailable cache

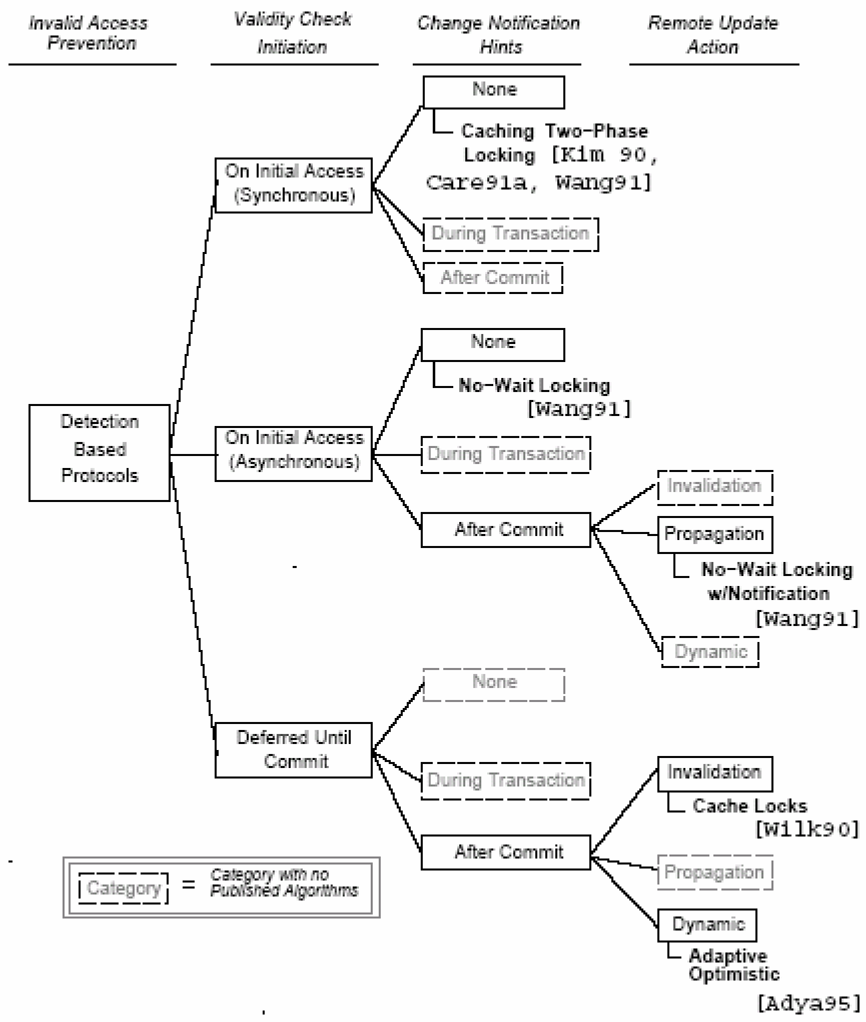
Taxonomy:
Detection-based Algorithms

- Stale data is allowed to remain in client caches, but transactions that are allowed to commit have not accessed stale data
- Stale data = older than latest committed value
- LAZY Approaches:
  - require transaction validity check
  - asynch update notifications (hints)



Taxonomy:
Detection-based Algorithms

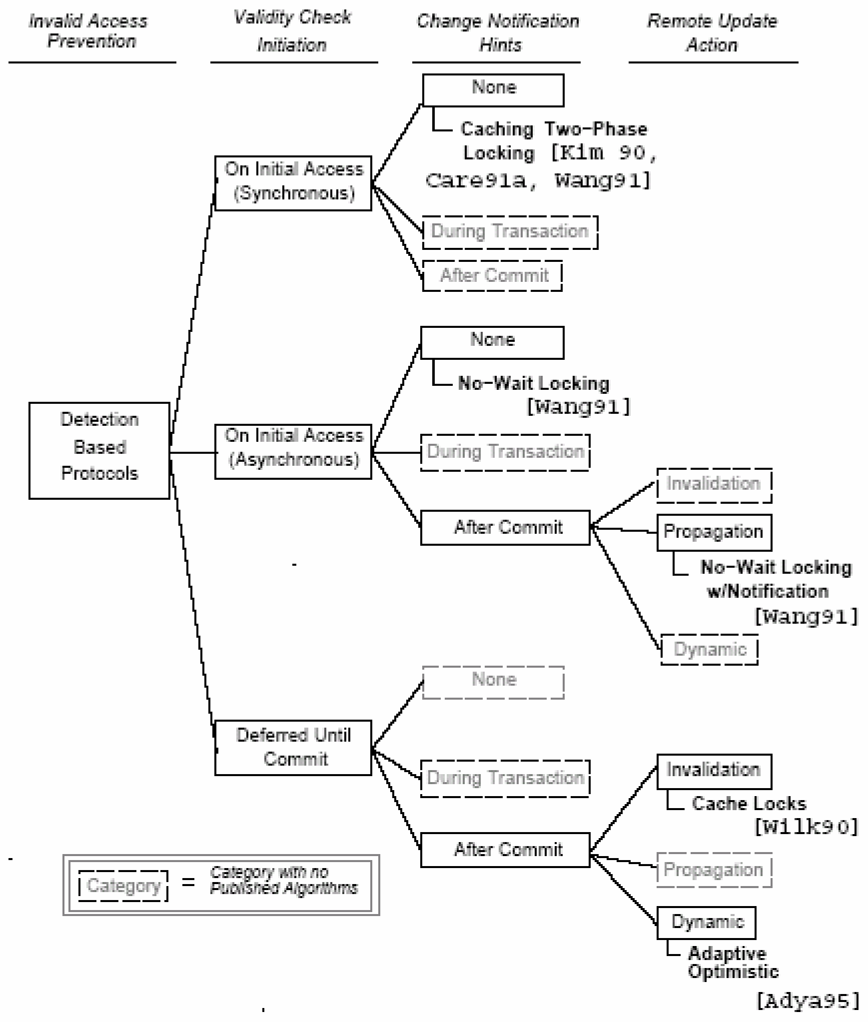
- Simple Clients:
  - No strict need for server's callbacks
- Greater dependency on servers:
  - Overhead



Taxonomy:
Detection-based Algorithms

Validity Check Initiation

- Once validity is established it’s guaranteed for the transaction duration:
  - Until this does not commit/abort, no other transaction can commit updates
  - Before committing any transaction must obtain server permission!
- Synch:
  - Upon first access to a data item
  - No access until validity verification
- Asynch:
  - No wait for validity verification
- Deferred:
  - Even more optimistic!

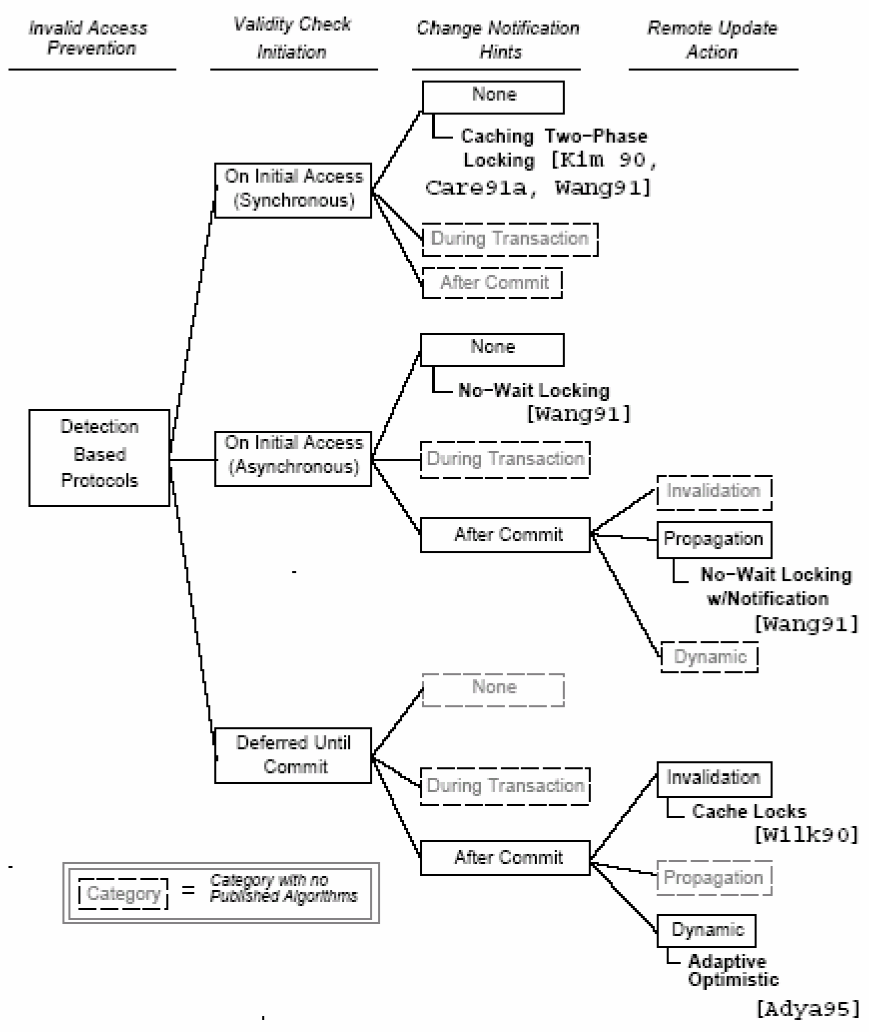


Taxonomy:
Detection-based Algorithms

Validity Check Initiation, Tradeoffs:

+ Deferring allows bundling control operations:
  < overhead

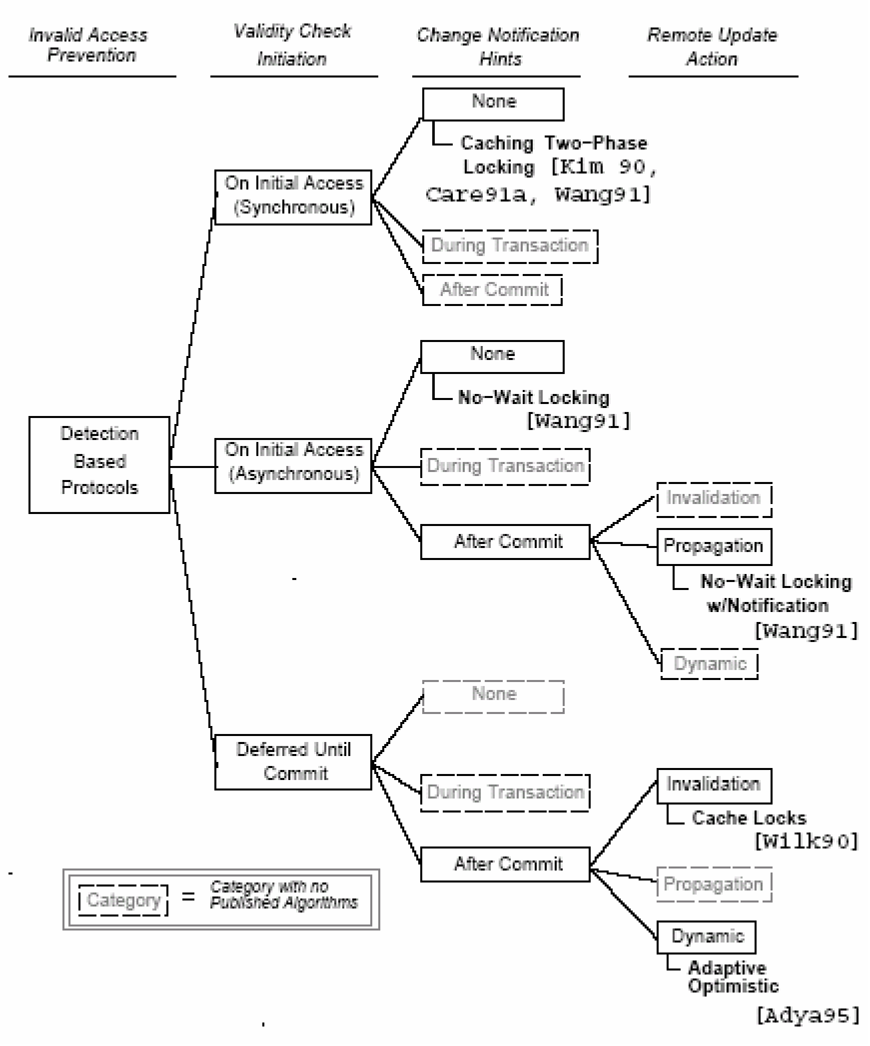
- Late conflict detection can cause late abort of one or more transactions:
  - Possibly requiring duplicate work in interactive environments



Taxonomy: Detection-based Algorithms

Change Notification Hints

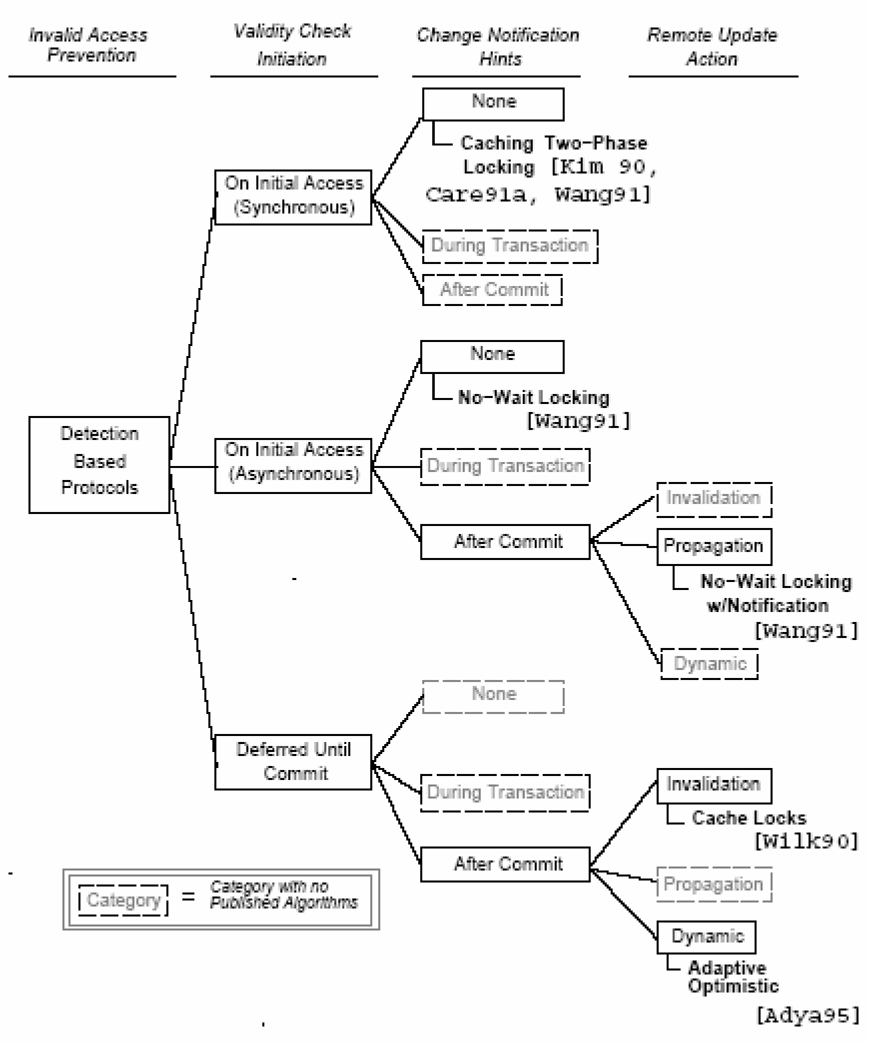
- Idea: reducing the abort rate by spreading updates
- A transaction can send notification hints before or after commit time:
  - If done before & then transaction aborts we get cascading aborts/unnecessary aborts at the other clients!
  - So it's typically done after commit…



Taxonomy:
Detection-based Algorithms

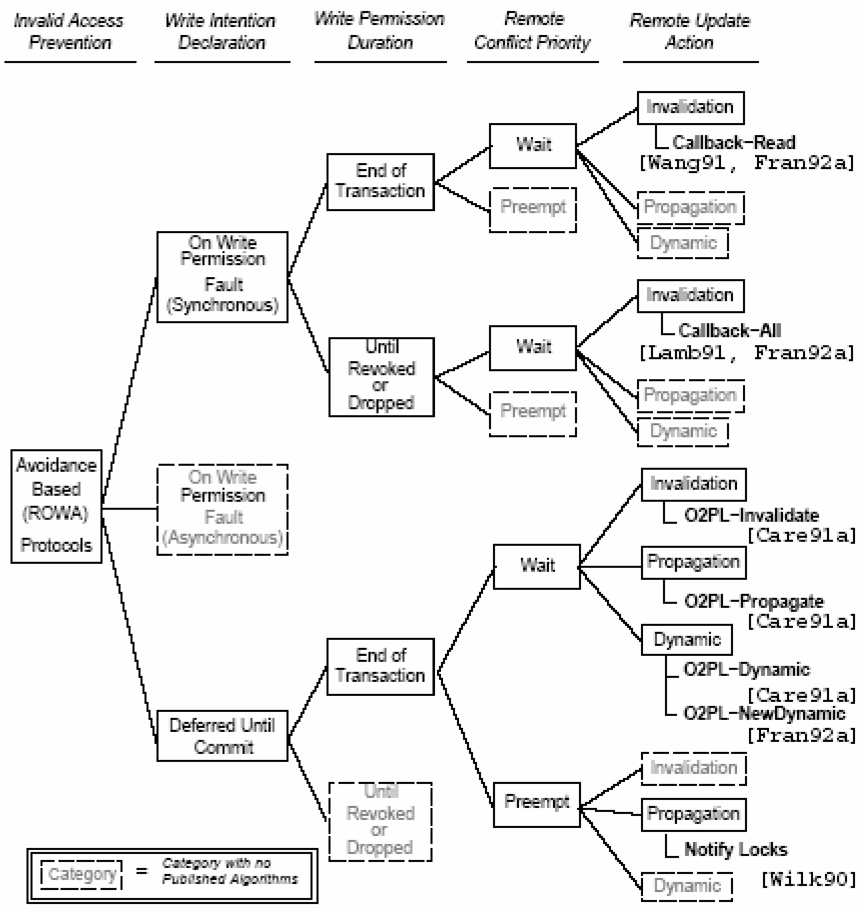
Remote Update Action

- Propagation:
  - Update installation at remote site
- Invalidation:
  - Page eviction at remote site
- Dynamic:
  - Adapt between two depending on perceived workload



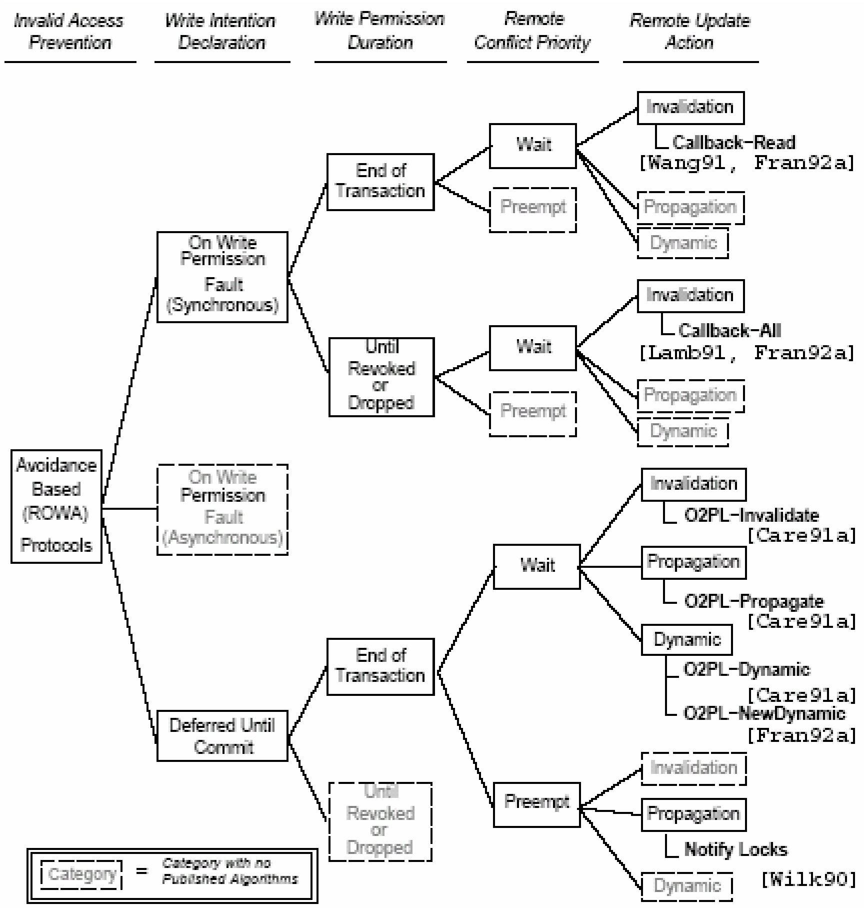
Taxonomy: Avoidance-based Algorithms

- All cached data is valid (no staleness)
- Eager approach:
  - Invalid data is atomically removed from client caches
  - Read-one / Write-all, just evict any unavailable cache



Taxonomy: Avoidance-based Algorithms

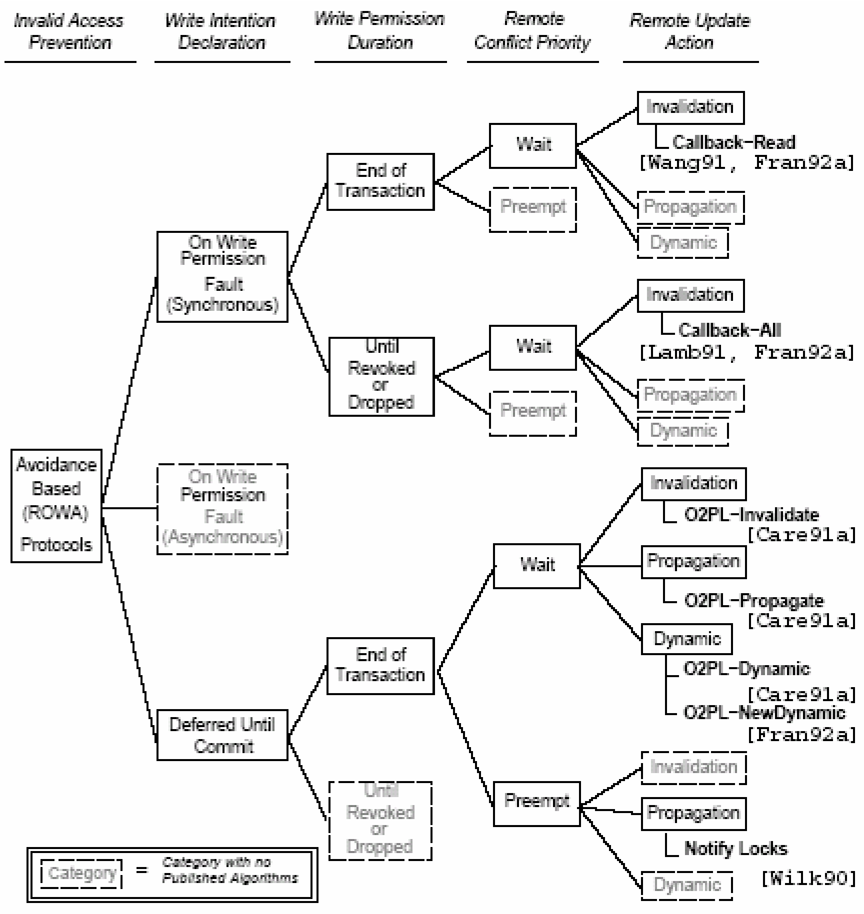
- More complex client caches (e.g., fully-fledge lock manager) vs reduced reliance on server
- More information on the server:
  - ROWA, requires ability to track location of page copies:
    - Broadcast-based
      - good performance
      - low scalability
    - Directory-based
      - higher overhead
      - higher scalability



Taxonomy:
Avoidance-based Algorithms

Write Intention Declaration

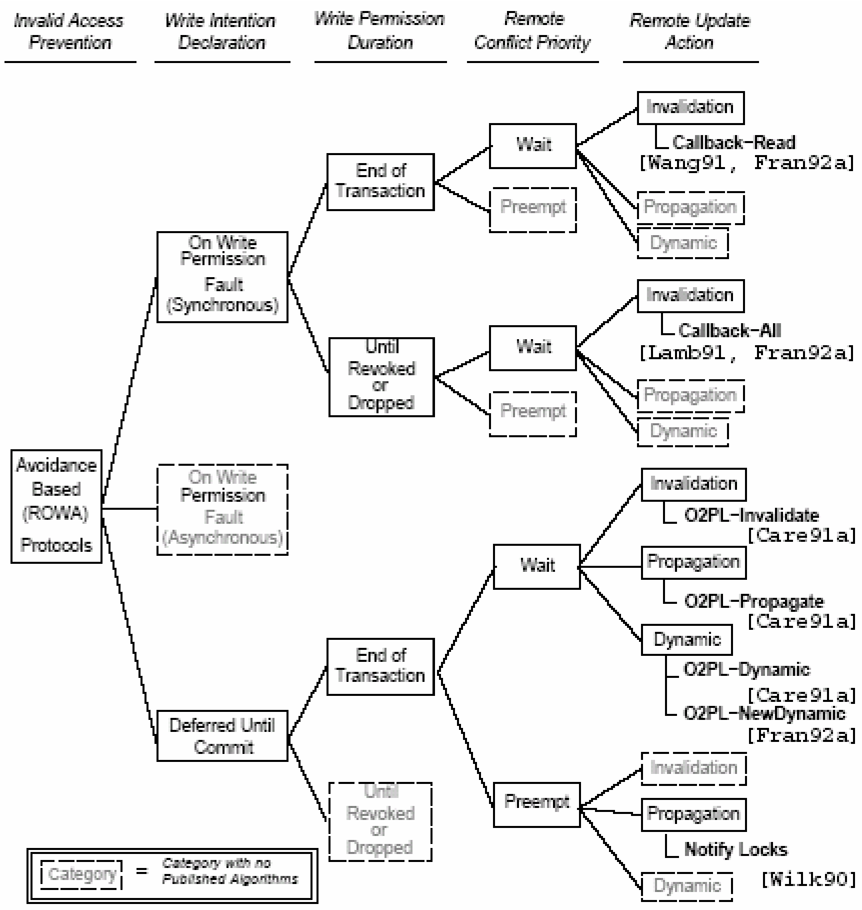
- Reads are alway valid (ROWA)
  - Interactions with server only for pages retrievals and updates
  - Upon page retrieval, the server implicitly guarantees it will inform the client if the page becomes invalid
- If a transaction wishes to update a cached page copy the server must be informed:
  - Write permission must be explicitly granted
  - Once a write permission is granted, data can be updated without contacting the server



Taxonomy: Avoidance-based Algorithms

Write Intention Declaration

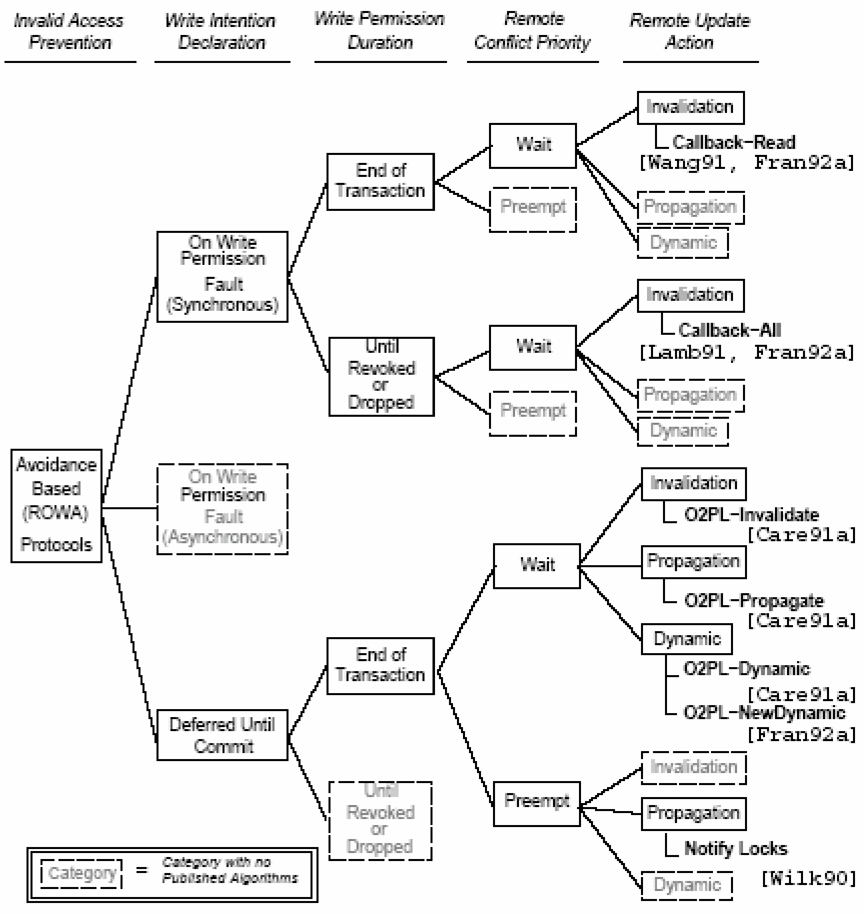
- Write permissions are similar to write locks, but:
  - Are granted to a client site not to a single transaction
  - Doesn’t obey two-phase constraint.
- Such algorithms require costly interactions with remote clients to grant write permissions!
- Three level of optimism:
  - Synch, pessimistic
  - At commit time (unless page has to be evicted before), optimistic
  - Asynch, in the middle…



Taxonomy: Avoidance-based Algorithms

Write Permission Duration

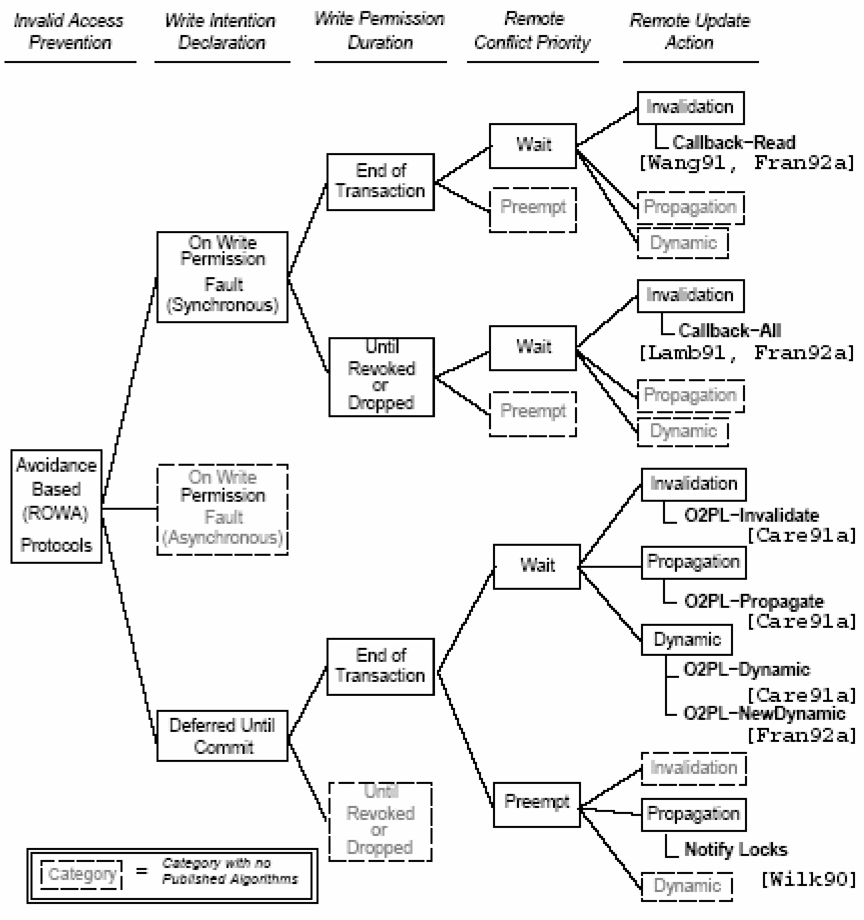
- How long should the write permission be retained for?
  - Single Transaction:
    - all page update intentions must be declared
  - Across transaction boundaries:
    - Until the page is evicted from the cache (due to the replacement algorithm)
    - Untill the server does not drop the permission due to consistency actions



Taxonomy: Avoidance-based Algorithms

Remote Conflict Priority

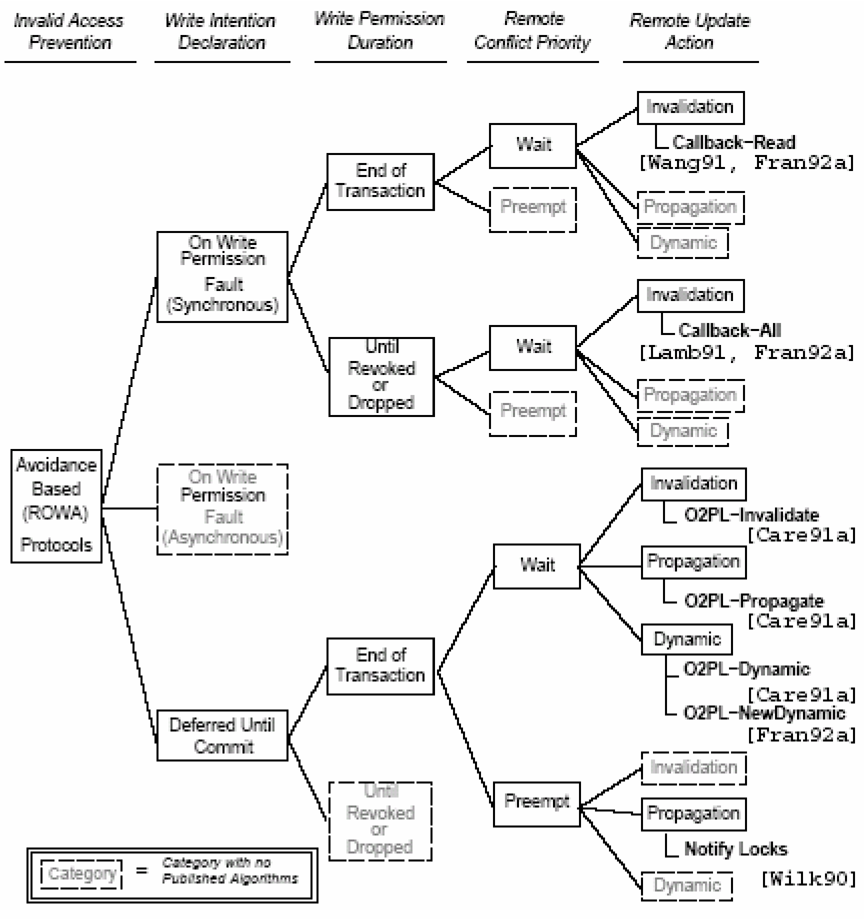
- What if the page is currently being used by a remote client?
  - Wait until transaction completes:
    - Priority to readers
  - Preempt (abort) remote transaction:
    - Priority to writers



Taxonomy: Avoidance-based Algorithms

Remote Update Action

- Similar to Detection-based but with a remarkable difference:
  - Remote update actions must be completed before the local transaction commits for the ROWA scheme:
    - Two-phase commit (2PC) is required for propagation
    - No need for 2PC when using invalidation



Server-based Two Phase Locking (S2PL)

- Detection based with synch page validation upon initial access
- Based on primary-copy replication scheme:
  - Before commit, a transaction must first access a designated (primary) copty of any page it reads or writes:
    - Reads must have the same value
    - Writes must be installed at the primary copy
- Variants:
  - Caching 2PL
  - Basic 2PL

Caching 2PL (C2PL)

- “check-on-access” policy
- Page copies are tagged with a version identifier
- Page lock requests are synch. sent to the server (along with version ids if already in cache):
  - Centralized strict 2PL Lock Management & Deadlock Management
  - Upon read-lock request, a valid page is returned if necessary
  - Inter-transaction caching enabled
- Basic 2PL:
  - Just like C2PL, but only intra-transaction caching:
    - cached pages are purged upon transaction termination

Callback Locking
(CB)

- Avoidance-based, synchronous write intention declaration:
  - Local cached pages are always valid
  - No additional consistency controls upon commit
- Clients issue page requests upon cache miss:
  - Server returns a valid copy only if no other client has write permission granted
- Need for server tracking of remote page copies:
  - Clients inform server of eviction using piggybacks:
    - Server has a conservative view of cached pages
- Clients have a local lock manager:
  - Never wait for read lock and wait for write lock only if no write permission

Callback Locking
(CB)

- Write permission request management:
  - Server issues callback requests to other clients holding a copy
- Callbacks are treated as write lock request at the client side + the page is evicted from the buffer (*invalidation*)
- To simplify recovery, updated pages are sent to the server upon commit.
- Two variants:
  - Callback-Read:
    - Write permissions granted for a single transaction
    - Server blocks read requests till the end of the writing transacion, if any
  - Callback-All:
    - Write permissions must be explicitly revoked from the server
    - Server issues downgrade requests if a client has write permission and an other client performs a read request

Optimistic Two Phase Locking (O2PL)

- Avoidance-based, commit deferred write intention declaration

- Clients have a local lock manager:
  - No locks are acquired at the server during transaction execution

- Transaction tentatively update pages in their local cache (unless they have to be evicted)

- At commit time, updated pages are sent to the server

Optimistic Two Phase Locking (O2PL)

- The server acquire write locks on such pages and sends a message to each client holding a page copy

- Remote clients in their turn acquire exclusive locks on their local page copies and update/invalidate them:
  - In case pages are updated we need an extra round (2PC):
    - After the server collects write lock acks (=2PC vote msgs) from ALL the clients, it actually sends the updates.
    - Upon receipt of the updates the client installs them and releases the lock

- Centralized deadlock detection, based on periodic collection of local wait-for graphs