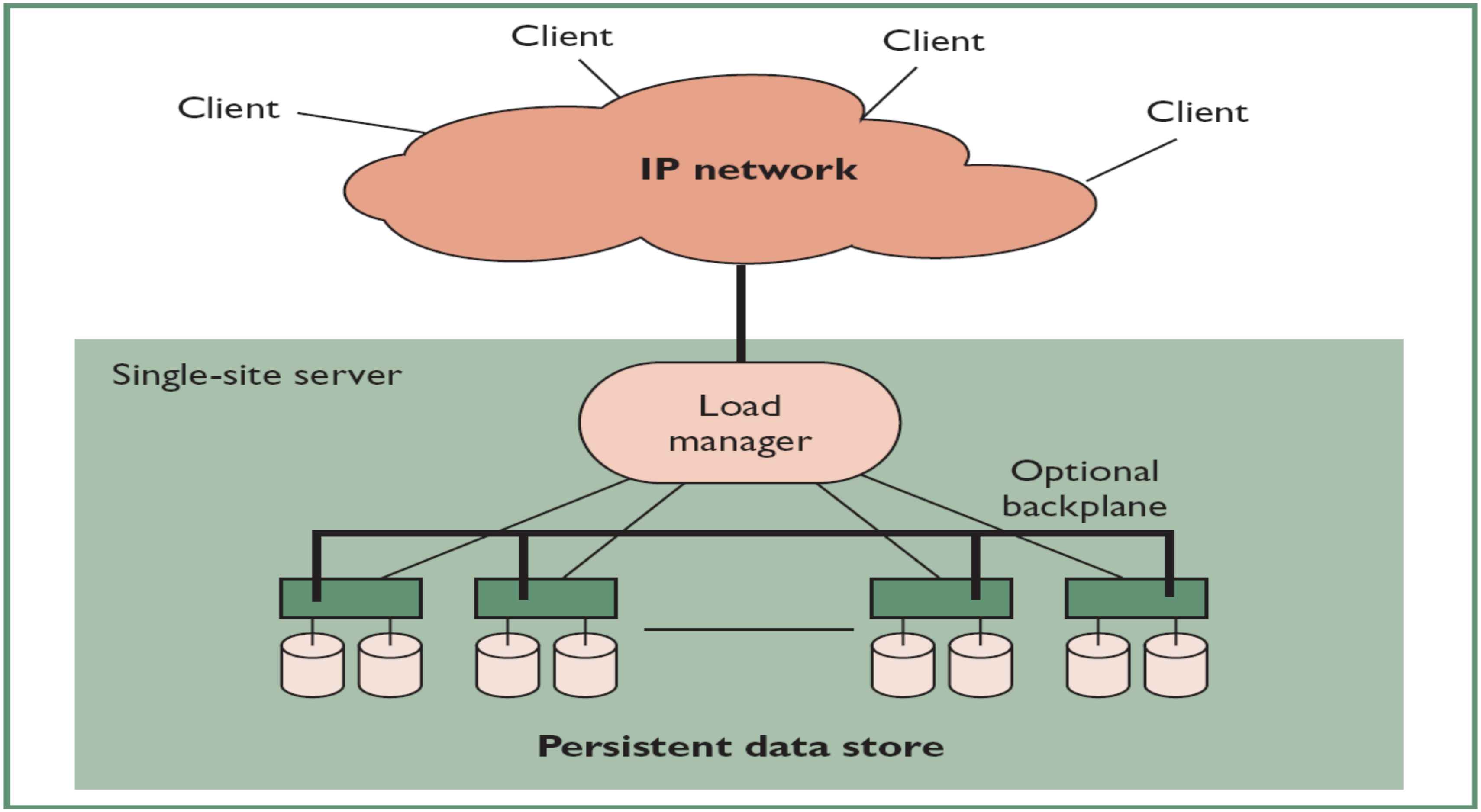
Data Center Performance

George Porter
CSE 124
Feb 11, 2016

*Includes material taken from Barroso et al., 2013, UCSD 222a, and Cedric Lam and Hong Liu (Google)

Part 1: Partitioning work across many servers

Network Service Components



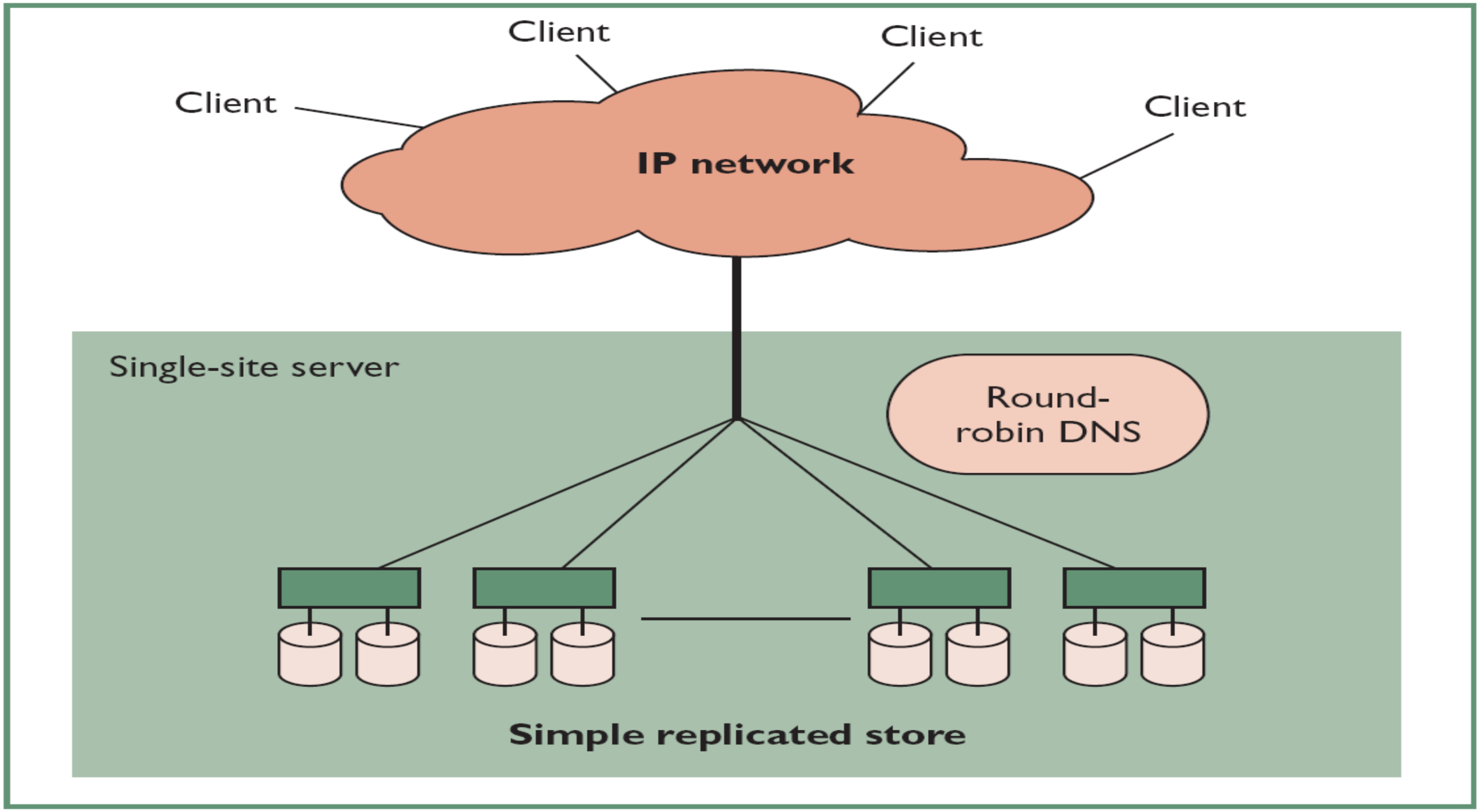
Load Management

- Started with “round-robin” DNS in 1995
  - Map hostname to multiple IP addresses, hand out particular mapping in a round robin fashion to clients
- What is the main limitation of this?
  - A: Does not hide failure or inactive servers
  - B: Can not scale to millions of users
  - C: Exposes structure of underlying service
  - D: A&C
  - E: B&C

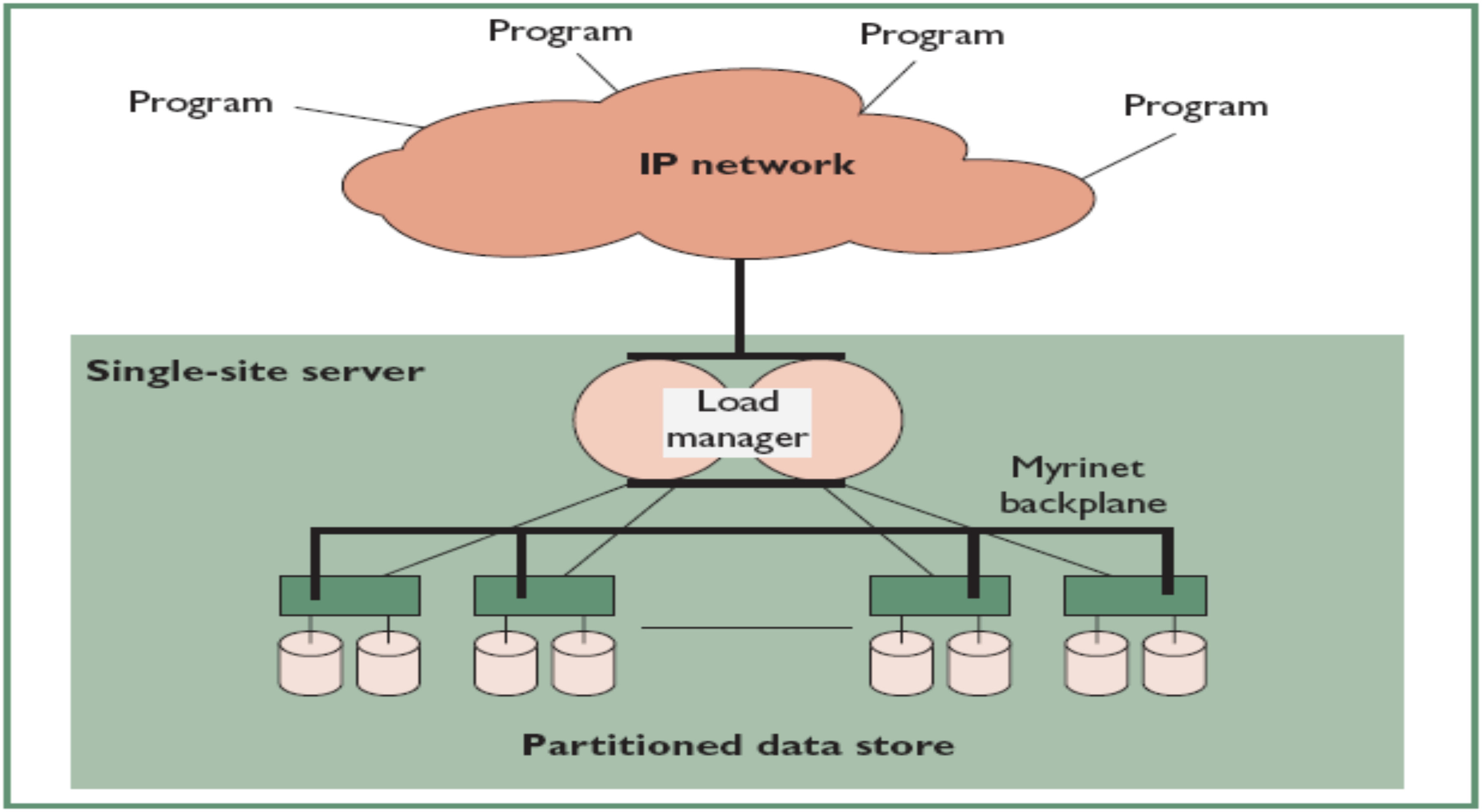
Load Management

- Started with "round-robin" DNS in 1995
  - Map hostname to multiple IP addresses, hand out particular mapping in a round robin fashion to clients
  - Does not hide failure or inactive servers
  - Exposes structure of underlying service
- Today, middleboxes can inspect TCP session state or HTTP session state (e.g., request headers)
  - Perform mapping of requests to back end servers based on dynamically changing membership information
- "Load balancing" still an important topic

Service Replication



Service Partitioning



Case Study: Search

- Map keywords to set of documents containing all words
  - Optionally rank the document set in decreasing relevance
    - E.g., PageRank from Google
- Need a web crawler to build *inverted index*
  - Data structure that maps keywords to list of all documents that contains that word
- Multi-word search
  - Perform *join* operation across individual inverted indices
- Where to store individual inverted indices?
  - Too much storage to place all on each machine (esp if you also need to have portions of the document avail as well)

Case Study: Search

- Vertical partitioning
  - Split inverted index across multiple nodes (each node contains as much of index as possible for a particular keyword)
  - Replicate inverted indices across multiple nodes
  - OK if certain portion of document database not reflected in a particular query result (even expected)

- Horizontal partitioning
  - Each node contains portion of inverted index for *all* keywords (or large fraction)
  - Have to visit every node in system to perform full join

Availability Metrics

- Mean time between failures (MTBF)
- Mean time to repair (MTTR)
- Availability = (MTBF – MTTR)/MTBF
- Example:
  - MTBF = 10 minutes
  - MTTR = 1 minute
  - A = (10 – 1) / 10 = 90% availability
- Can improve availability by increasing MTBF or by reducing MTTR
  - Ideally, systems never fail but much easier to test reduction in MTTR than improvement in MTBF

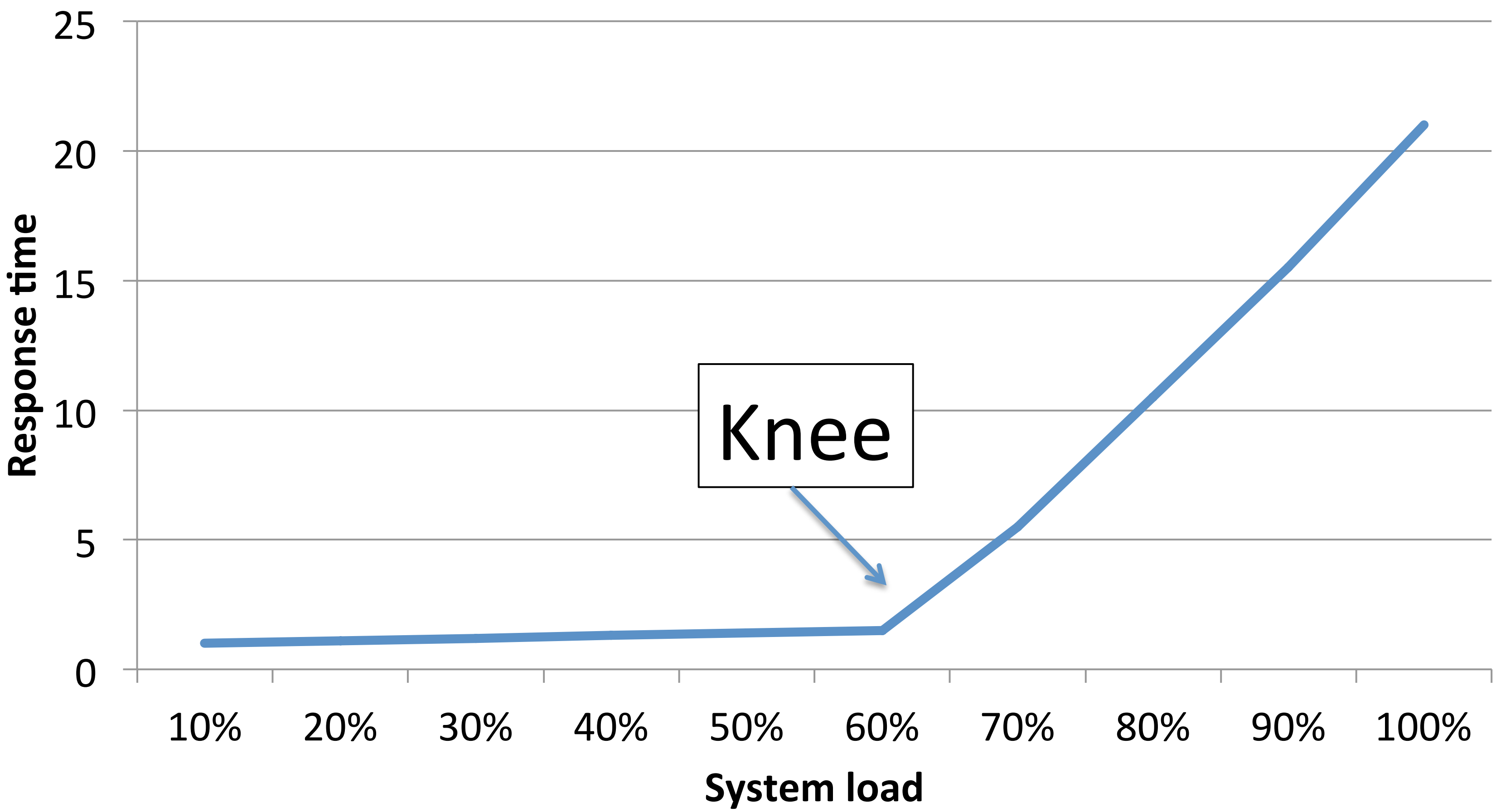
Harvest and Yield

- *yield = queries completed/queries offered*
  - In some sense more interesting than availability because it focuses on client perceptions rather than server perceptions
  - If a service fails when no one was accessing it…
- *harvest = data available/complete data*
  - How much of the database is reflected in each query?
- Should faults affect yield, harvest or both?

DQ Principle

- *Data per query * queries per second* → *constant*
- At high levels of utilization, can increase queries per second by reducing the amount of input for each response
- Adding nodes or software optimizations changes the constant

Performance
"Hockey Stick" graph



Graceful Degradation

- Peak to average ratio of load for giant-scale systems varies from 1.6:1 to 6:1
- Single-event bursts can mean 1 to 3 orders of magnitude increase in load
- Power failures and natural disasters are not independent, severely reducing capacity
- Under heavy load can limit capacity (queries/ sec) to maintain harvest or sacrifice harvest to improve capacity

Graceful Degradation

- Cost-based admission control
  - Search engine denies expensive query (in terms of D)
  - Rejecting one expensive query may allow multiple cheaper ones to complete
- Priority-based admission control
  - Stock trade requests given different priority relative to, e.g., stock quotes
- Reduced data freshness
  - Reduce required data movement under load by allowing certain data to become out of date (again stock quotes or perhaps book inventory)

Online Evolution and Growth

- Internet services undergo rapid development with the frequent release of new products and features
- Rapid release means that software released in unstable state with known bugs
  - Goal: acceptable MTBF, low MTTR, no cascading failures
- Beneficial to have *staging* area such that both new and old system can coexist on a node simultaneously
  - Otherwise, will have to transfer new software after taking down old software → increased MTTR
  - Also makes it easier to switch back to old version in case of trouble

Part 2:
Quantifying performance

In-class activity: The effect of the "long tail"

Reading: "The Tail at Scale"

by Dean and Barroso

Quantifying performance of a cluster

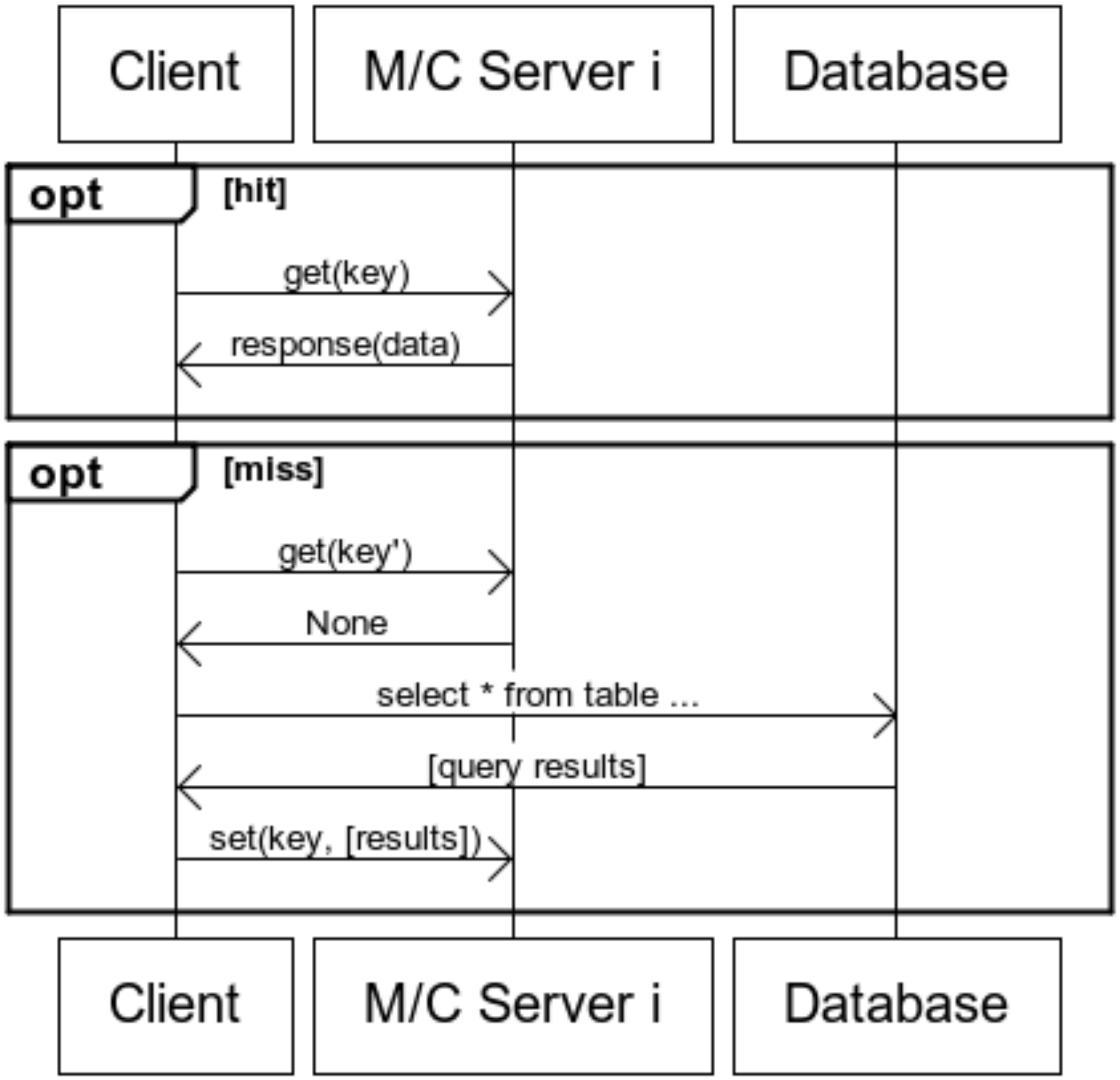
- Typically we think of performance in terms of the mean or median
  - Fine for a single processor/server
  - Not fine for an ensemble of 100s or 1000s of machines
  - Why?

Memcache

- Popular in-memory cache
- Simple get() and put() interface
- Useful for caching popular or expensive requests

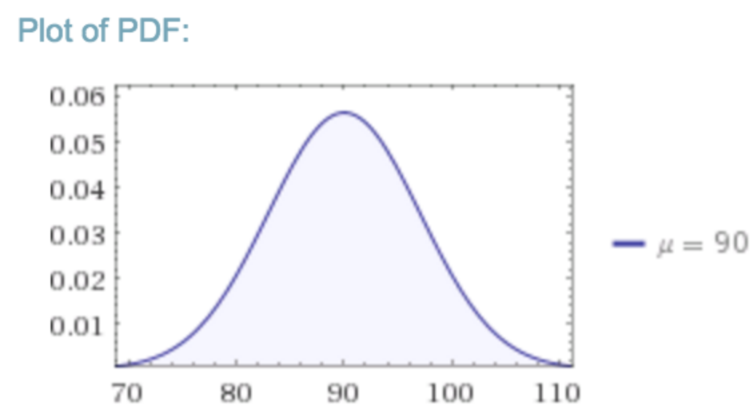
```
function get_foo(foo_id)
    foo = memcached_get("foo:" . foo_id)
    return foo if defined foo

    foo = fetch_foo_from_database(foo_id)
    memcached_set("foo:" . foo_id, foo)
    return foo
end
```

Memcached data flow



Tail Tolerance: Partition/Aggregate

- Consider distributed memcached cluster
  - Single client issues request to S memcached servers
    - Waits until all S are returned
  - Service time of a memcached server is normal w/ μ = 90us, σ = 7us
    - Roughly based on measurements from my former student



- A service has a response time drawn from a Gaussian (Normal) random variable N(μ, σ)
  - $\mu = 90$, $\sigma^2 = 50$us, so $\sigma \approx 7$
  - In python: numpy.random.normal(90,7,num)

- Scenario 1:
  - *c* clients each issue one independent request to the service

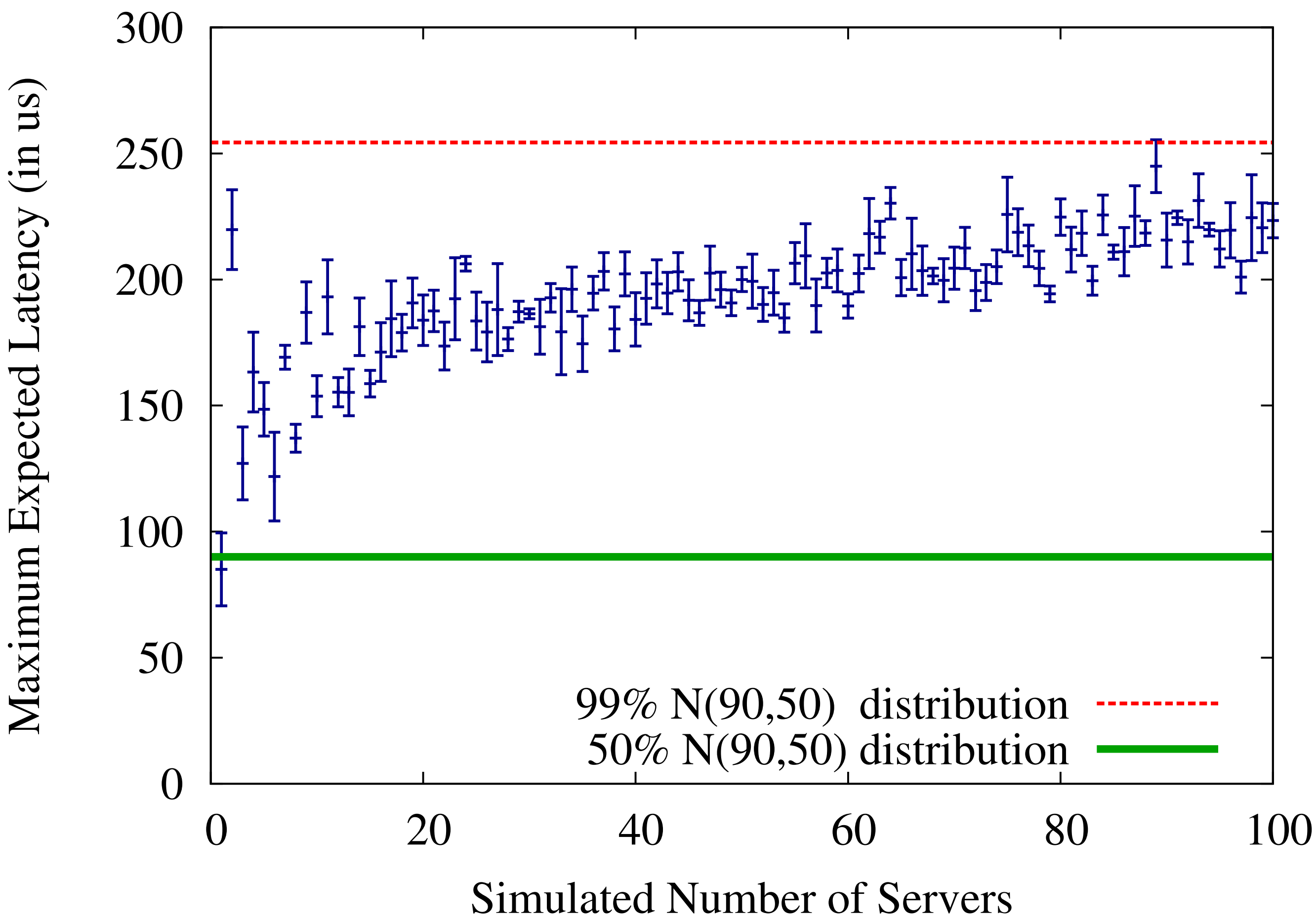
  c = {1,10,100,1000}
- Calculate the average service response time across all *c* clients
  - Hint: define avg(list)

- Scenario 2:
  - One client issues *p* requests to the service, and waits till all finish

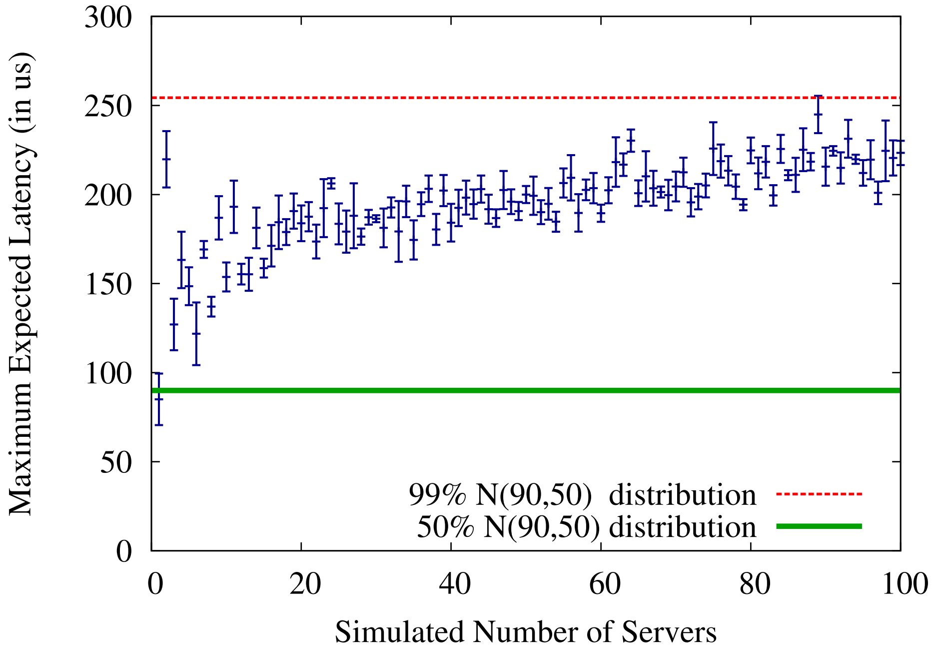
  p = {1,10,100,1000}
- Calculate the service response time seen by the client
  - Hint: define max(list)

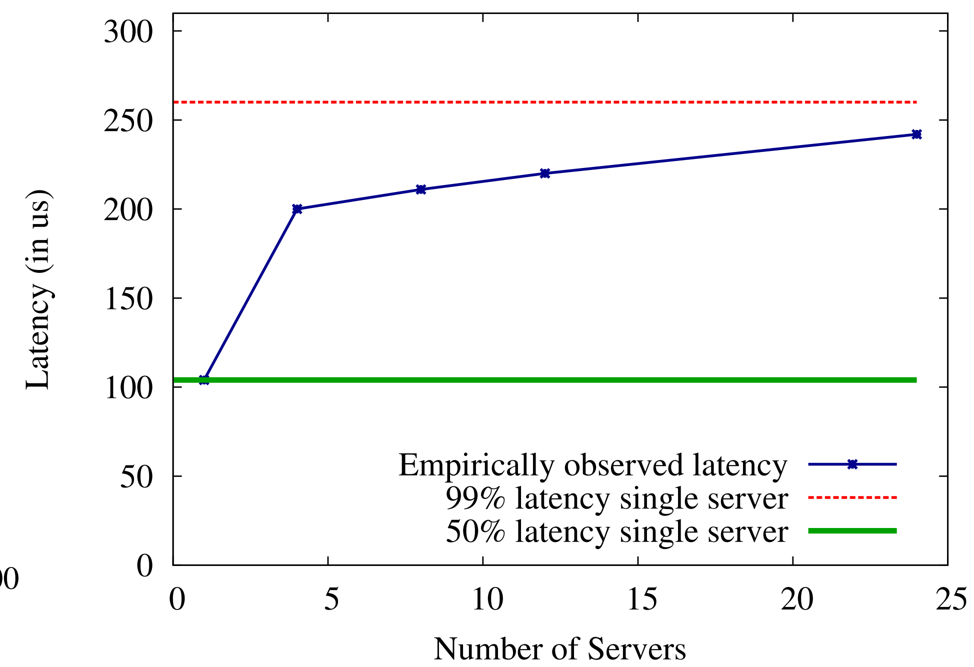
Part 3:
Memcache case study

Matlab simulation



Comparing Matlab to the real world





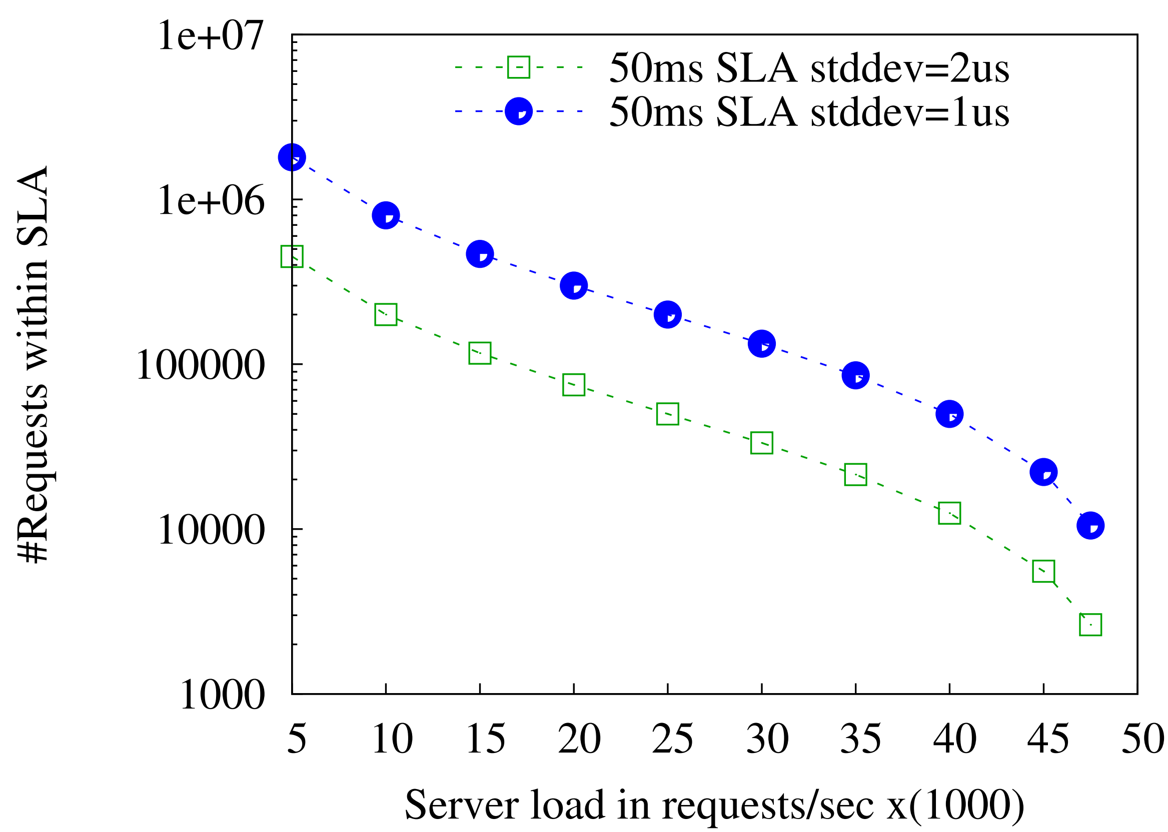
Tail tolerance: Dependent/Sequential pattern

- Consider iterative lookups in a service to build a web page
  - E.g., Facebook
- Issue request, get response, based on response, issue new request, etc…
- How many iterations can we issue within a deadline D?
  - For reference, Facebook limits # of RPCs to ~100

Dependent/Sequential pattern

- Service time of a web service is a function of the load (as is the variance)
- We carried out a queuing theory analysis to calculate the waiting time of the service as a function of the service time and variance

Dependent/Sequential pattern Simulation



Dependent/Sequential pattern Matlab vs. real world