Project 3: Scalable Web Service

15-440/640 Spring 2023

March 14

Goals:

- Build a scalable web store on top of cloud infrastructure
- Service emulated clients
  - Each client makes a randomized sequence of requests: several “browse” requests, possibly followed by a “purchase” request
  - If a client completes the sequence, they are Happy Clients (even if they do not purchase)
  - Clients expect prompt replies – if a reply takes too long or is dropped, they go away unhappy
- Metrics to minimize:
  - Number of unhappy clients (due to timeouts, dropped requests, etc.)
  - Total VM time rented from cloud service

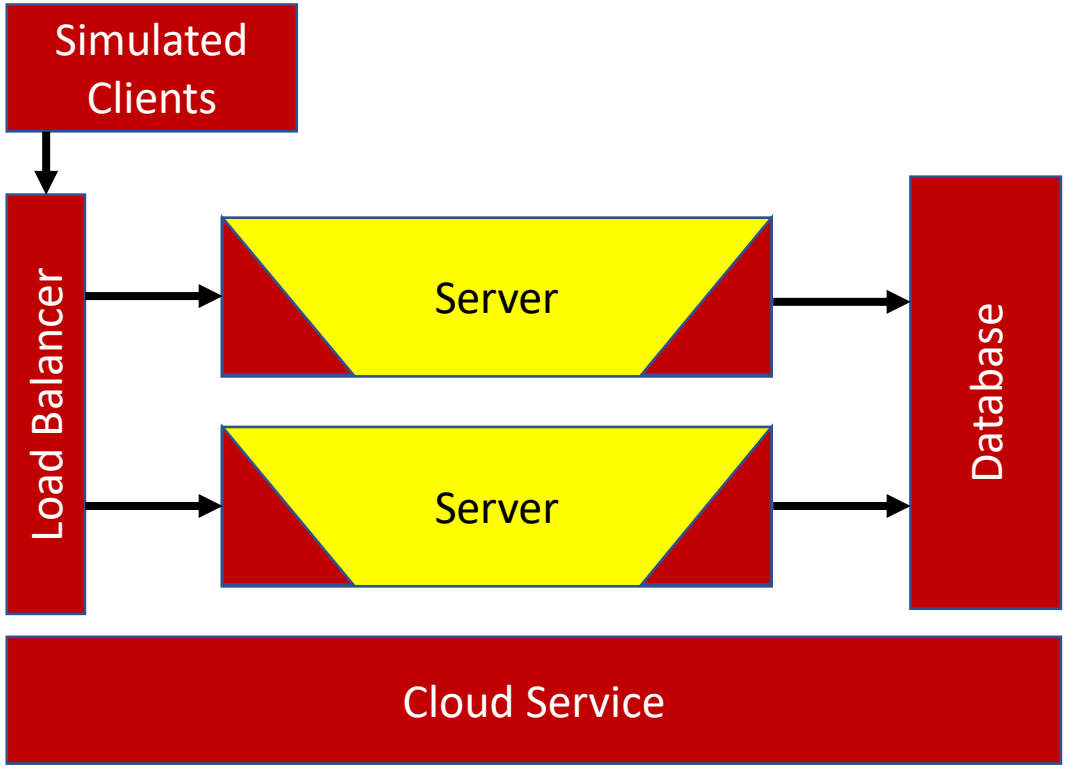
- Key focus areas:
  - Determine when to scale-out service to provide low service times
  - Deal with the inherent nondeterminism of this setting

Checkpoint 1: 2-tier system

- Architecture: Each server handles both web request and application logic
- DB tier keeps state of web store
- We provide:
  - Simulated Cloud service
  - Simulated Clients
  - Load balancer and DB server
- Basic server pseudocode:

```
register_with_load_balancer()
do forever:
    h = acceptConnection()
    r = parseRequest(h)
    processRequest(r)
```

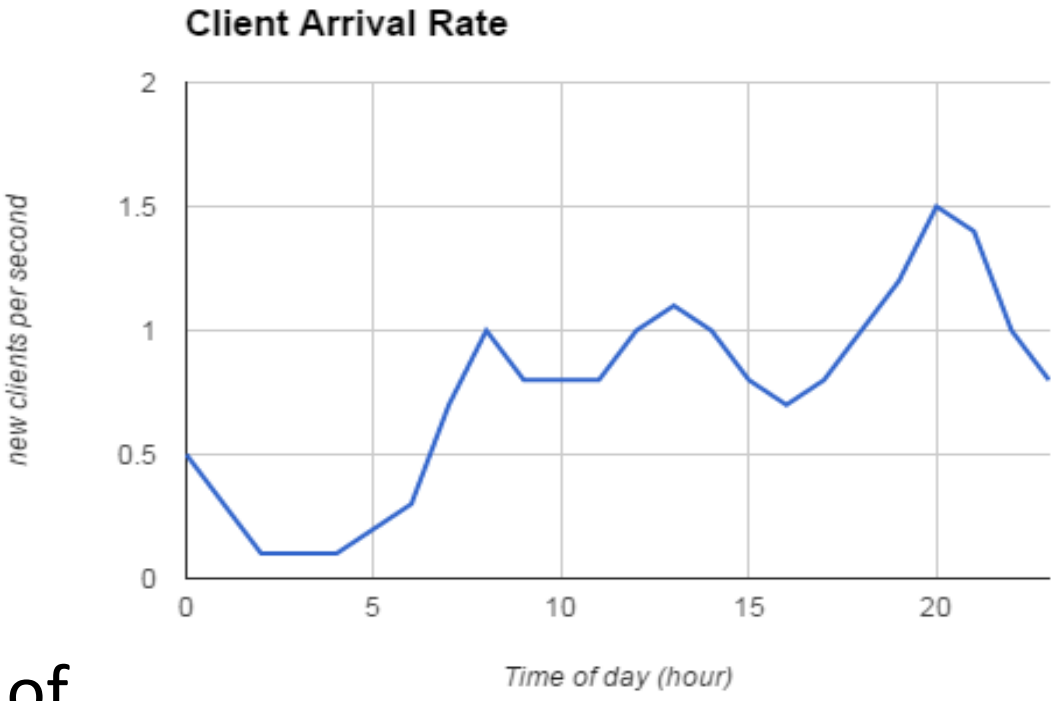
- You need to add logic to determine how to scale out the servers (launch more “VMs”)



- You will need to coordinate your servers, and handle any communications between them (more important in Ch 2 and Ch 3)

Checkpoint 1: Load based on time of day

- Uses known load based on time of day
- Diurnal curve for load provided in writeup
- Assume load will follow this curve i.e., can set number of servers based on the hour of the day
- Note that this load specifies the number of new clients arriving per second, not requests

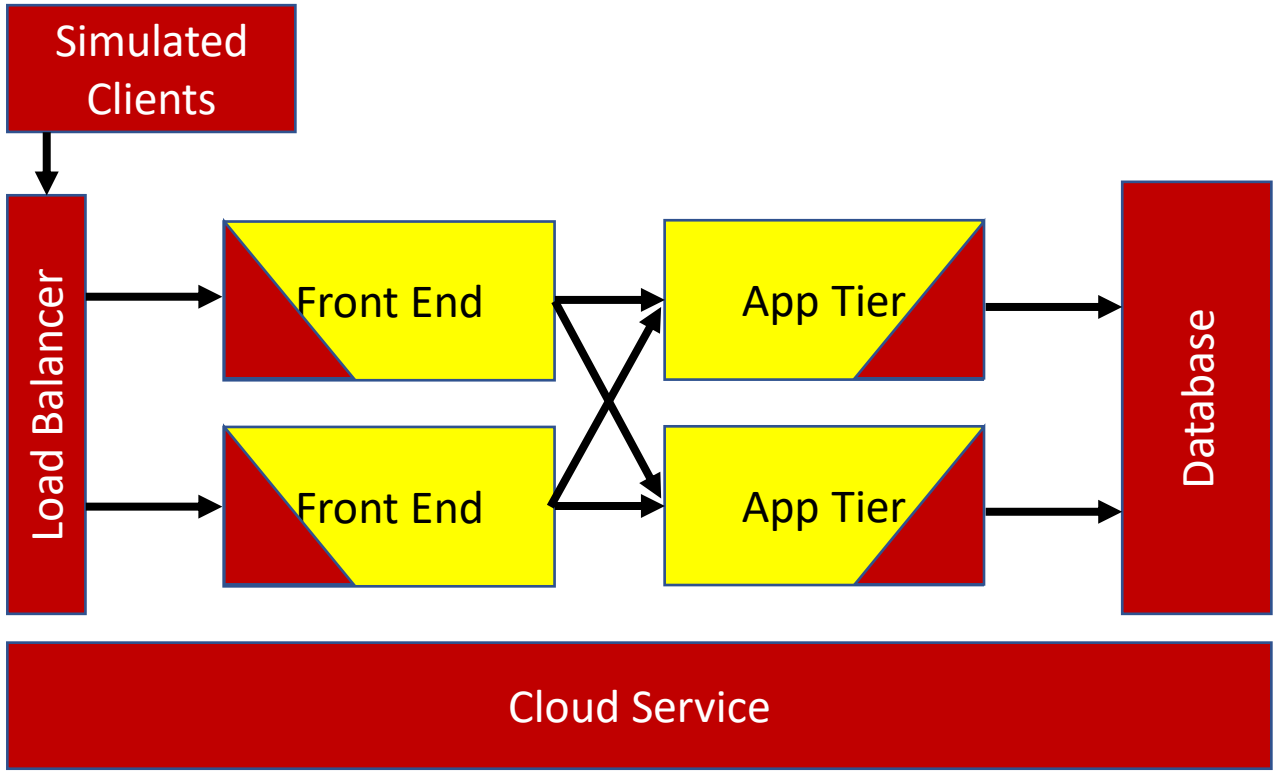


How many servers do I need?

- You need to minimize the number of “unhappy” clients, while simultaneously minimizing the number of servers/VMs used
- Surprisingly complex behavior in a simple distributed systems
- Use benchmarking to better understand the system performance and to determine the number of servers to handle a particular client load
- Example procedure: For a particular load level (client arrival rate)
  - Run system with 1 server, find out how many unhappy clients result
  - Run again with 2 servers; should see fewer unhappy clients
  - Repeat – should get better up to a point, after which adding more servers won’t help
  - Plot the results – this should help you determine how many servers you need for the particular load
  - Repeat the whole process multiple times and for multiple load levels
- You will need to make and hand in one such plot with your Checkpoint 1 submission.

Checkpoint 2 & 3: A 3-tier system

- Separate web front-end and application tier
- Only front ends register with load balancer, call `parseRequest`, `acceptConnection`
- Only App servers call `processRequest`
- Each of these tiers is scaled out independently



- Each new “VM” launches your server program – it will need to coordinate and determine its role / tier, communicate requests between tiers

Checkpoint 2: Arbitrary loads

- Loads do not follow diurnal curve
- Need to determine dynamically if you have enough servers
- What measures can help you detect if your servers are overloaded?
- Be careful – many measures are very noisy!
- Also, tradeoff between accurate measures and speed of adaptation
- Can still use benchmarking with static scaling to understand the ideal scaling for each load. Do your adaptive techniques get to the right number of servers in each tier?

Checkpoint 3: Dynamically Changing Loads

- Client arrival rates can change suddenly
- Will include tests where load increases suddenly, and drops suddenly
- Your system needs to adapt quickly to the load changes
- Same issues as in CH 2, but now the speed of detection and adaptation are more critical

General Comments

- Very little actual code needs to be written – most of the work is in benchmarking, determining good heuristics for adaptation, and testing on your own
- You have the whole infrastructure, including client simulation – you can run any load we can. Specifics of the autolab tests, and grading cutoffs will not be provided.
- Almost everything your Server needs to do is provided by methods of ServerLib class. Please read the documentation (pdf handout and the html in the doc folder of 15440-p3.tgz).
- Each "VM" is an instantiation of your server process. Do not launch multiple processes yourself. You can use additional threads in your code, but this will not give you more cpu resources (i.e., acts like a single-core VM)
- Autolab grading is VERY SLOW. Only 10 submissions are allowed penalty-free per checkpoint. Do not wait until the last day – it may take **hours** to get your score back!
- Runs are non-deterministic – there will be some randomness in your results. Depending on how your code works, it may dampen these effects, or exaggerate them. You will not be able to pick and choose the best of multiple autolab runs – only the last one counts.
- Bonus points – the numerical score on each test is based on how low you can get VM time and unhappy clients (not just all or nothing). Do well on a test and you can get more than the nominal points for that test.