CS-310 Scalable Software Architectures

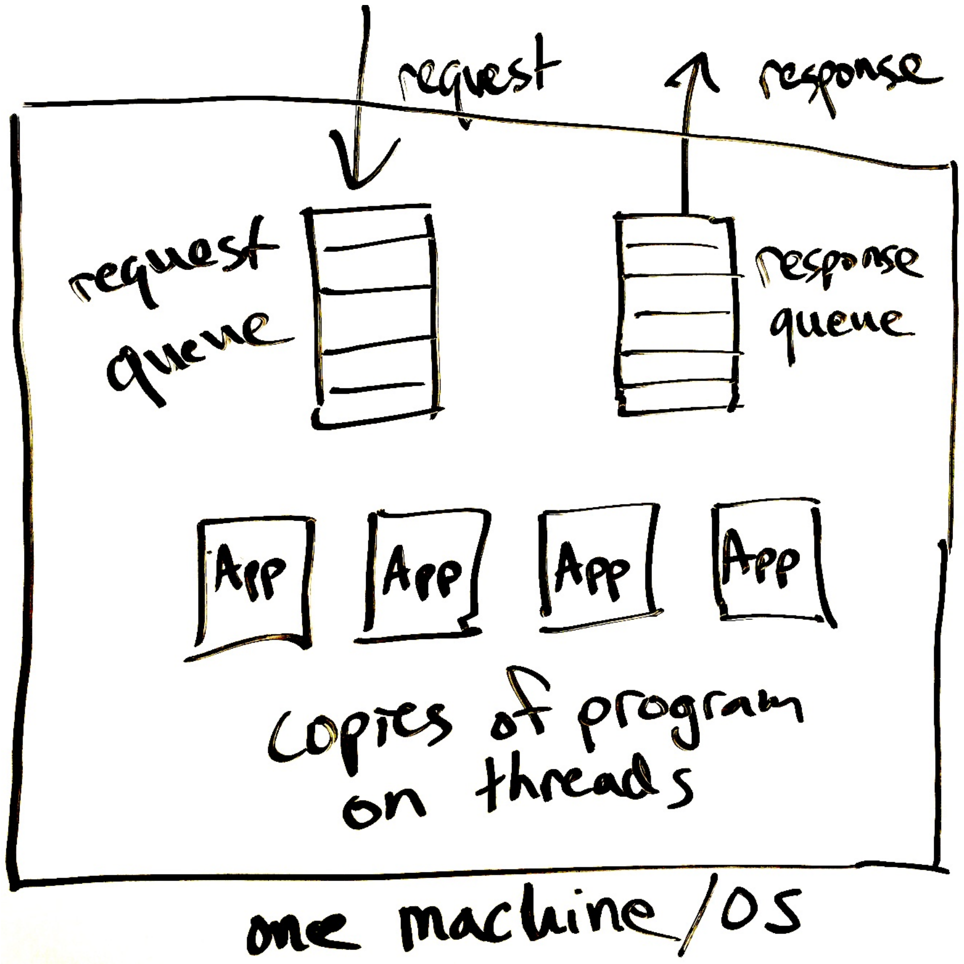
Lecture 3: Stateless Services, Proxies, and Caches

Steve Tarzia

Last time:

- Showed that web server frameworks let you translate a simple program into a multi-threaded service with concurrency.
- Introduced HTTP as the most common type of service.
    - Client **requests** a document (specified in path/url)
    - Server sends document in the **response**.
- High-level overview of Wikipedia's architecture.
- Showed examples of traditional dynamic web code, where HTML is programmatically generated.

Are all workers equal?



- Each request can be handled by one of several possible "worker" threads.
- Does it matter which is chosen?

STOP and THINK

- It depends on how the app code is written!

Stateless and Stateful worker threads

state == memory

- A **stateless** thread/process/service remembers nothing from past requests.
  - Behavior is determined entirely by two things:
    ⟨input request, request handling code⟩.
  - Different copies of the service are running the same code, so they will give the exact same response for a given request.
  - Has no <u>local</u> state.

We'll see later that state is pushed up to client or down to a database.

- A **stateful** thread (or service) changes over time, as a side effect of handling requests.
  - Persistent, global variables are modified by the request processing code.

Stateless code has no long-term "memory"

- It's almost a "pure function" in programming language terminology.
- Output is not affected by previous inputs.
- We do **not** say "output is determined entirely by the current input," because we allow nondeterministic (random) behavior.
- Eg:
  - `float` **`cosine`**`(float x)`
  - `int` **`sum`**`(int a, int b)`
  - `List` **`sort`**`(List myList)`
  - `List<T>` **`listAppend`**`(List<T> myList, T newItem)`
  - `float` **`generateRandomNumber`**`()`

Real random number generators actually do keep some state, but ideally they would not.

Stateful code has side effects (long-term memory)

- It's like an object or a code that changes global variables.
- Object-oriented mutator:

```
Class Counter {
  private int count;
  public Counter() {
    count = 0;
  }
  public void increment() {
    count++;
  }
}
```

- Imperative code changing globals:

```
int count;

void increment() {
    count++;
}
```

Side note on OOP

- What are the main purposes of object-oriented programming?
- *You probably learned:*
  - **Inheritance:**
    - This allows strong typing without losing abstraction.
    - Creates generic, abstract interfaces, enabling abstraction.
  - **Modeling** real-world concepts.
    - Animal → Mammal → Cow!)
- *But another major OOP benefit is*:
  - Grouping sets of related state (memory/variables).
  - Well-defined, **limited side effects**.
  - A class defines a set of member functions whose side-effects are limited to a small set of variables (the object's data members).

STOP and THINK

Horizontally scaling Stateful vs Stateless code

- Let's say we want to run many copies of our service code in parallel to handle lots of requests (horizontal scaling).
- **Stateless** code (has "no memory"):
  - All copies will give same response,
    it does not matter which copy processes a given request.
    - **Parallelism is trivially easy!**
- **Stateful** code (does have "memory"):
  - Since different copies handled different past requests, their state differs, and they may give a different response to the exact same request.
    - **Related requests (from the same client) must go to same handler.**

Stateful example: SMTP

C: means client request
S: means server response

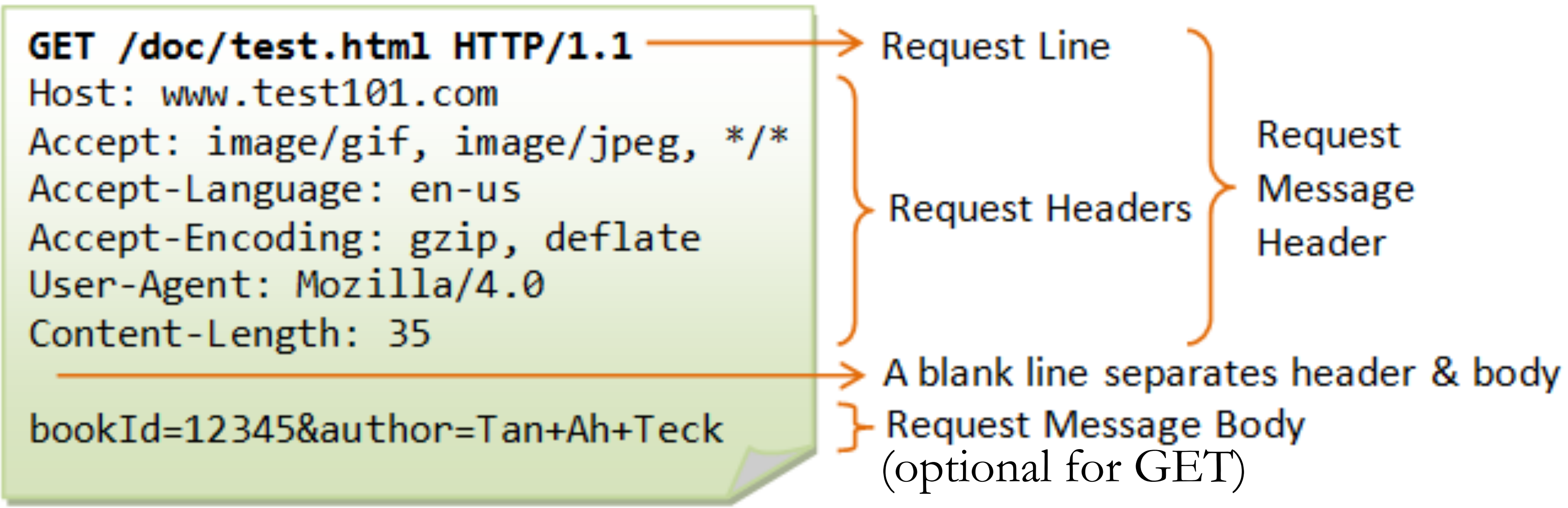
- The text at right shows a series of 6 requests and responses sent from an email client to an email server.
- The result is a single new email.
- Server must remember information from the previous requests to finally build the email message.
- If we were running the email server code in parallel on many machines, then all these requests must be handled by the same server to complete the task.

```
C: HELO relay.example.com
S: 250 smtp.example.com, I am glad to meet you
C: MAIL FROM:<bob@example.com>
S: 250 Ok
C: RCPT TO:<alice@example.com>
S: 250 Ok
C: RCPT TO:<theboss@example.com>
S: 250 Ok
C: DATA
S: 354 End data with <CR><LF>.<CR><LF>
C: From: "Bob Example" bob@example.com
C: To: Alice Example alice@example.com
C: Cc: theboss@example.com
C: Date: Tue, 15 January 2008 16:02:43 -0500
C: Subject: Test message
C:
C: Hello Alice.
C: This is a test message with 5 header fields
C: and five lines in the message body.
C: Your friend,
C: Bob
C: .
S: 250 Ok: queued as 12345
```

Stateless example: HTTP

Request:

One big request that is self-sufficient (independent)

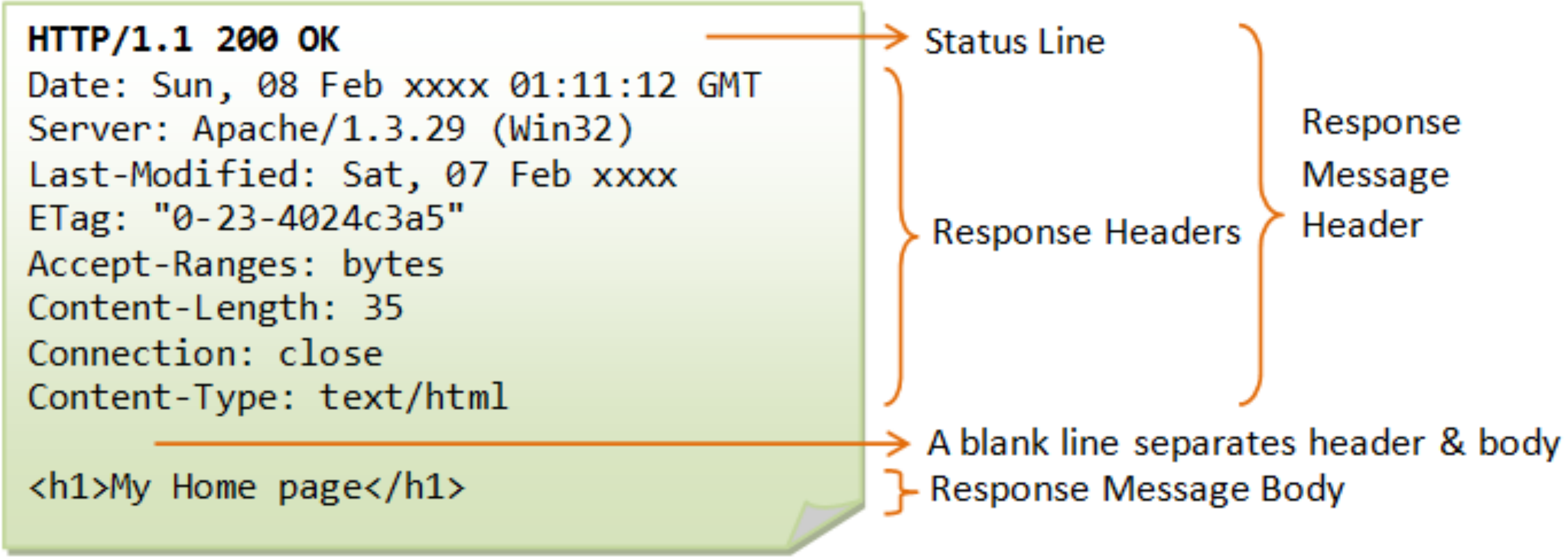


```
GET /doc/test.html HTTP/1.1
Host: www.test101.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Content-Length: 35

bookId=12345&author=Tan+Ah+Teck
```

(optional for GET)

Response:



```
HTTP/1.1 200 OK
Date: Sun, 08 Feb xxxx 01:11:12 GMT
Server: Apache/1.3.29 (Win32)
Last-Modified: Sat, 07 Feb xxxx
ETag: "0-23-4024c3a5"
Accept-Ranges: bytes
Content-Length: 35
Connection: close
Content-Type: text/html

<h1>My Home page</h1>
```

From https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html

Should MediaWiki (Wikipedia) be stateful or stateless?

Tasks:

- Get corresponding wiki text from DB.
- Translate wiki text to HTML.
- Add wrapping content and banners.
- Add user-specific page header, based on cookies in request.

Recall:

- *Stateless* applications do **not remember** anything from previous requests.
  - Each request can be handled independently based exclusively on the input request.
  - Can be trivially parallelized because handling a request has no side effects in the handler.
- Which of these tasks have side effects?
- Are there other MediaWiki tasks that have side effect?

STOP and THINK

Page Edit might have side effects in MediaWiki

- Most visitors just read Wikipedia pages, but some also **edit** pages.
- Edits are sent as HTTP POST requests to the same MediaWiki app.
- Clearly, these edits should affect the results of future page fetches.
    - If I edit a page on server A, then a user requesting the same page from server A or server B should see my edits.
    - The edits should have system-wide side effects.
    - Can MediaWiki still be stateless?

STOP and THINK

Push state **down** to a database.

Yes! **Databases** separate system state from stateless request handlers.

- The edit's results are stored by MediaWiki in an external, shared DB.
- The DB must be queried for every page fetch.
    - Thus the PHP code in MediaWiki can remain stateless and easily parallelized.

Sign In might have side effects in MediaWiki

- After signing in, all later response HTML will have a different page header, including your username, notifications, etc.
- Handling a "sign in" request has a side effect on later page fetches.
- How can we avoid keeping this "sign in" state in MediaWiki?

STOP and THINK

Push state **up** to the client

Cookies solve this problem

- Sign-in leads to a cookie being stored in the DB and returned to the client browser. So, client and DB keep the sign-in state.
- Client sends the cookie as an HTTP header in all future requests.
- Cookie is provided as an input to MediaWiki, and MediaWiki checks the cookie against cookies stored in the shared database.
  - Even better, *signed* cookies can be verified without a backend database.

Response to sign-in request gives user a **cookie**

- Cookies are how web applications track **state,** often to track user identity.
- If username and password were correct, server will return a cookie in the response:

```
HTTP/1.1 302 Found
Location: http://somewebsite.com/account
Set-Cookie: someweb-id=kfj203d14t9s
```

- Response tells the browser to redirect to `http://somewebsite.com/account`, but it also gives the browser a cookie to remember.
- Browser will include the cookie in all future HTTP requests to `somewebsite.com`:

Is HTTP with cookies still stateless?

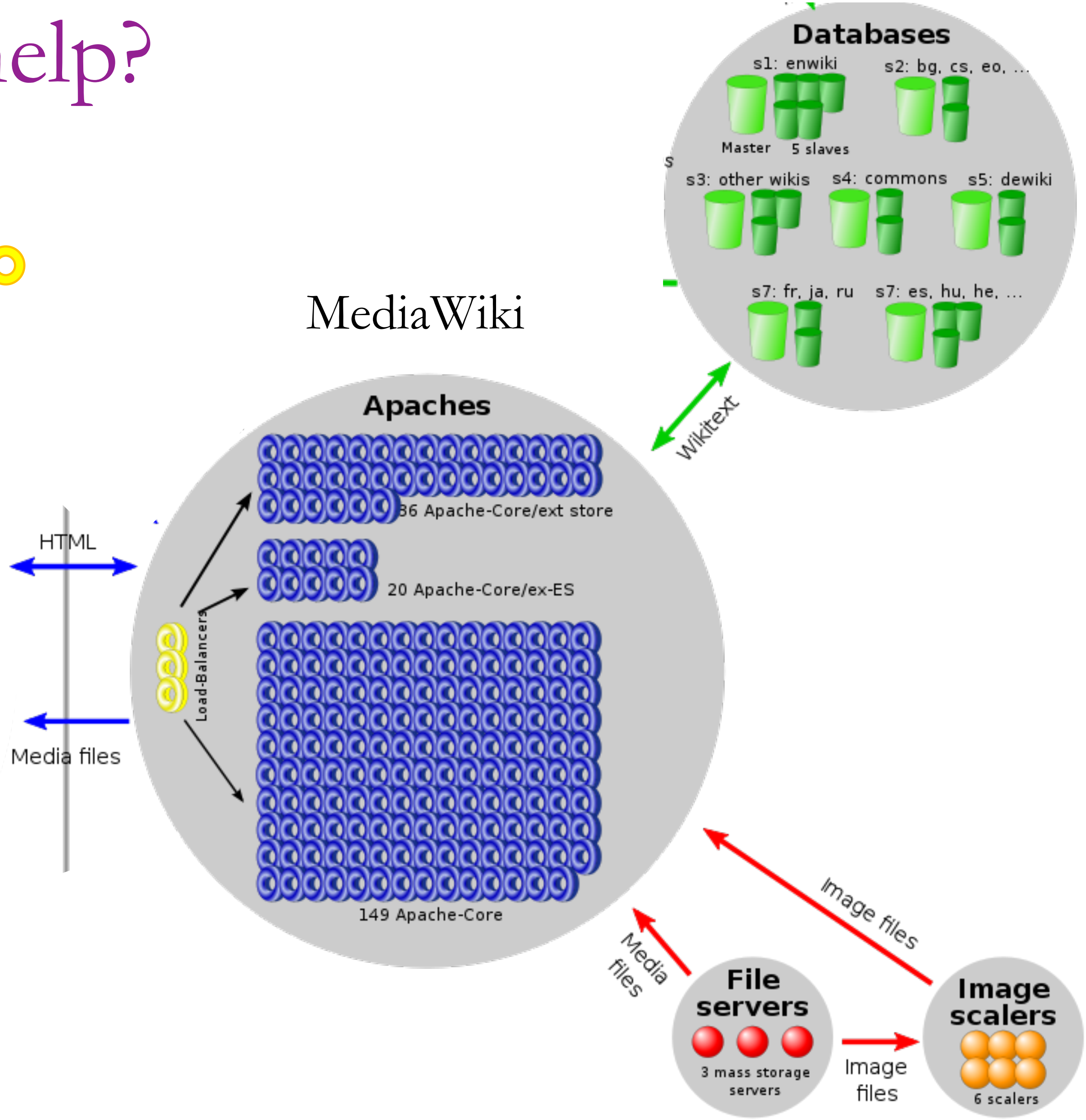
STOP and THINK

```
GET /account HTTP/1.1
Host: somewebsite.com
Referer: http://somewebsite.com/bin/login
Cookie: someweb-id=kfj203d14t9s
…
```

- Server getting this request can use the cookie to determine which user it came from!

How does statelessness help?

- 200 instances of MediaWiki can be run behind a **load balancer.**
  - Load balancing is done both by DNS and by efficient, simple software proxies.
- Any of the 200 instances can handle any request.
  - Each of those 200 machines also has many CPU cores and dozens of software threads.
- Coordination only happens by writing to shared databases.

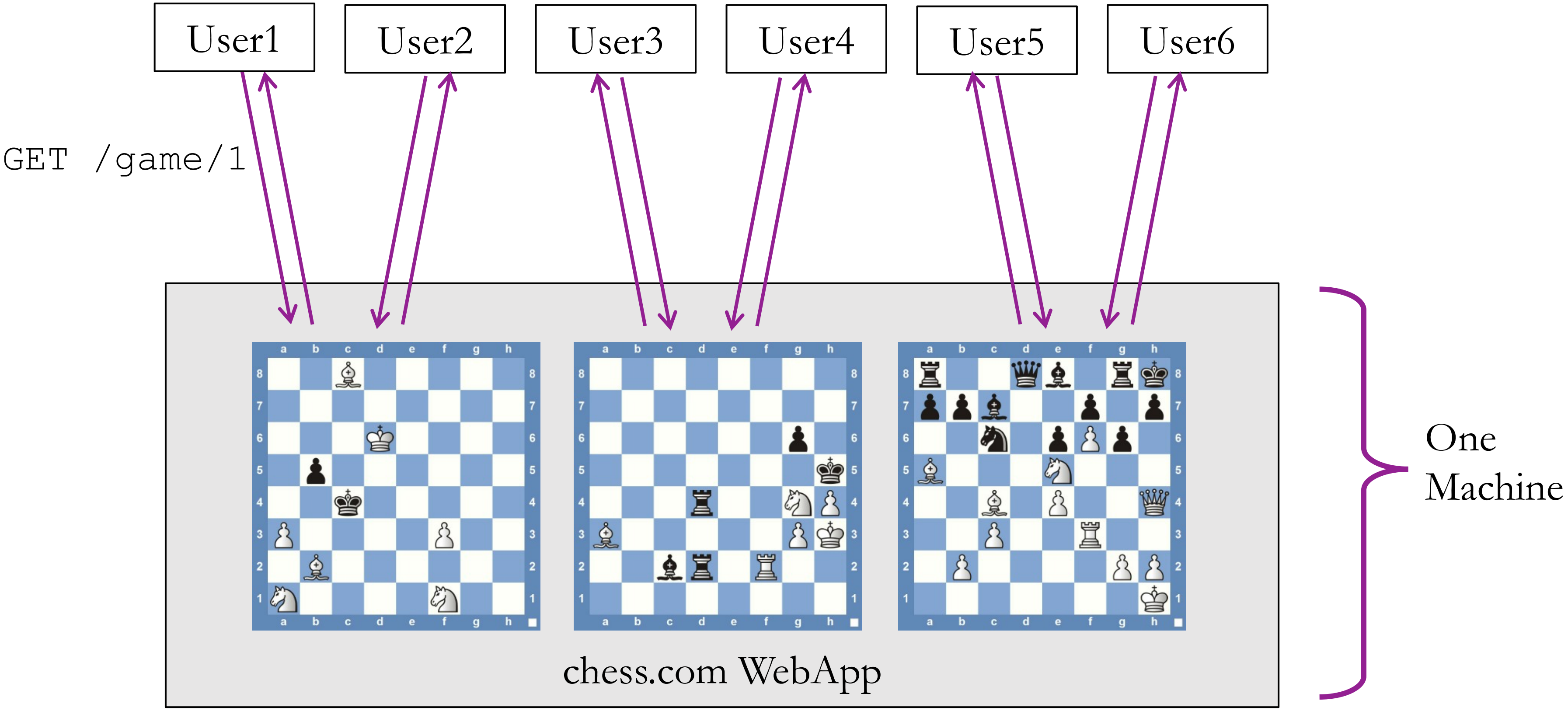


Design Example: A chess website

- We need to track the state of many games being played at once.
  - We want to render pages like this: https://chess.com/game/23
- Simplest design is to store game state in memory (eg., in a dictionary)

- How can we scale this app?
  - Vertically?
  - Horizontally?

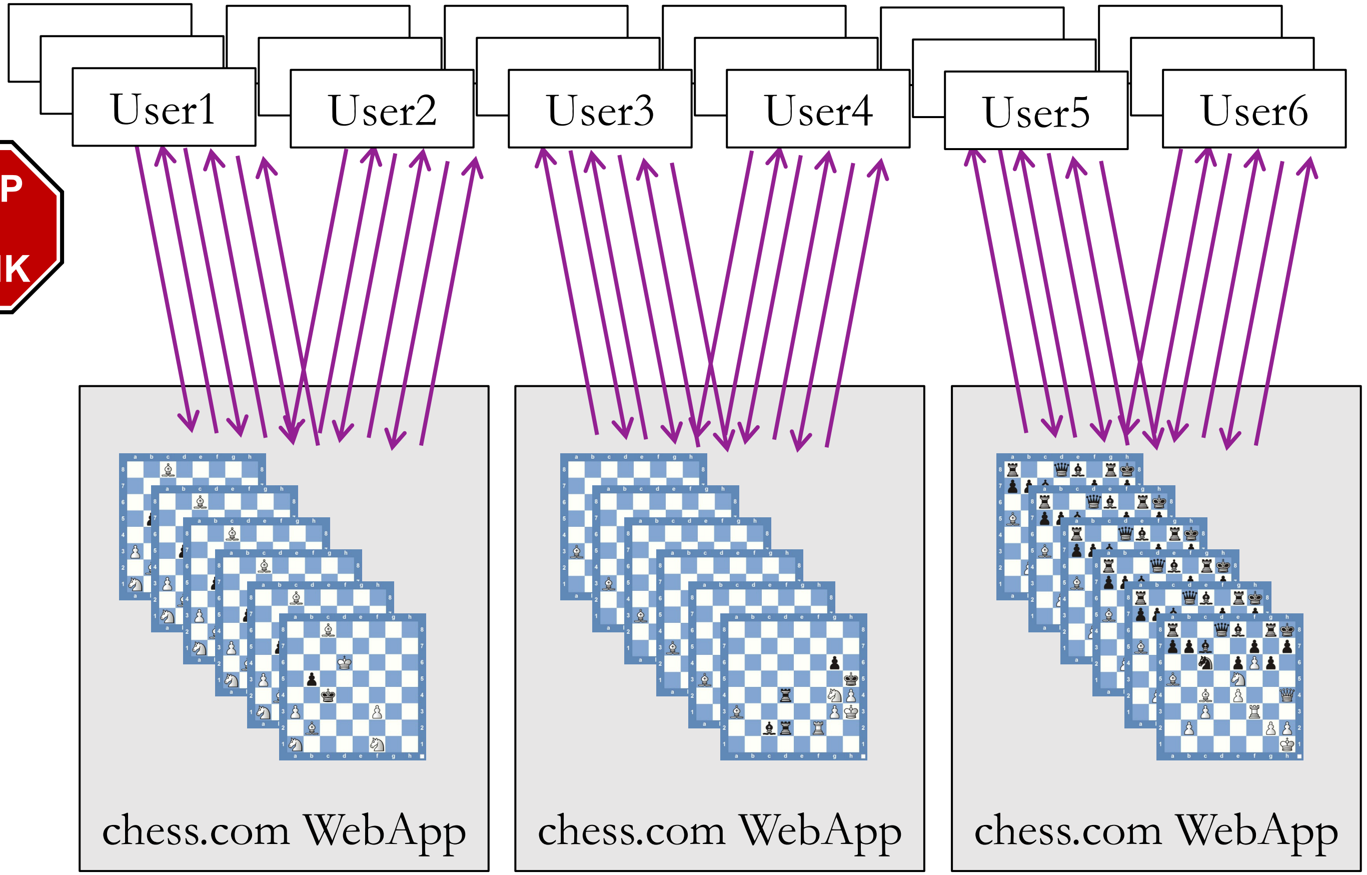
STOP and THINK



Horizontal scaling of chess app

- Our first attempt will run the same simple code on multiple servers.
- Each game runs on one of many servers. Each server handles a fraction of games.
- Can you see any problems with this scaling approach?
  - User must connect to exact same server to continue their game. How to direct user?
  - If a server fails, 1/n games are lost (or at least interrupted).
- These are **stateful** web apps.

STOP
and
THINK

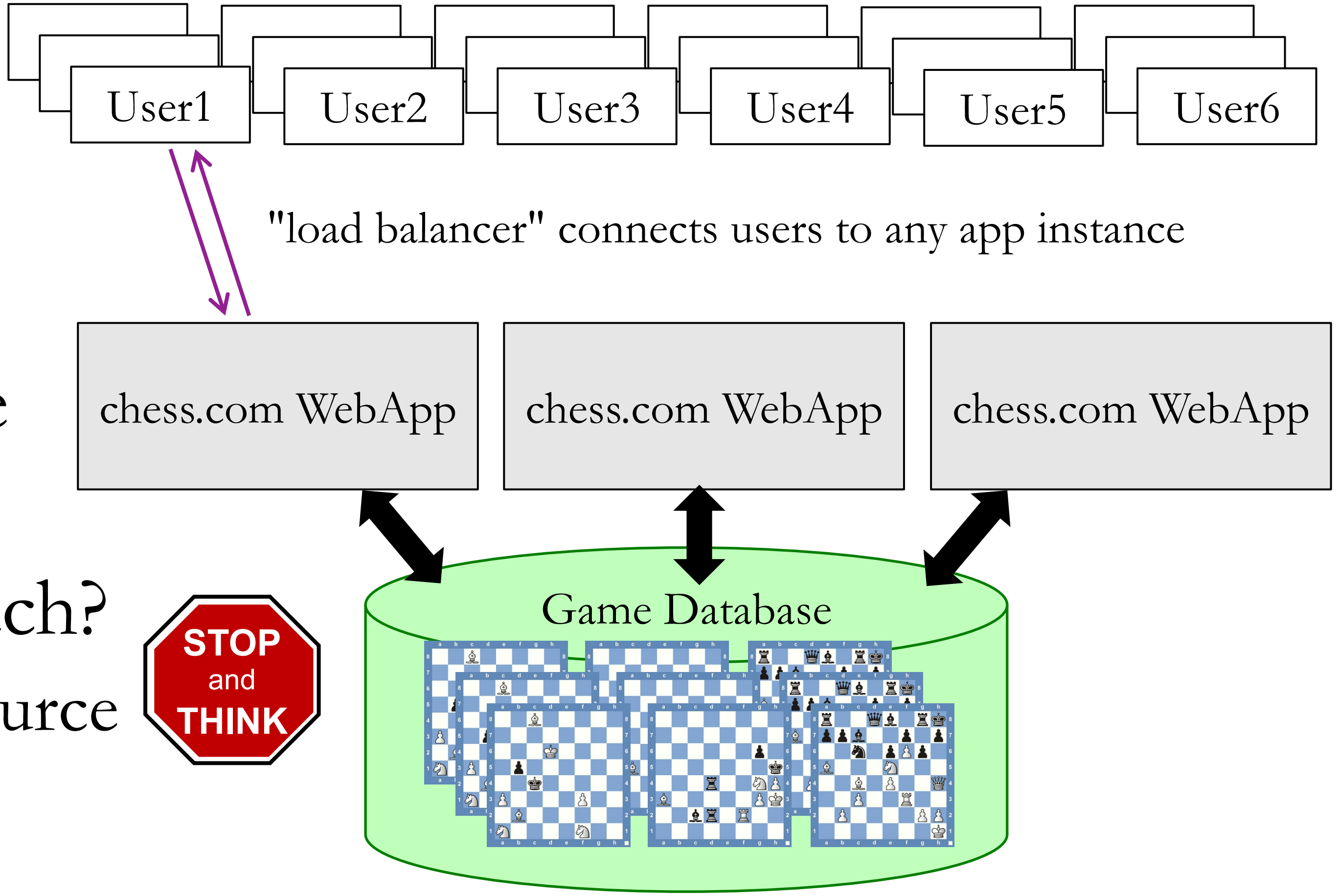


Stateless design of horizontally-scaled chess app

STOP and THINK

- Push all the game state to a central, shared database.
- This is equivalent to MediaWiki pushing all article data to a DB.
- User can connect to any one of the chess webapp instances to play any game.
- Some kind of **load balancer** directs user to a server instance (more on this in later lectures).
- Any problems with this approach?
    - The DB is a central, shared resource that will limit scalability.

STOP and THINK



Review

- Defined **stateless** and **stateful** services.
- Showed how databases and cookies make MediaWiki stateless and scalable.
- In other words, we achieved parallelism and distributed execution while avoiding difficult coordination problems. Just push away all shared state. Push state **up** to client and/or **down** to database.
- First lesson of scalability: **Don't share!**

Unsolved problems:

- How to direct users to an instance of a service (load balancing)?
- How to avoid a performance bottleneck in the database?