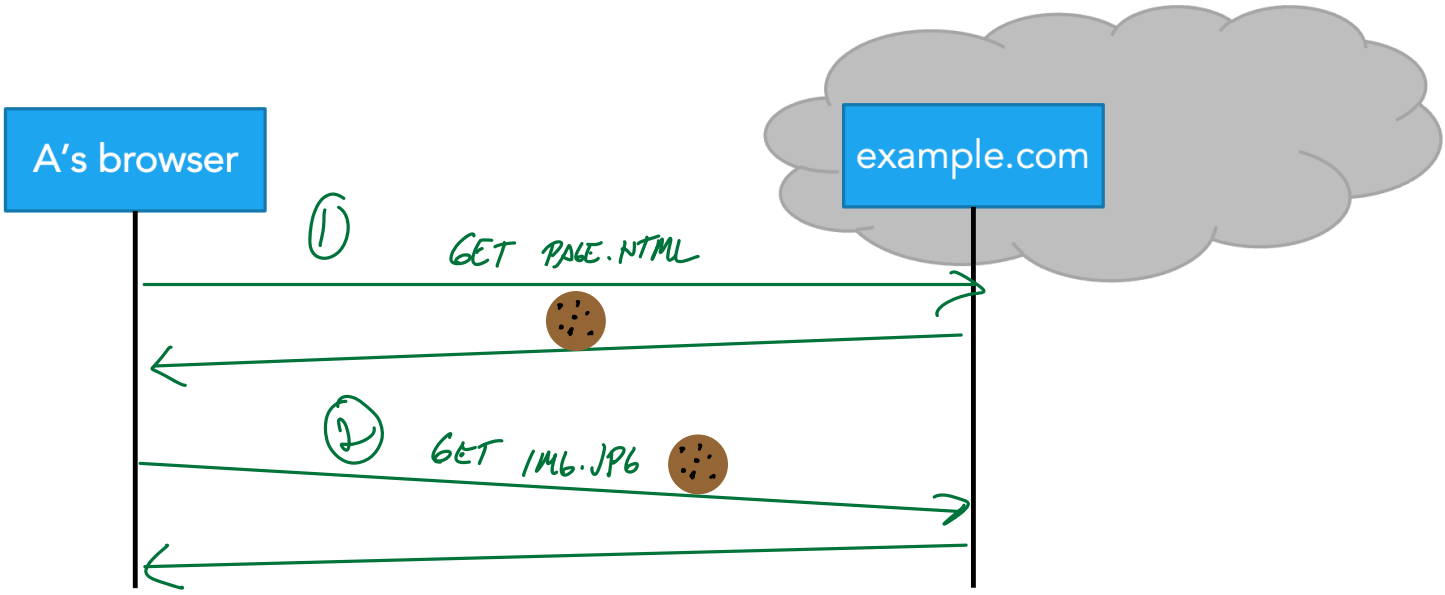
Today
- More on CDNs
- HTTP push

Administrivia
- TCP due a week from today
- Look for a reference update
- We will have one more HW, due after TCP

Lecture 21: More on CDNs, HTTP Push

Warmup: If client A makes two separate HTTP requests to example.com. How does the ~~serer~~ server know that both requests came from A? How many reasons can you list?

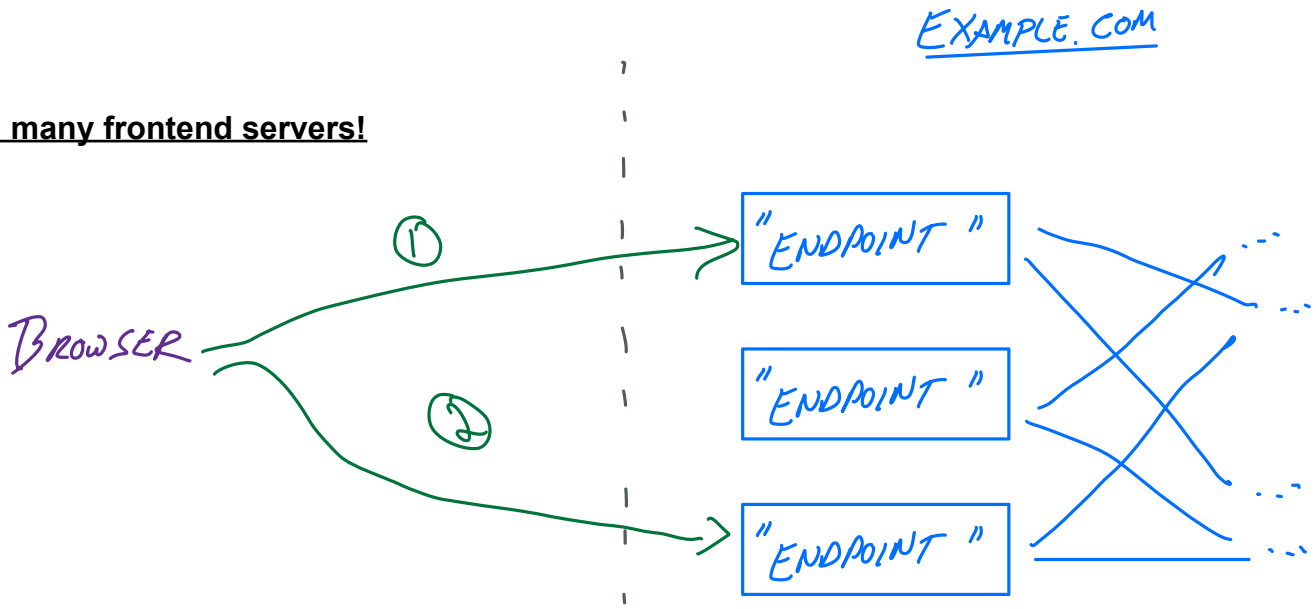


Ideas?
- Can re-use a connection
- Cookies sent with request
- Extra info stored-server-side
  => cookie becomes like a key to look up stuff
- Can also set some kind of token in header (Authentication token)

=> Stateless by default, requests can add other info to identify client

What about socket-level info, like the client's IP address? There are several reasons this would be problematic:
- *The same user might move to a different network, which will change their IP*
- *One IP might be re-used for different devices via DHCP*
- *One IP might be used for multiple devices, which we'll talk about next lecture*

Consider: many frontend servers!



For a larger web service, the "endpoints" that receive requests don't necessarily have state about the client, and are likely different servers!

=> Client needs to send some info to identify itself

("Endpoint" is a device at the "edge" of the app's network that receives requests -- it could be a webserver or some kind of proxy designed to send requests to other servers, depending on how the network is set up.)

Reverse proxy

=> Proxy server that lives somewhere in the network, transparent to the client

Can perform many functions for scaling and security

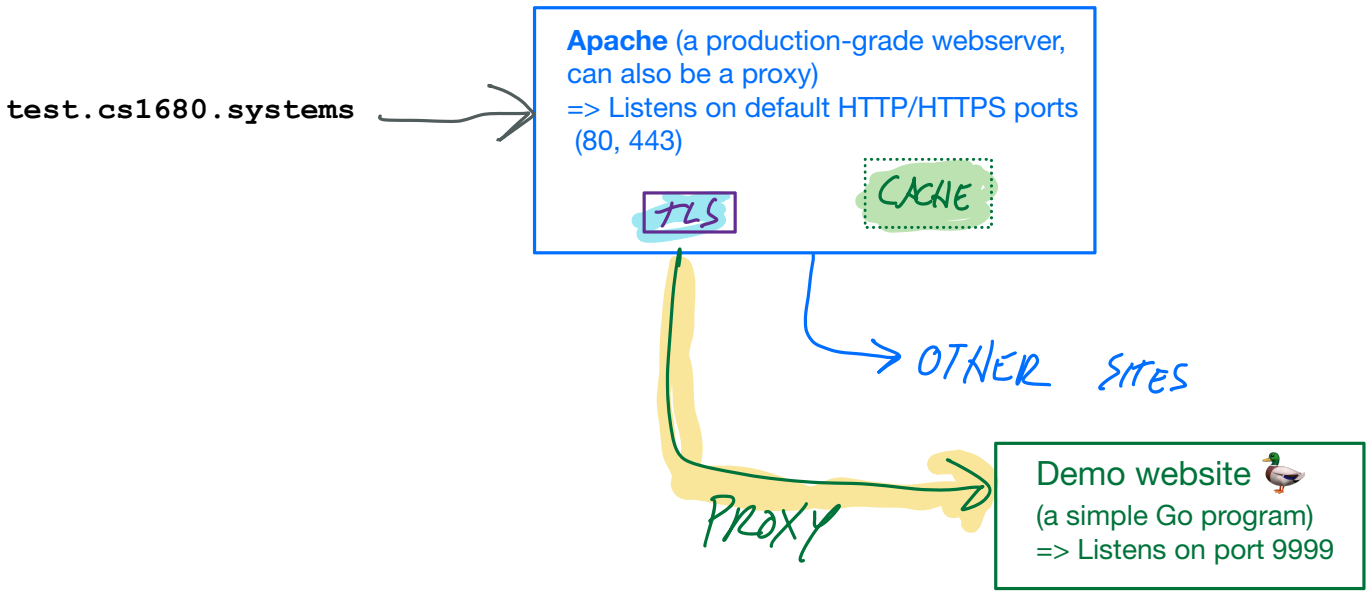
- Cache
- Load balancing
- ...

Small-scale proxy example: Nick's webserver

- Our little demo website just used a tiny Go webserver, just a few lines of code
- But to run on the Internet, should have more features (like security via TLS, caching, etc)

Idea: use a production-grade webserver as the Internet-facing webserver, set it up as a reverse proxy for the demo site

=> A common way apps are deployed!



In the example, Apache is configured as a reverse proxy, with a few features turned on
=> Uses TLS: clients can connect securely via apache, which sends requests to demo website
=> No need to modify the demo website to do this!

=> Apache can act as cache => can return data for cached requests without consulting demo website

A simple reverse proxy config

(You absolutely do not need to understand this configuration language)

```
<VirtualHost *:443>
      ServerName test.cs1680.systems
      ErrorLog "/var/log/httpd/test-error_log"
      CustomLog "/var/log/httpd/test-access_log" combined
      ProxyPass "/" "http://127.0.0.1:9999/"
      ProxyPassReverse "/" "http://127.0.0.1:9999/"

      CacheEnable disk /
      CacheRoot /var/cache/apache2/..

      SSLCertificateFile /etc/letsencrypt/live/test.cs1680.systems/fullchain.pem
      SSLCertificateKeyFile /etc/letsencrypt/live/test.cs1680.systems/privkey.pem
      Include /etc/letsencrypt/options-ssl-apache.conf
</VirtualHost>
```

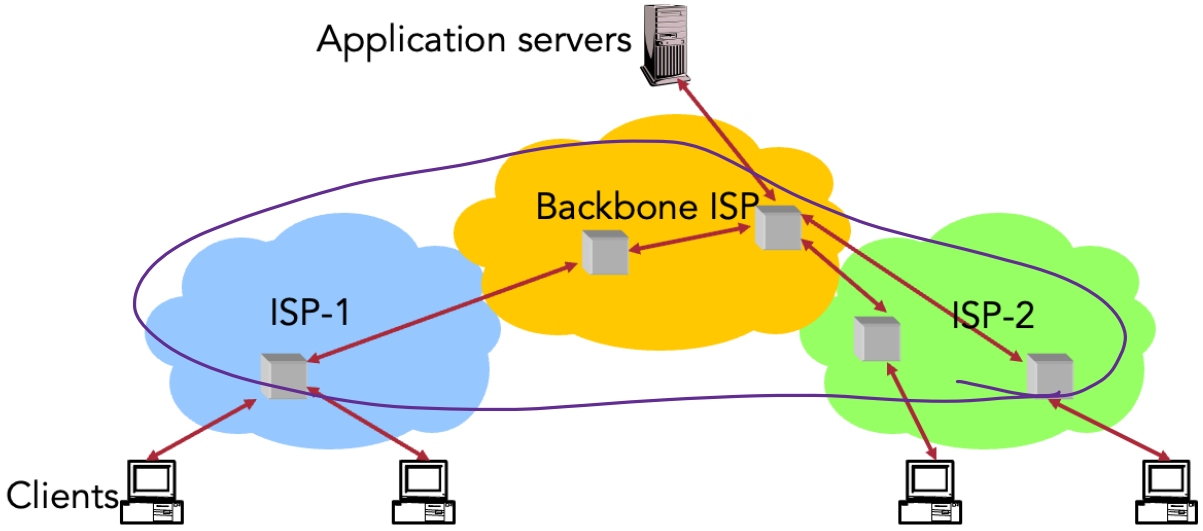
TURN ON PROXY

CACHE

TLS

CDNs

Companies that specialize in providing caching services (among other things)
=> Akamai, Cloudflare, ...



Can also be used to secure traffic...

DDoS Attack: overwhelm a target host/network with packets, denying resources
=> Performed by large groups of compromised devices
=> "botnet"

CDNs for securing traffic

DDoS attacks: overwhelm a target host/network with packets, denying resources for legitimate traffic

↳ CDNs CAN ALSO PROVIDE "DDoS MITIGATION"

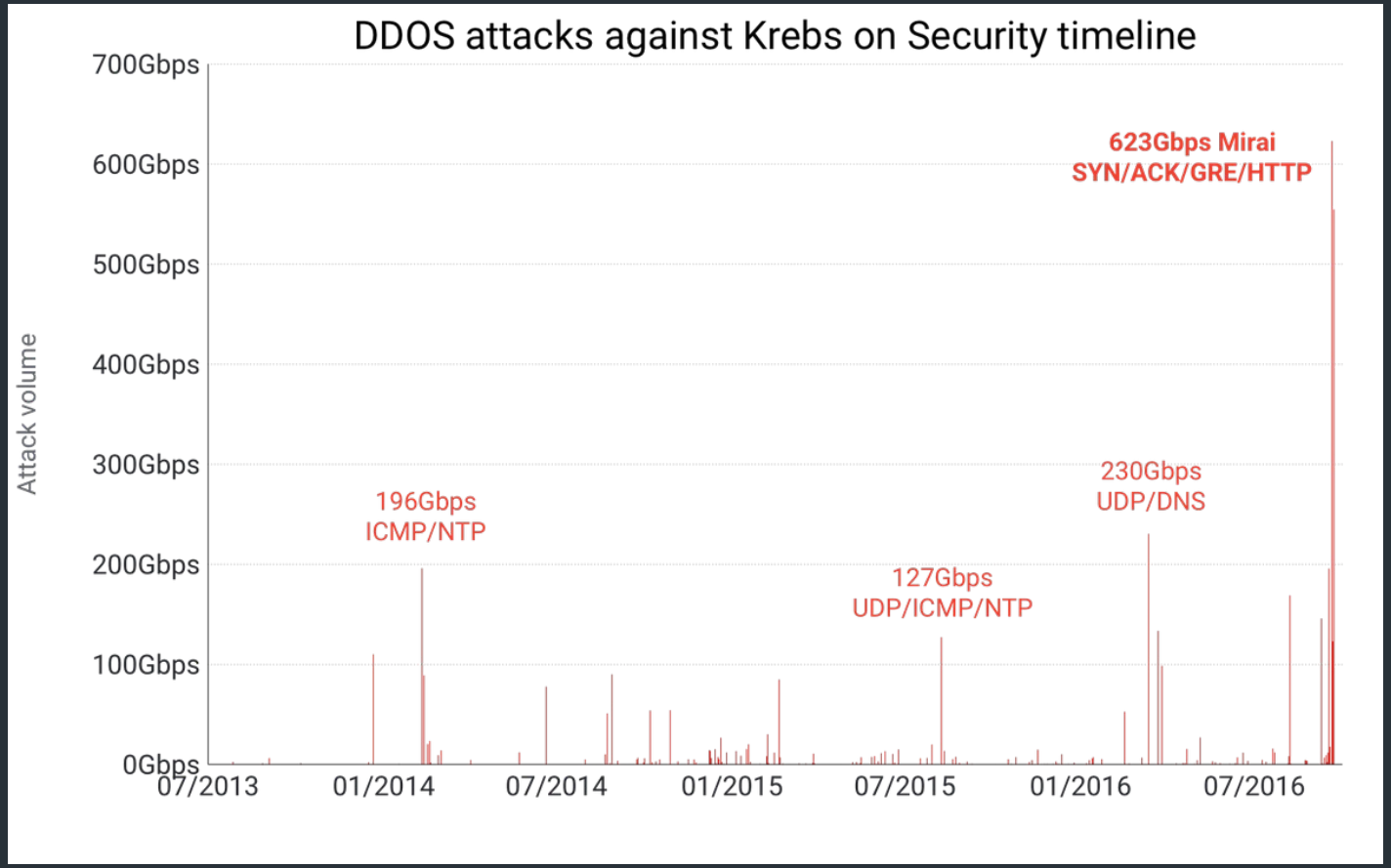
DDoS attacks: overwhelm a target host/network with packets, denying resources for legitimate traffic

=> Often performed by "botnets" of compromised devices

=> Attack traffic can take many forms: lots of SYNs, DNS requests, exploiting bugs in protocols, …

⇒Want to learn more? CS 1660.

Lots of compromised devices overwhelming network with bogus traffic
=> A CDN or other DDoS mitigation service can "absorb" traffic, can also do some detection



DDoS mitigation via CDN

CLOUDFLARE The Cloudflare Blog

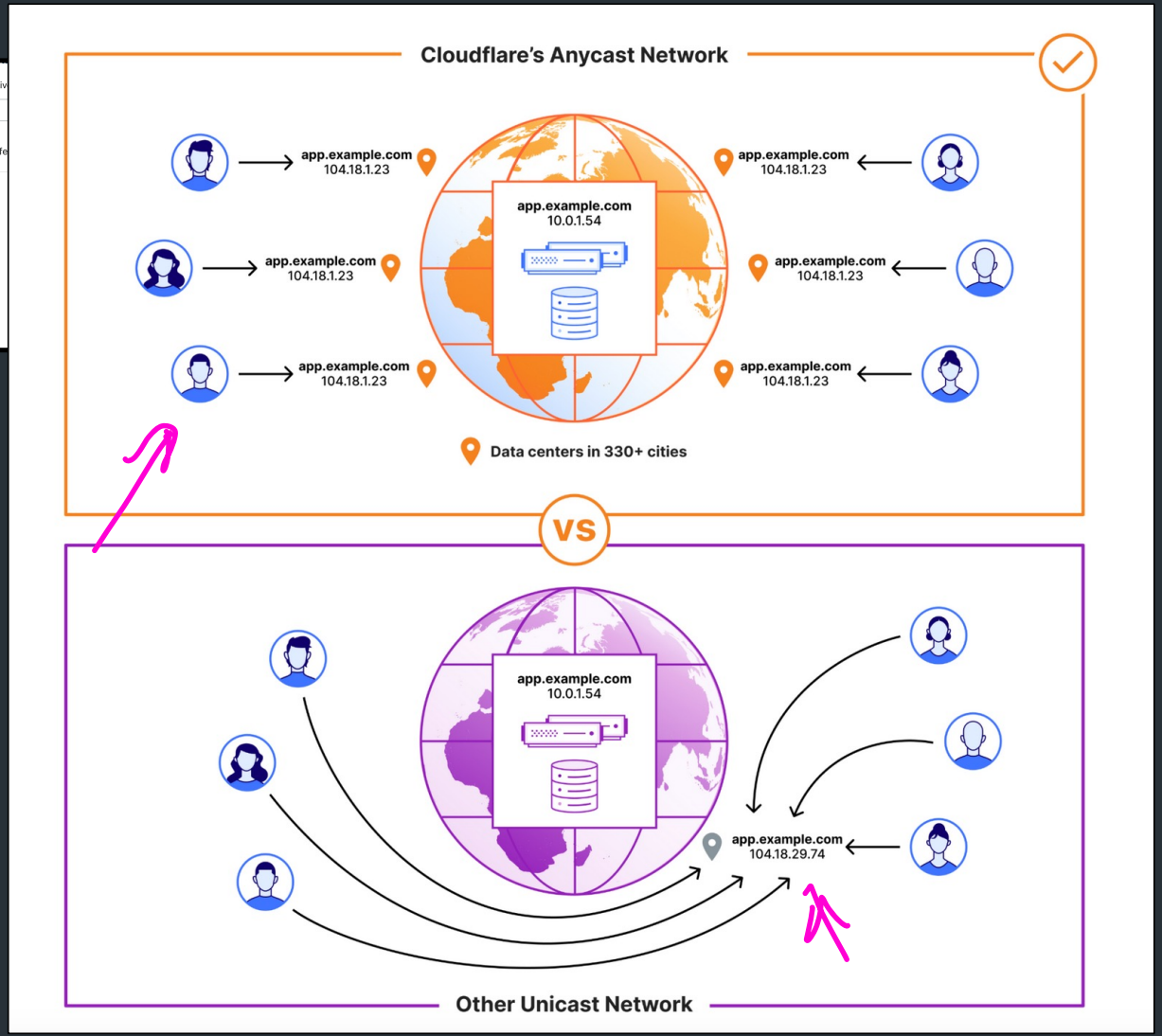
Subscribe to receiv

Email Address

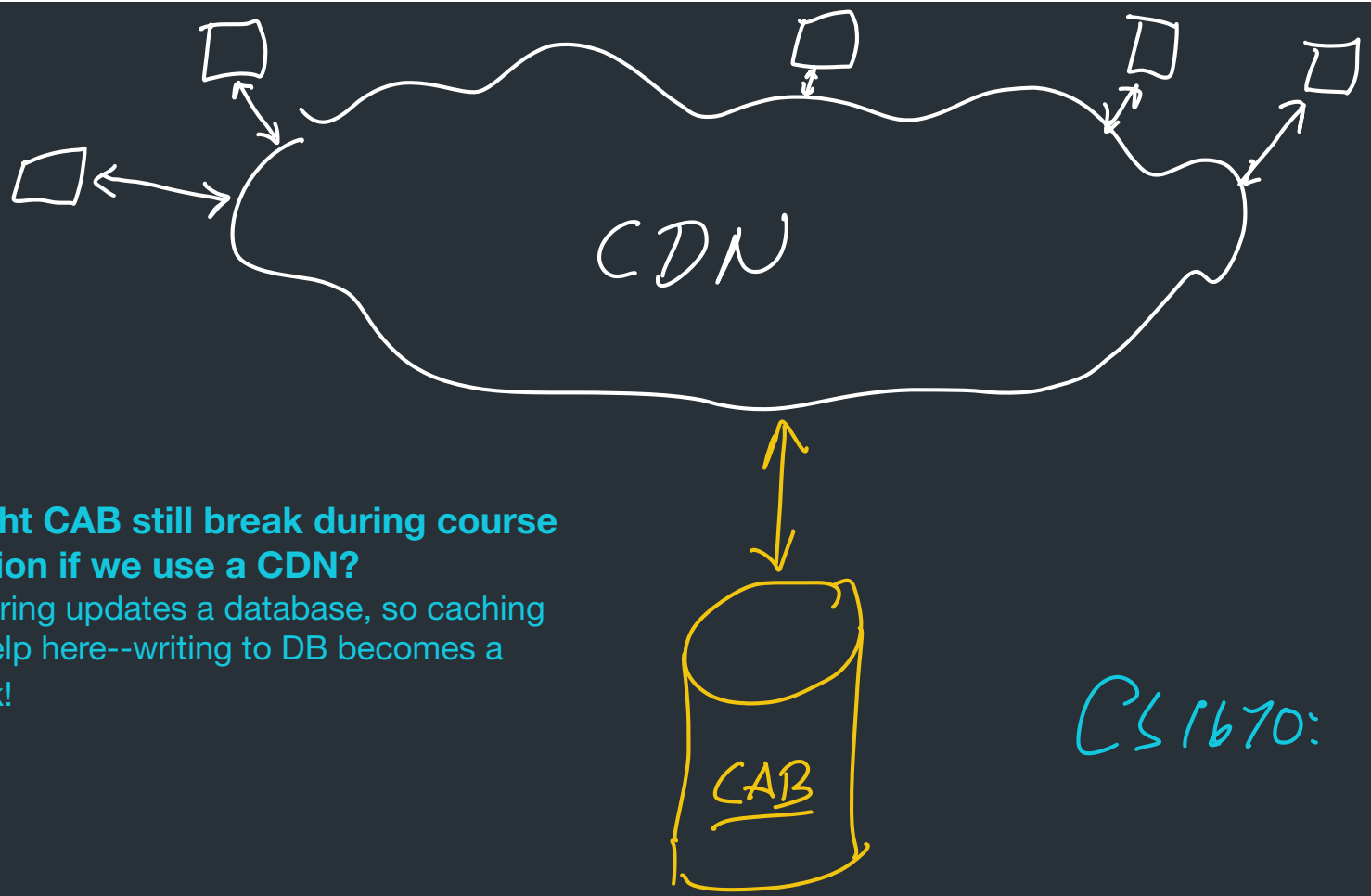
Product News | Speed & Reliability | Security | Zero Trust | Developers | AI | Policy | Partners | Life

How Cloudflare auto-mitigated world record 3.8 Tbps DDoS attack

2024-10-02



Link



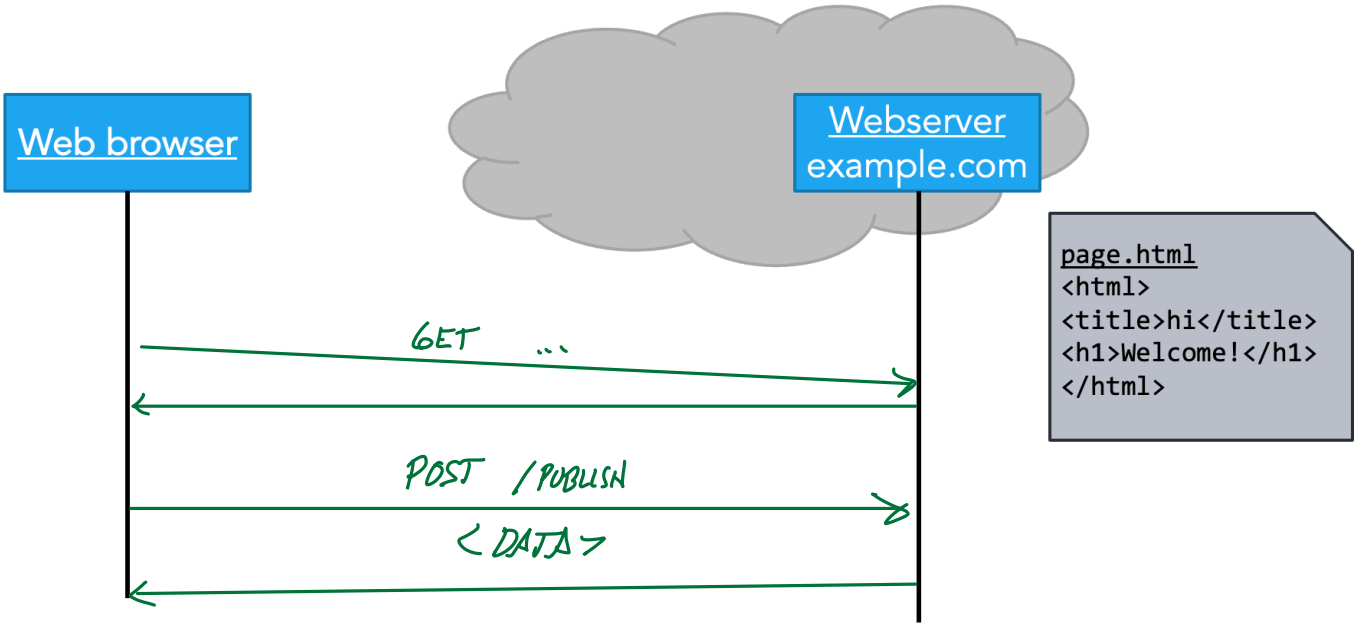
Why might CAB still break during course registration if we use a CDN?

=>Registering updates a database, so caching doesn't help here--writing to DB becomes a bottleneck!

CS1670: ~~160~~ 159 SEATS LEFT

HTTP: What more do we need?

The story so far:

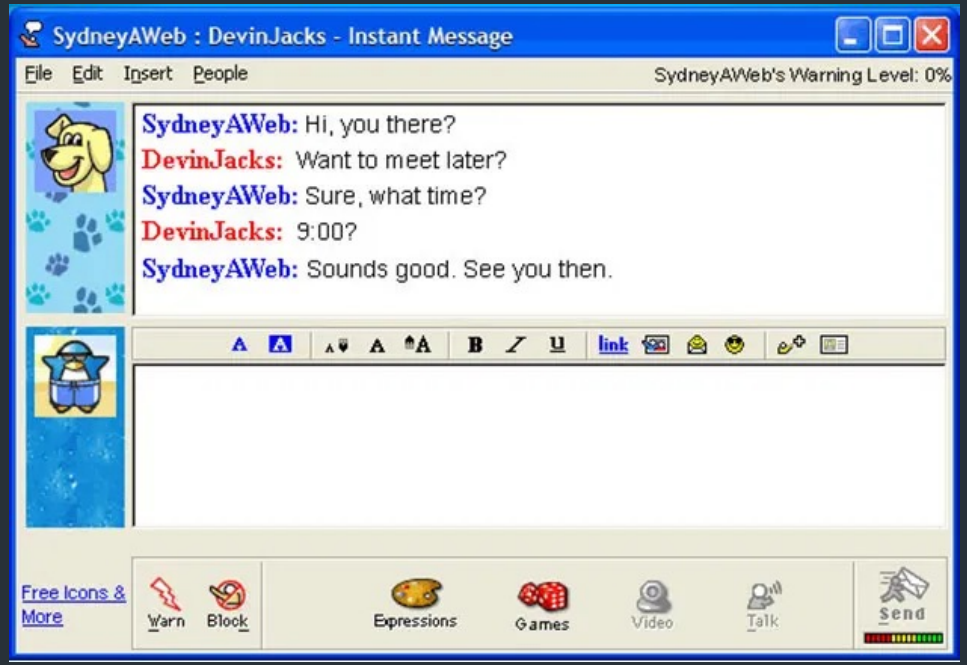


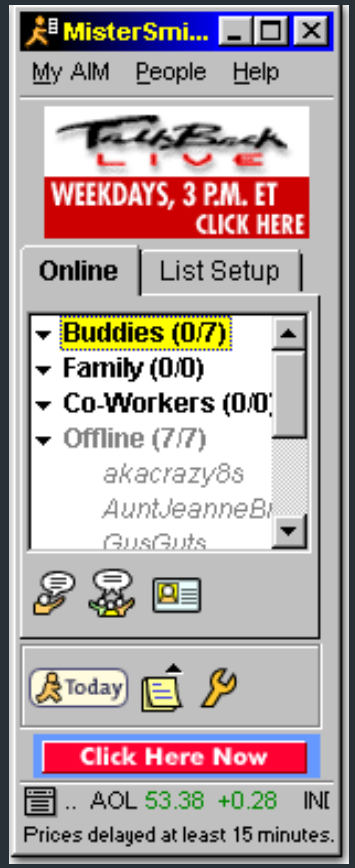
Goal: make a chat client
Idea: when a message received, server tells clients

Why is this hard with what we know about HTTP????
=> HTTP has a "pull" model: clients request info from serve
=> Server can't easily send something to client asynchronously

Example: Instant Messaging (~2005)

=> IMPLEMENTED AS ONE TCP CONNECTION TO SERVER



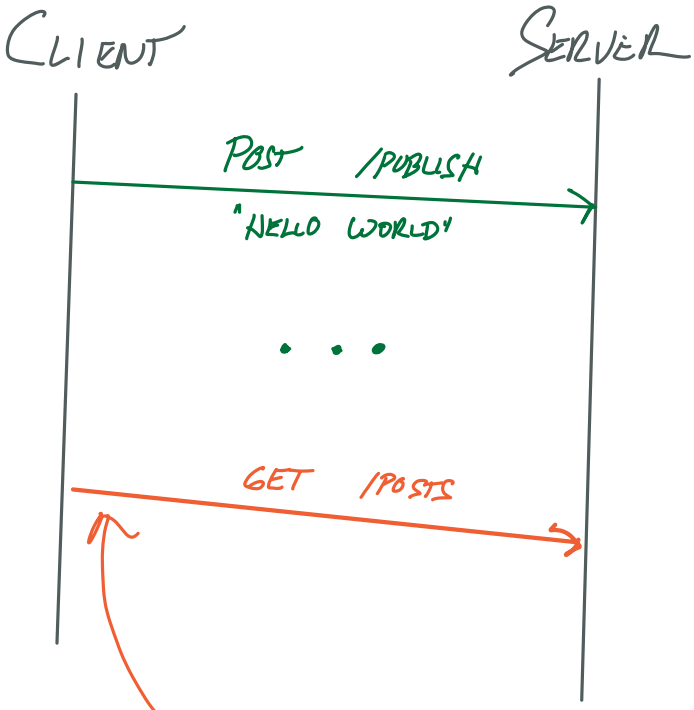


Examples: AIM, ICQ, Yahoo Messenger, MSN Messenger, ...

=> Relies on server sending messages to client asynchronously
=> We build web applications that do this all the time => how??

When does this model not work?

How might we implement this using what we currently know about HTTP?



At some point, client would need to periodically poll the server for new posts!
=> Slow to update, or sends a large number of requests

How to wait for info from the server?

One way: Polling

```
for {
      resp, err := doRequest("http://example.com/do-you-have-my-data")
      if resp != nil {.
      doThing(resp)
      }
      time.Sleep(1 * time.Second)
}
```

Lots of load on server
Lots of CPU on client
=> Lower rate => slower updates, less responsive

Another way: long polling

```
for {
      resp, err := doRequest("http://example.com/do-you-have-my-data")
      // ^ Assume this will block for very long time

      doThing(resp)
}
```

Server blocks until data is available
As soon as it returns, client requests again
=> Requires server to hold connection open with long timeout, server might not want to do this

Another way: HTTP/2: HTTP server push
=> Built in way for server to send data
=> Not really used, hard to implement in HTTP

Another way: Websockets (RFC6455, 2011)

Internet Engineering Task Force (IETF) — I. Fette
Request for Comments: 6455 — Google, Inc.
Category: Standards Track — A. Melnikov
ISSN: 2070-1721 — Isode Ltd.
December 2011

The WebSocket Protocol

Abstract

The WebSocket Protocol enables two-way communication between a client running untrusted code in a controlled environment to a remote host that has opted-in to communications from that code. The security model used for this is the origin-based security model commonly used by web browsers. The protocol consists of an opening handshake followed by basic message framing, layered over TCP. The goal of this technology is to provide a mechanism for browser-based applications that need two-way communication with servers that does not rely on opening multiple HTTP connections (e.g., using XMLHttpRequest or <iframe>s and long polling).

=> Bidirectional communication between client and server
=> Starts over HTTP
=> Gets back some of the semantics of a raw TCP socket

Starts with an HTTP request!

```
GET /chat
Host: javascript.info
Origin: https://javascript.info
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Key: Iv8io/9s+lYFgZWcXczP8Q==
Sec-WebSocket-Version: 13
```

Client asks server to "Upgrade" this connection to websocket
=> Server responds with "101 switching protocols"
=> This TCP connection now repurposed for websocket protocol

This allows us to get the semantics of a new protocol, but while still starting off using HTTP requests (which is what browsers, webservers, proxies, etc, are built to understand)

Examples

See recording for some fun demos:
- An example web application

Some responsive applications in practice:
- Edstem: uses websockets
- Google docs: uses another method where an HTTP request continually returns data (i.e., request remains open)

Next few pages are extra (optional) content we might discuss later!

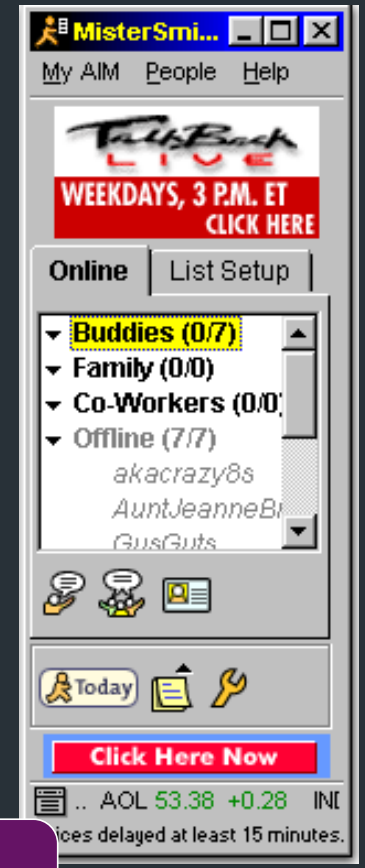
Returning to the early 2000s IM example...

- List of clients, when you get a message, send to each client

=> Server didn't buffer messages, store a history

=> Client had to be connected to receive messages!

Would this work today? Why not?



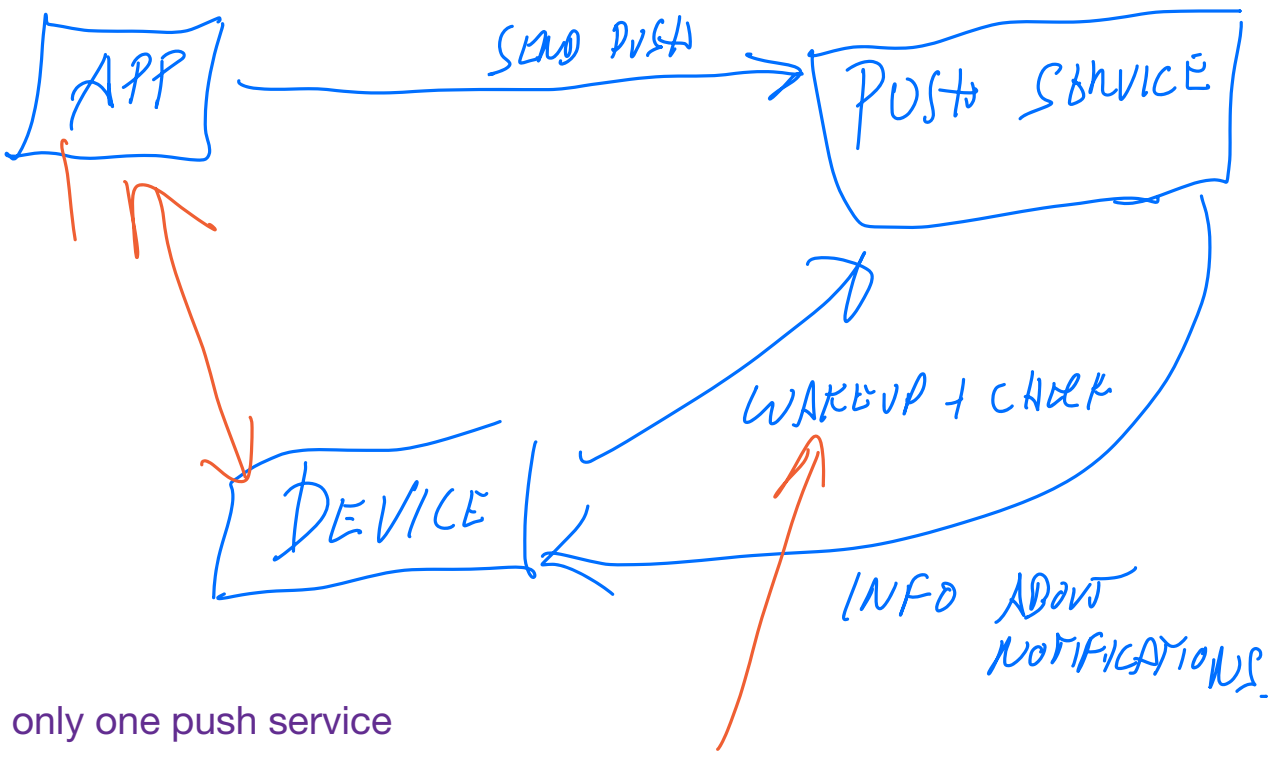
Old chat/IM applications: one TCP connection
=> Can we still do that?

Related: how do push notifications work?

On a mobile device: can't use persistent connections to get real-time messages when phone isn't online (when screen is not on)

Push notifications: service provided by OS to handle pushing events to an application
- iOS/Android matinatins this

Setup
- App developer needs to register application with service
- Each user's device is registered



- Use only one push service

=> device only needs to wake up once
=> OS can control how this works (instead of app)

=> Still update quickly, but preserves battery

~~PERIODIC~~, BUT OPTIMIZED.