Univerza v Mariboru

FERI
Fakulteta za elektrotehniko, računalništvo in informatiko

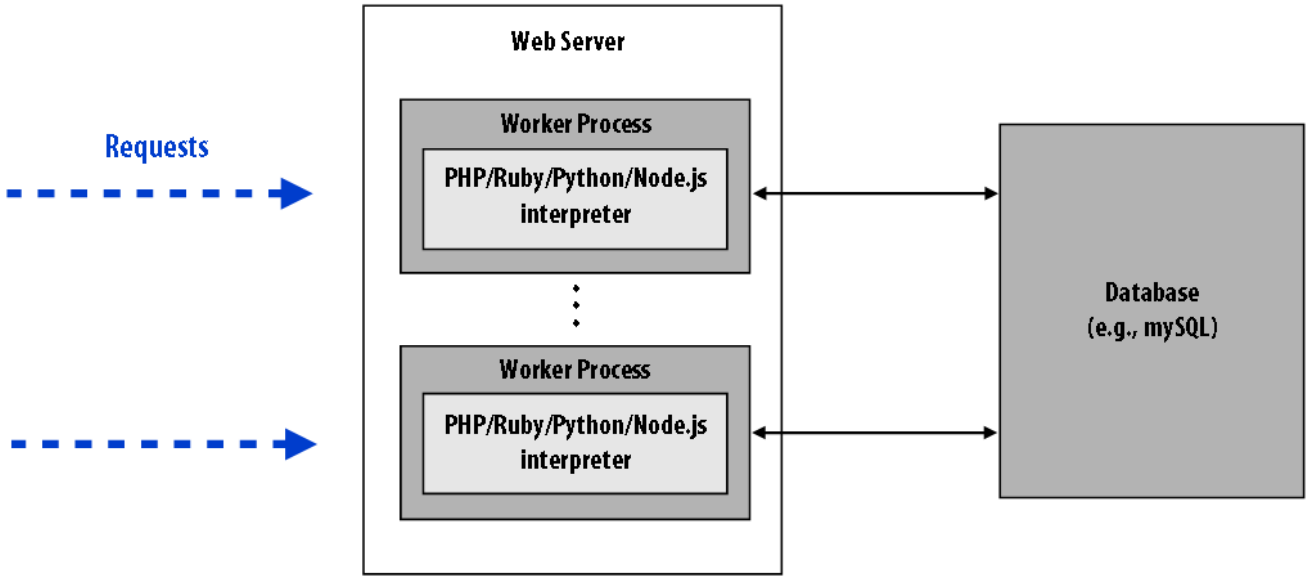
Spletne tehnologije

Pregled back-end spletnih rešitev

Niko Lukač

Uvod: tipična strežniška aplikacija danes…

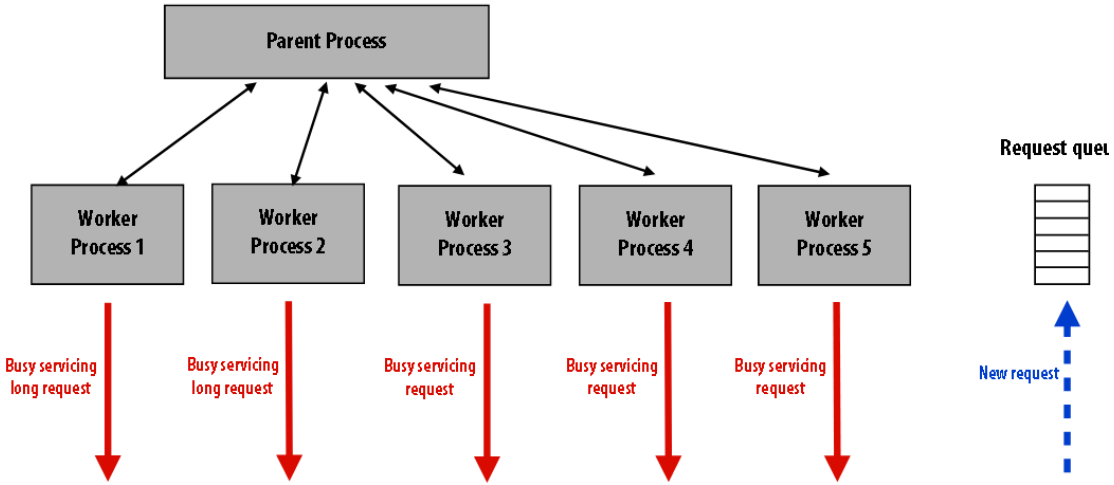
- Strežniške aplikacije so lahko **statične** ali **dinamične**, v odvisnosti od vsebine ki jo ponujajo odjemalcem. V realnosti za spletne aplikacije velikokrat koriščamo oba tipa strežniških aplikacij.
- Dinamične strežniške aplikacije nudijo **aplikacijski programski vmesnik** (API) do backend funkcionalnosti, pri čemer jih vedno več uporablja programski arhitekturni stil **REST** (Representational state transfer (REST))



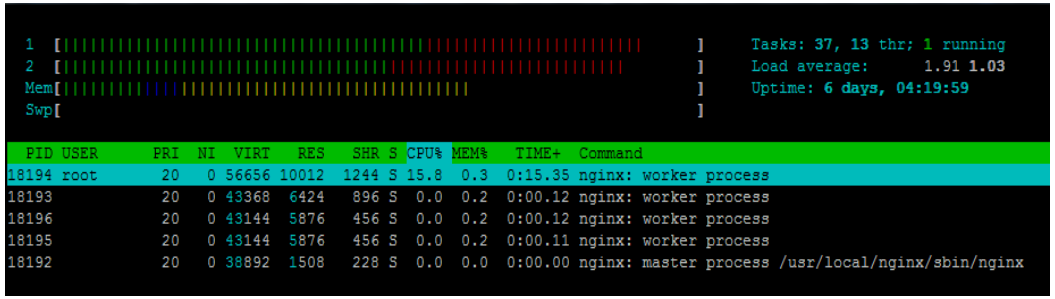
Vir slike: K. Fatahalian, CMU 15-418

Uvod: samodejni strežnik za statične podatke

- Tipični **statični podatki**: slike, dokumenti, tekst, video, audio, itd.
- Znani predstavniki: Apache, Nginx, lighttpd, IIS
- **Uporaba paralelizma**:
  - Maksimiziramo izkoriščanje CPE
  - Minimiziramo latenco (povprečni čas procesiranja zahtevka v ms)
- Zakaj dani strežniški aplikaciji kot sta npr. Nginx in Apache uporabljata **procese namesto večnitnost** za razdelitev bremena vhodnih zahtevkov (requests)?
  - Varnost: ne želimo, da ena nit zruši strežnik.
  - Omogočamo tudi klic zunanjih knjižnic, ki niso thread-safe.
  - Robustnost do uhajanja pomnilnika.



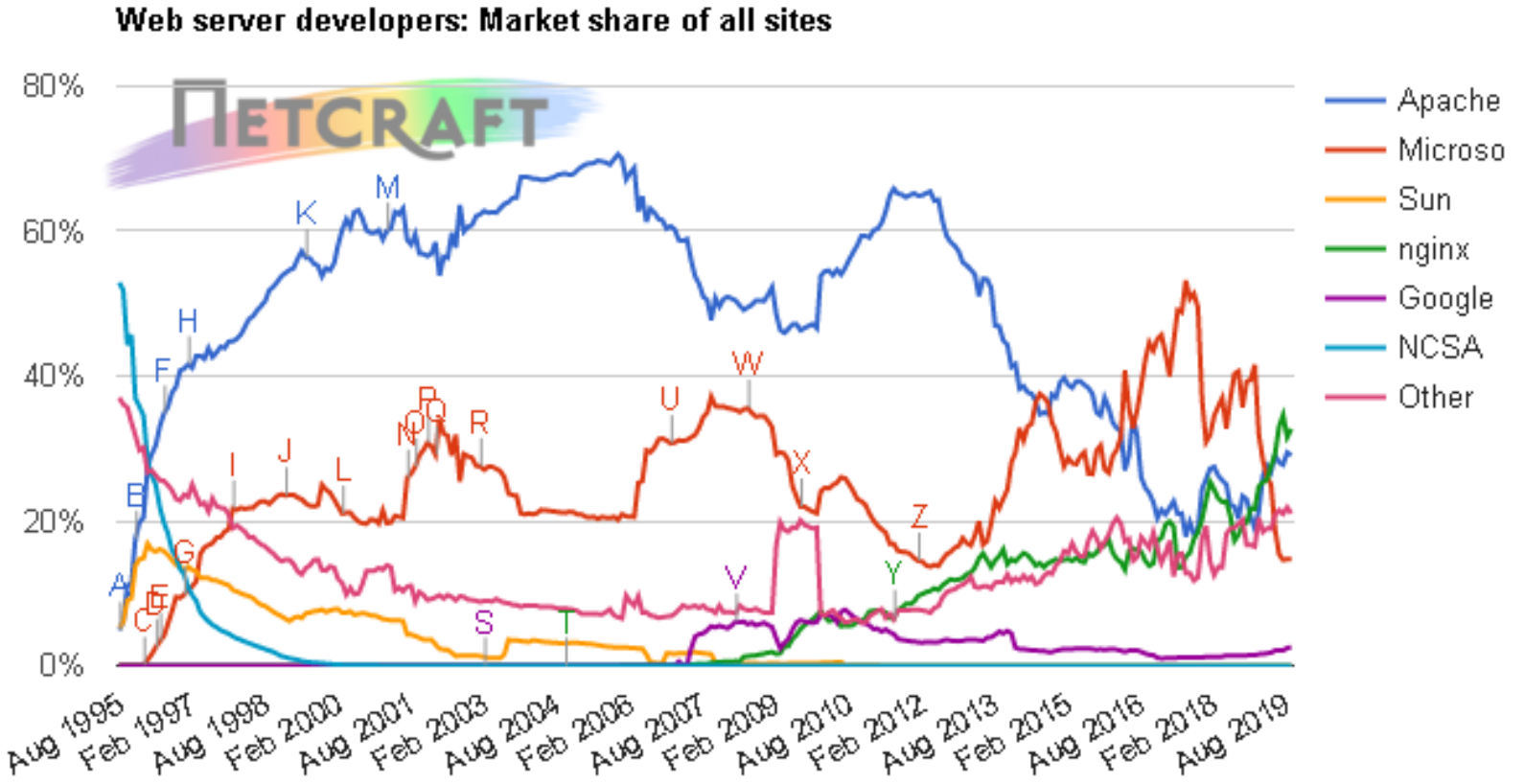
Vir slike: K. Fatahalian, CMU 15-418

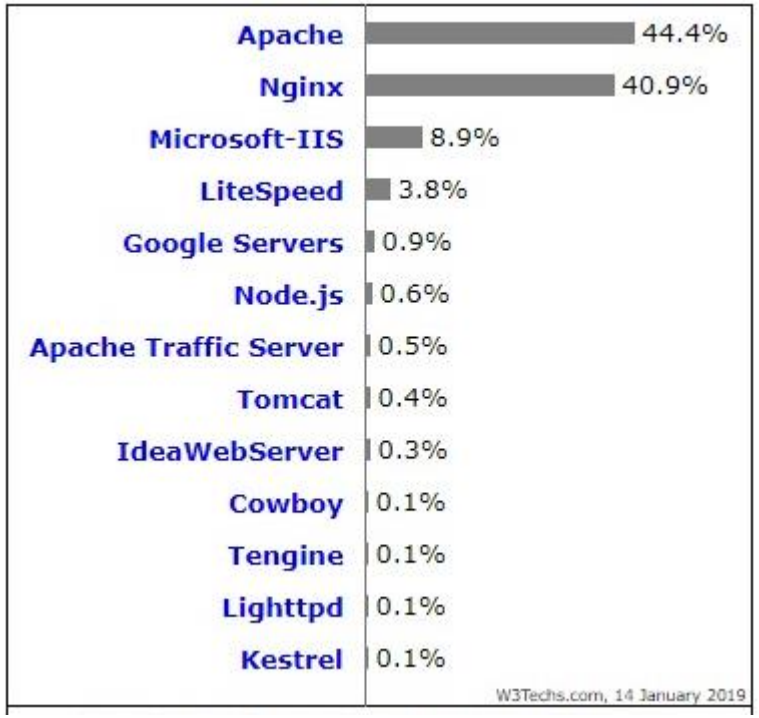


Uvod: dinamična strežniška aplikacija

- V “starih časih”: CGI (common gateway interface) funkcionalnost preko Nginx,Apache,Lighttpd itd., kjer smo klicali zunanje interpreterje (npr. za PHP) – slabosti?
- Danes: Node.js, .NET core, PHP/Hack HHVM, Django, Java Spring
- Zakaj se je Node.js (npr. z express-js) dobro obnesel v industriji ?
  - **Uporabljamo JS** za kodiranje backend (enako kot za frontend)
  - JS nam omogoča veliko različnih **modulov** (knjižnic) za hitro integracijo (**npm**)
  - **Performansa**: uporabljamo vse optimizacije, ki jih nudi **V8 JS pogon** (smo že spoznali), asinhrono dogodkovno delovanje (event-driven), npr. v starejših PHP ver. imamo le sinhrono procesiranje dogodkov (PHP 8 JIT dobro rešuje..)
  - Dobra podpora za grajenje **RESTful** aplikacij z uporabo **JSON** (enostavna integracija v JS objekte)
  - **Limitacije**: v osnovi single-core procesiranje (sicer obstajajo rešitve z Worker niti)

Uvod: katere strežniške aplikacije uporabljamo danes ?

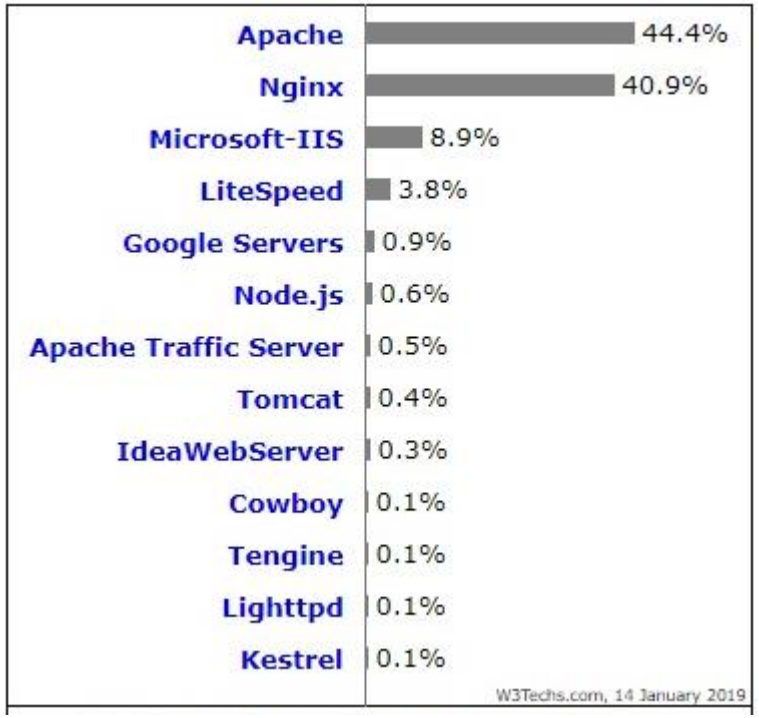




- Zakaj takšna situacija?

Uvod: katere strežniške aplikacije uporabljamo danes ?

- Zakaj takšna situacija?
  - Veliko spletnih stranih uporablja kompletne rešitve (npr. **WordPress** uporablja PHP), kjer se običajno uporablja Apache ali Nginx strežnik v ozadju
  - Večino prometa na spletu so **statične datoteke** (audio, video, itd.) za kar so strežniki za statične vsebine (npr. Nginx) bolj primerni
  - **Google** ima svoje **lastne strežnike** in prevzema skoraj 1% vseh strežnikov na internetu (spomnimo, da sta google.com in youtube.com najbolj iskani strani na svetu)
  - Večjo **zaupanje starejšim rešitvam** (Apache, Nginx), ki so že "preživeli" velik del napadov in so zato robustnejši ter že bili testirani v različnih okoljih. Npr. Apache tudi podpira ogromno "modulov", ki razširijo funkcionalnost spletnih strani – slabost: "debelina".
  - **IIS** je privzet strežnik za **MS strežniški OS**, omogoča enostavno konfiguracijo in uporabo napram odprtim UNIX rešitvam (npr. Apache ima lahko konfiguracijo z več sto vrstic)



Obnovimo znanje: RESTful spletne aplikacije?

- **REST** (Representational state transfer) je programski arhitekturni stil, ki se lahko uporablja pri HTTP komunikaciji (govorimo **HTTP REST**).
- Spletna aplikacija je **RESTful**, če podpira REST
- Klasična rešitev za grajenje **API** vaše strežniške storitve/aplikacije
- V HTTP REST so **viri/resursi** navedeni z URI, pri čemer imamo hierarhijo dostopa
- Klasične **akcije preko HTTP zahtevkov** ?
- Odjemalci poskrbijo za pravilno dostopanje do virov, vsaka REST poizvedba je neodvisna od ostalih (**stateless**)
- **Alternative** HTTP REST ?

Obnovimo znanje: RESTful spletne aplikacije?

- **REST** (Representational state transfer) je programski arhitekturni stil, ki se lahko uporablja pri HTTP komunikaciji (govorimo **HTTP REST**).
- Spletna aplikacija je **RESTful**, če podpira REST
- Klasična rešitev za grajenje **API** vaše strežniške storitve/aplikacije
- V HTTP REST so **viri/resursi** navedeni z URI, pri čemer imamo hierarhijo dostopa
- Klasične akcije preko HTTP zahtevkov: **GET** (branje vira), **POST** (kreiranje), **PUT** (posodabljanje), **DELETE** (brisanje)
- Odjemalci poskrbijo za pravilno dostopanje do virov, vsaka REST poizvedba je neodvisna od ostalih (**stateless**)
- **Alternative** HTTP REST: SOAP, XML-RPC (v tem primeru nimamo URI-jev)

Obnovimo znanje : RESTful spletne aplikacije?

- REST tipično uporabljamo kot **CRUD** (Create Read Update Delete), REST podpira še širše funkcionalnosti (npr. Hypermedia as the Engine of Application State, **HATEOAS**)
- Uporaba poizvedbeni niz (**query string**) – klasični način je še vedno dovoljen, le z a nastavljanje ali iskanje atributov resursa

HTTP command	Database operation	/dogs	/dogs/3
GET	Read	List all dogs	Get dog details
POST	Create	Create new dog	–
PUT	Update	–	Update detail/s
DELETE	Delete	Delete all dogs	Delete this dog

```
PUT http://dog-db/api/dogs/3
name=Fifi&type=poodle

{
  id:3,
  name:”Fifi",
  dob:"2009-05-21",
  type:”poodle”,
GET http://dog-db.com/api/dogs/3

{
  id:3,
  name:"Spot",
  dob:"2009-05-21",
  type:"spaniel",
  photo:"http://dog-db/images/…
```

Obnovimo znanje : RESTful spletne aplikacije?

- **Slaba praksa** REST (npr. Twitter API):
  - POST statuses/destroy/:id
  - GET statuses/show/:id
  - POST direct_messages/new

- Boljši način (**simplifikacija**):
  - DELETE status/:id
  - GET status/:id
  - POST direct_message ali PUT direct_message/:id

Skaliranje strežniške aplikacije ?

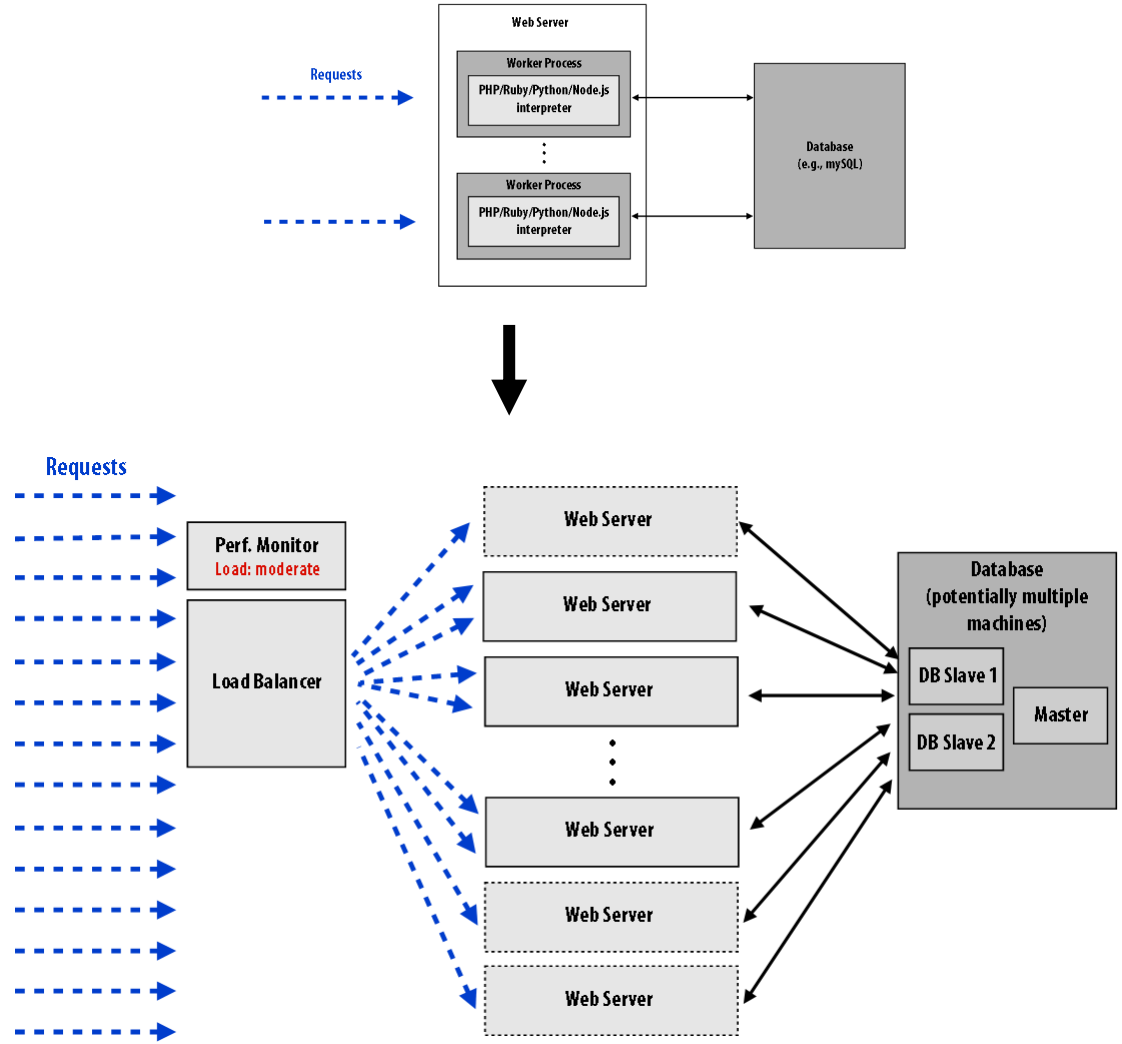
- Namen ?
  - Več procesorske moči ?
  - Več pomnilnika ?
  - Znižamo latenco ?
  - Povečamo prepustnost ?
- Običajno želimo vse dane pohitritve
- Vertikalno skaliranje:
  - ???
- Horizontalno skaliranje:
  - ???

Skaliranje strežniške aplikacije ?

- Namen ?
  - Več procesorske moči ?
  - Več pomnilnika ?
  - Znižamo latenco ?
  - Povečamo prepustnost ?
- Običajno želimo vse dane pohitritve
- Vertikalno skaliranje:
  - + CPU, + RAM, itd
- Horizontalno skaliranje:
  - Več strežniške strojne opreme

Moderne tehnike skaliranja

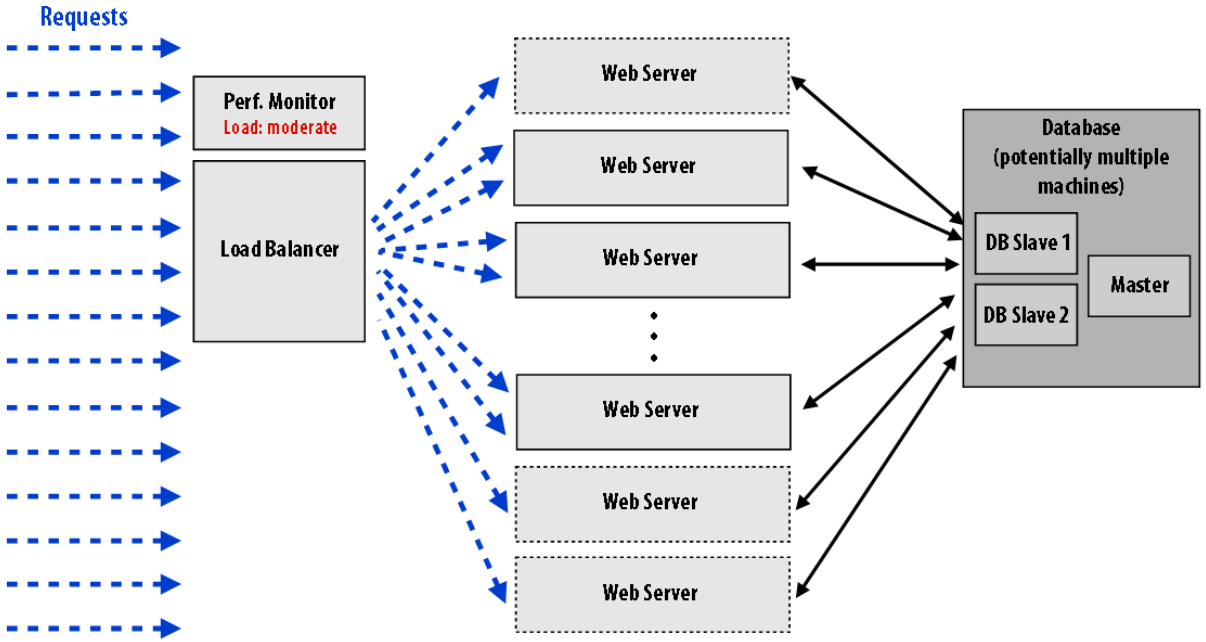
- Danes lahko vertikalno in horizontalno skaliranje učinkovito počnemo virtualno s pomočjo oblačnih storitev (npr. AWS).
- V vsakem primeru moramo iz programskega in strojnega nivoja razmislit **o topologiji "lokalnega omrežja"**, ki definira spletno aplikacijo, ki jo skaliramo
- **Potrebno se je zavedat različnih strategij skaliranja**
  - **Uravnoteženje prometa (load balancing)**
  - **Predpomnenje (caching)**
  - **CDN (content based distribution) in DNS**



Vir slike: K. Fatahalian, CMU 15-418

Uravnoteženje bremena (load balancing)

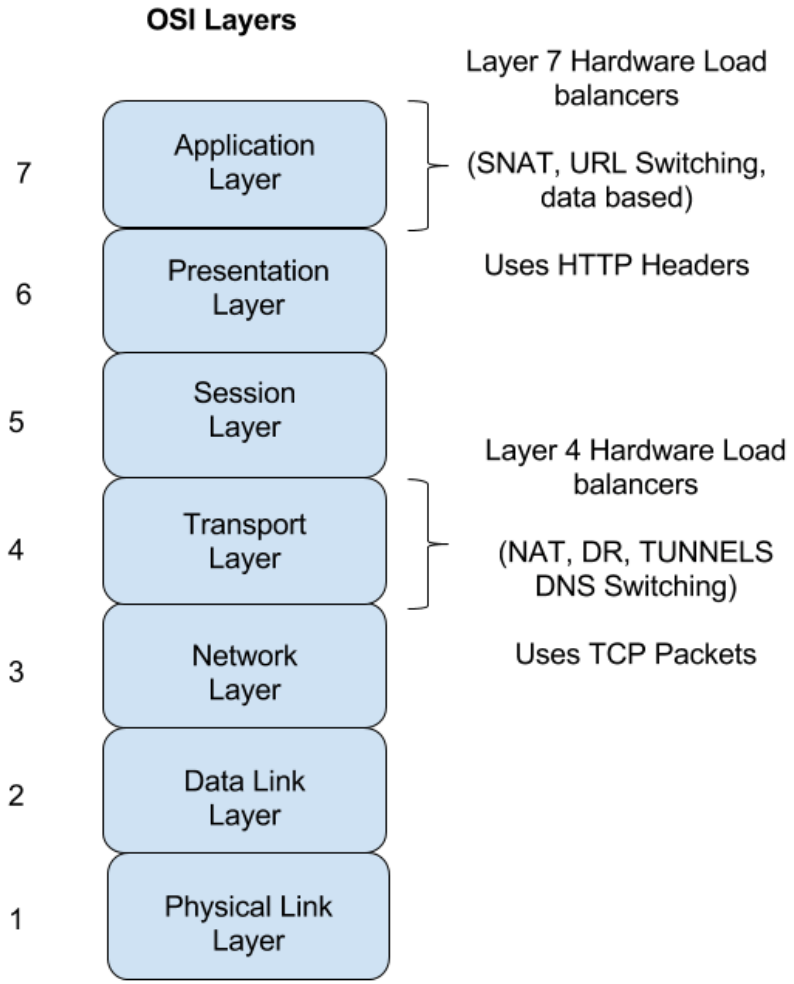
- **Breme razdelimo na več neodvisnih strežnikov**
- **Različne strategije** uravnoteženja – kateri strežnik izberemo za procesiranje poizvedbe?:
  - **Round robin** (krožna vrsta)
  - **Weighted round robin** (utežena krožna vrsta, uteži so lahko različne prioritete)
  - **Least connection** (najmanj povezav)
  - **Least response time** (najbolj odziven – latenca)
  - **Least bandwidth** (najmanj obremenjen, npr. glede porabe mrežnega prometa)
  - **IP hash** (iz IP odjemalca izračunamo sekljalno funkcijo ter tako izberemo strežnik)



Vir slike: K. Fatahalian, CMU 15-418

Load balancing strategije

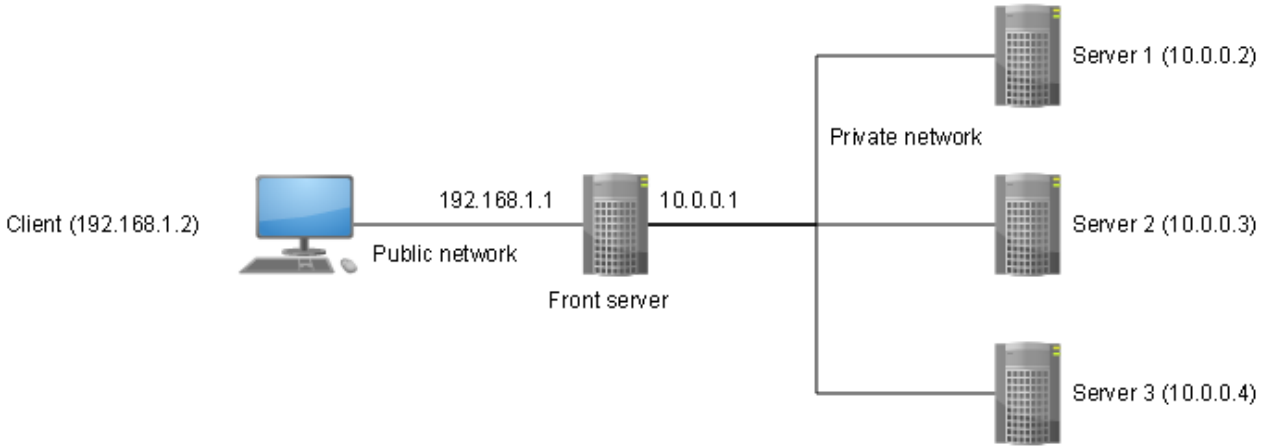
- Load balancing lahko uvedemo na različnem mrežnem sloju (če upoštevamo npr. **OSI mrežne sloje**).
- Na katerih **dveh slojih se običajno izvaja load balancing** pri spletnih aplikacijah ?
  - **Sloj 7: HTTP**
  - **Sloj 4: TCP**
- Load balancing na HTTP:
  - **Izkoriščamo REST** način naslavjanja GET zahtevkov
  - Možno **filtriranje** glede vsebine & glave ali seje (npr. z regularnimi izrazi), seja bo persistirala
  - **Počasnejše** od load balancing na TCP paketkih

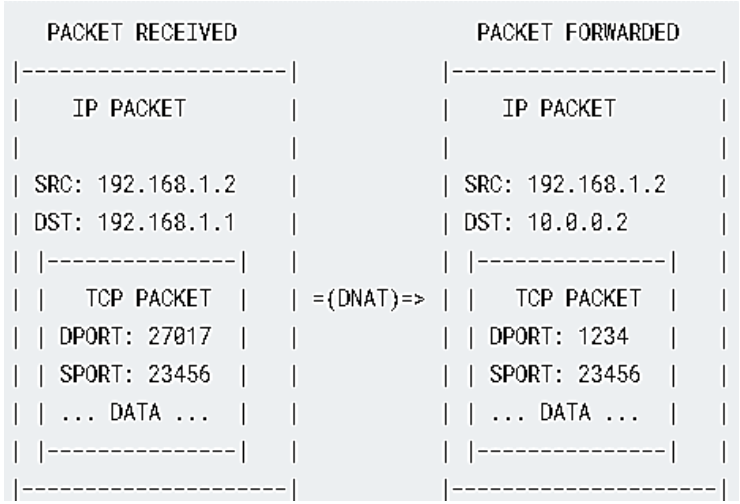


Load balancing rešitve

- Primer HTTP load balancing z **NGINX Plus** (komercialna razširitev):
  - https://nginx.org/en/docs/http/load_balancing.html
  - Še na TCP nivoju: https://docs.nginx.com/nginx/admin-guide/load-balancer/tcp-udp-load-balancer/

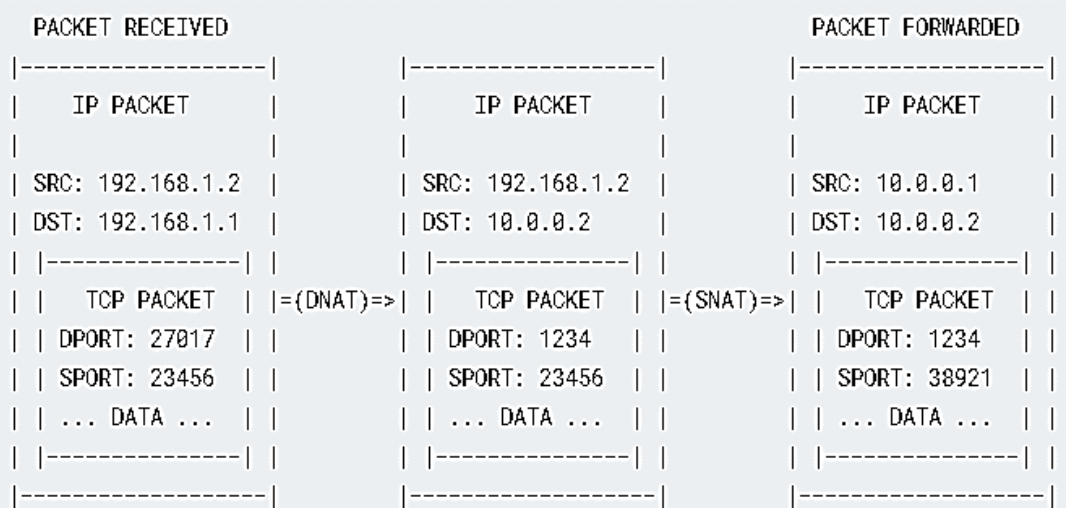
- Na TCP nivoju lahko tudi uporabljamo npr. znano rešitev kot je **iptables, nftables, itd. (UNIX)**
  - Izkoriščamo virtualno omrežje (**NAT** – network address translation), paketkom menjamo IP destinacije





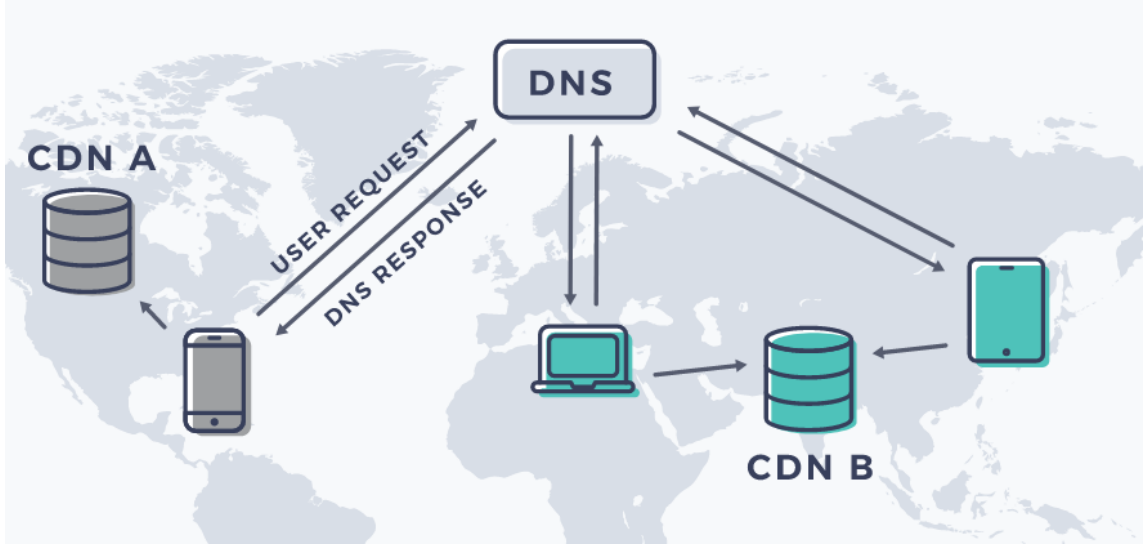
Load balancing z Linux iptables (primer)

- 1. Postavimo **DNAT** pravilo, ki preusmeri promet iz vhodne strežnika na interne strežnike
- 2. Redirekcija (**DNAT->SNAT**), pri čemer uporabimo strategijo round robin (primer za redirekcijo na strežnik 10.0.0.2):
  - iptables -A PREROUTING -t nat -p tcp -d 192.168.1.1 --dport 27017 -m statistic --mode nth --every 3 --packet 0 -j DNAT --to-destination 10.0.0.2:80



Content delivery network in DNS

- Na **DNS** (domain name service) nivoju domene ali poddomene nastavimo več različnih IP naslovov, ki so **geografsko različno porazdeljeni**.
- DNS protokol omogoča, da bo odjemalec dobil IP najbližjega geografsko nahajočega strežnika – s tem težimo k **zmanjšanju latence** (spomnimo ping!)
  - Ti. izraz **geographic balancing**
- Odprtokodne rešitve:
  - Powerdns
- Znani ponudniki storitev:
  - Cloudflare
  - Amazon CloudFront
  - Google

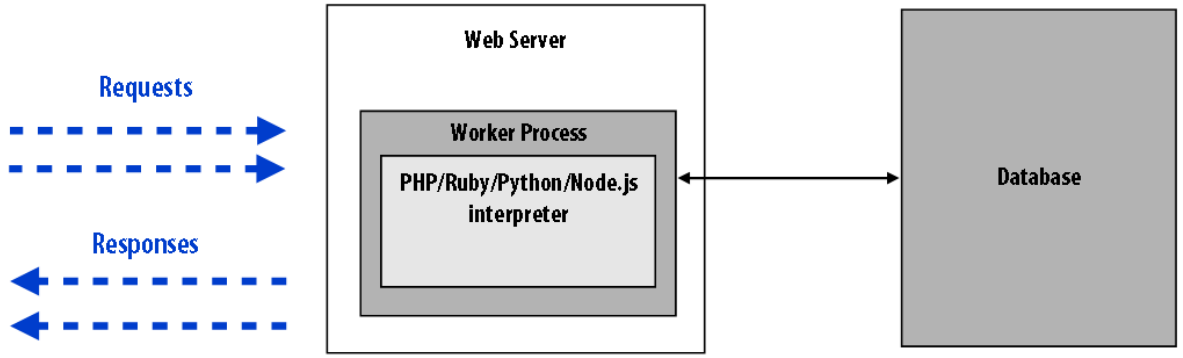


Content delivery network in DNS

- Kako pa DNS strežnik ve kje se geografsko nahaja nek IP ?
- **Traceroute**: vemo iz katerega segmenta (avtonomnega Sistema) internet omrežja komuniciramo, ter zakasnitvene čase. Pri tem se lahko uporabijo **BGP usmerjevalne tabele**.
- Geografska baze IP naslovov (velikokrat dobimo z **whois** poizvedbo nad IP) vodijo znane organizacije za dodeljevanje IP naslovov:
  - African Network Information Centre (AfriNIC)
  - American Registry for Internet Numbers (ARIN)
  - Asia-Pacific Network Information Centre (APNIC)
  - Latin American and Caribbean Internet Address Registry (LACNIC)
  - RIPE Network Coordination Centre (RIPE NCC)

Predpomnenje (caching)

- Spomnimo klasično delovanje dinamičnih strani, ki pretakajo podatke preko podatkovne baze (npr. MongoDB ali MySQL)



Vir slike: K. Fatahalian, CMU 15-418

- Predpomenenje nam omogoča **začasno hranjenje rezultatov poizvedb**, kar nam omogoča **pohitreni dostop** ob naslednji uporabi (v primeru kompleksnih poizvedb je še to posebej pomembno)
- Primer tipične PHP kode:

```
$query = "SELECT * FROM users WHERE username='kayvonf';
$user = mysql_fetch_array(mysql_query($userquery));

echo "<div>" . $user['FirstName'] . " " . $user['LastName'] . "</div>";
```

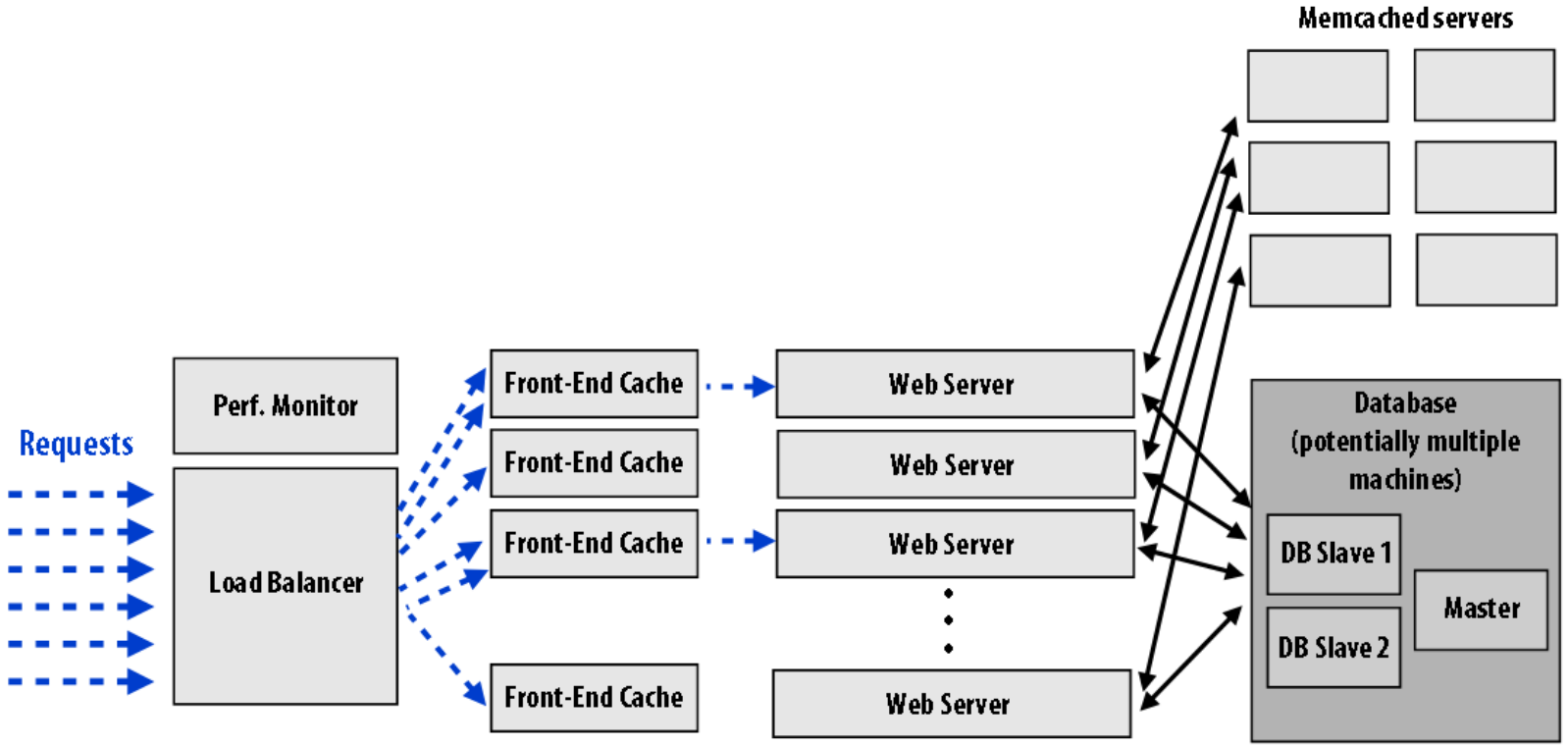
Memcached

- Znana rešitev za predpomenenje (v C-ju), **simplificirana uporaba** ter podprost v vseh dinamičnih jezikih za grajenje spletnih aplikacij.
- **Agnostičnost** glede vrste poizvedbe – vsebina se smatra kot tekstovna ali binarna.
- Za delovanje je potrebno zagnat **ločen memcached strežnik** (praviloma virtualni), ki uporablja **TCP ali UDP** protokol za komunikacijo
- Deluje na principu **sekljalne tabele** (**hash table oz. key-value store**), kjer posamezni objekt določa ključ (npr. poizvedbo) in vrednost rezultat predpomnenja
- Tipični primer uporabe memcached:

```
userid = $_SESSION['userid'];

check if memcache->get(userid) retrieves a valid user object

if not:
   make expensive database query
   add resulting object into cache with memcache->put()
   (so future requests involving this user can skip the query)

continue with request processing logic
```

Memcached

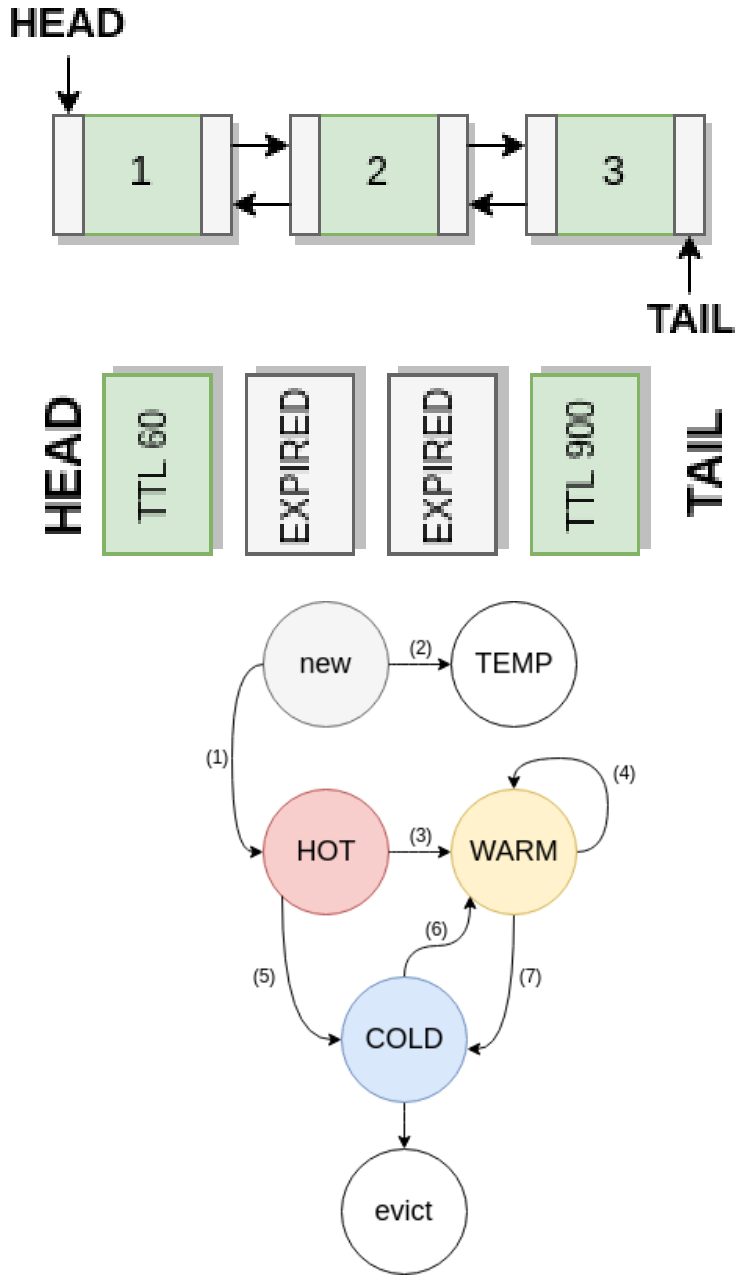
- Za doseganje višje skalabilnosti lahko **uporabimo več Memcached strežnikov** ter **balanciramo breme med njimi**.
- Predpomenenje je tudi **koristno za statične datoteke**, ki jih pošiljamo na frontend.



Vir slike: K. Fatahalian, CMU 15-418

Memcached - LRU

- Interno Memcached uporablja LRU (**Least recently used**) pomnilniški seznam, tako da se **starejši objekti sčasoma pobrišejo** (sicer bi lahko porabili preveč pomnilnika za doseganje predpomnenja, tipično predpomnimo cca. **5% od vseh možnih poizvedb**)
- LRU je implementiran kot **dvojno povezan seznam**, nove objekti gredo na začetek seznama, starejši se brišejo na repu seznama.
- **Brisanje** se doseže v primeru: namernega brisanja, prepisanja, **pretekel določen čas (TTL, privzeto 0=neskončno)**, ali pa pride do primankovanja pomnilnika
- Memcached dodatno uporablja ti. **segmentiran LRU**, kjer se objekti označijo za vroče, tople ali mrzle glede frekvenčnosti uporabe (uporabimo 3 LRU sezname).



Vir slike: https://memcached.org/blog/modern-lru/

Memcached - LRU

- Primer **interne implementacije** Memcached objekta (v C-ju):

```
typedef struct _stritem {
    /* Protected by LRU locks */
    struct _stritem *next;
    struct _stritem *prev;
    /* Rest are protected by an item lock */
    struct _stritem *h_next;    /* hash chain next */
    rel_time_t      time;       /* least recent access */
    rel_time_t      exptime;    /* expire time */
    int             nbytes;     /* size of data */
    unsigned short  refcount;
    uint8_t         nsuffix;    /* length of flags-and-length string */
    uint8_t         it_flags;   /* ITEM_* above */
    uint8_t         slabs_clsid;/* which slab class we're in */
    uint8_t         nkey;       /* key length, w/terminating null and padding */
    /* this odd type prevents type-punning issues when we do
     * the little shuffle to save space when not using CAS. */
    union {
... // scr: cas
        char end; // scr: flexible array member indicating the item header "end"
    } data[];
    /* if it_flags & ITEM_CAS we have 8 bytes CAS */
    /* then null-terminated key */
    /* then " flags length\r\n" (no terminating null) */
    /* then data with terminating \r\n (no terminating null; it's binary!) */
} item;
```

Redis

- Redis (**remote dictionary server**), implementiran v C-ju, je glavni konkurent Memcached, ki omogoča več funkcionalnosti za predpomnenje.
- Za razliko od Memcached LRU se uporablja LFU (**Least frequently used**)
- Ukazi: https://redis.io/commands

	Memcached	Redis
(multi)get	✓	✓
(multi)set	✓	✓
incr/decr	✓	✓
delete	✓	✓
Expiration	✓	✓
prepend/append	✓	
Range Queries		✓
Data Types!		✓
Persistence	**(sorta)**	✓
Multi-Threaded	✓	
Replication	**(sorta)**	✓

Skalabilnost v praksi

- Primer Facebook (CDN geo balancing -> load-balancing + memcached):
  - Zaradi performančnih vidikov predpomenenje statičnih datotek za frontend ne zahteva sinhronizacije seje itd., **možne povezave od zunaj**

Page URL:
https://www.facebook.com/photo.php?fbid=10151325164543897&set=a.10150275074093897.338852.722973896&type=1&theater

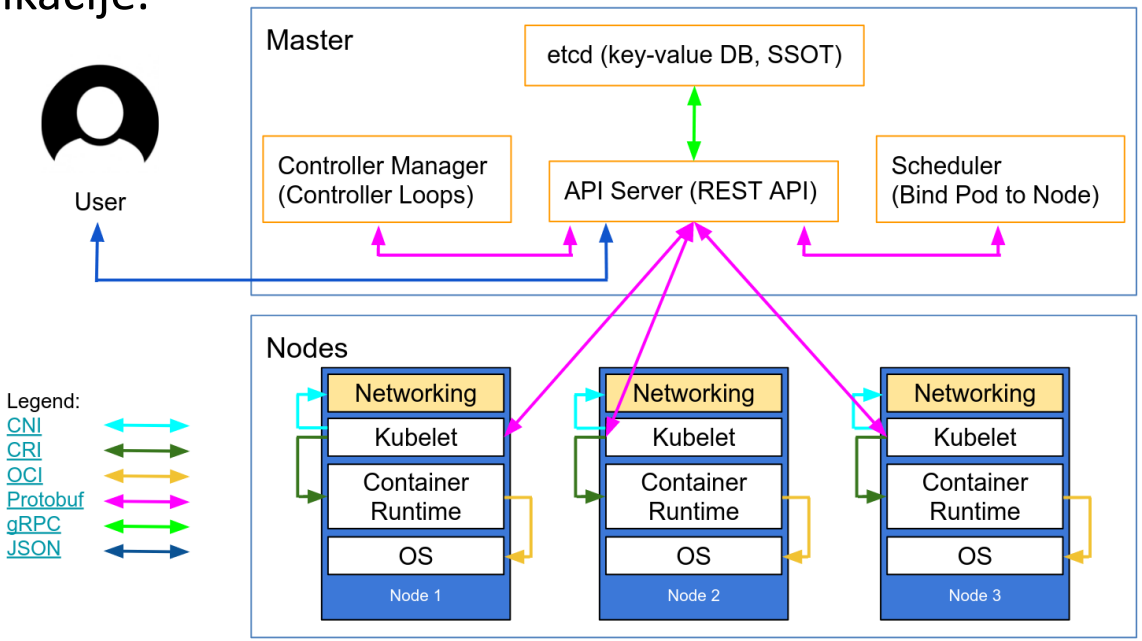
Image source URL:
https://sphotos-a.xx.fbcdn.net/hphotos-prn1/522152_10151325164543897_1133820438_n.jpg

Geo balancing strežnik xx

memcaced strežnik za slike prn1

Skalabilnost v oblaku ?

- **Elastičnost** (elastic web): npr. Docker lahka (lightweight) virtualizacija (na eni ali več fizičnih ali virtualnih strežnikih)
- Uporabimo npr. odprtokodno rešitev **Kubernetes** za doseganje visoke skalabilnosti
- Podprta rešitev npr. v DigitalOcean in AWS. Lažje **prilagajanje dinamični obremenitvi** spletne aplikacije.



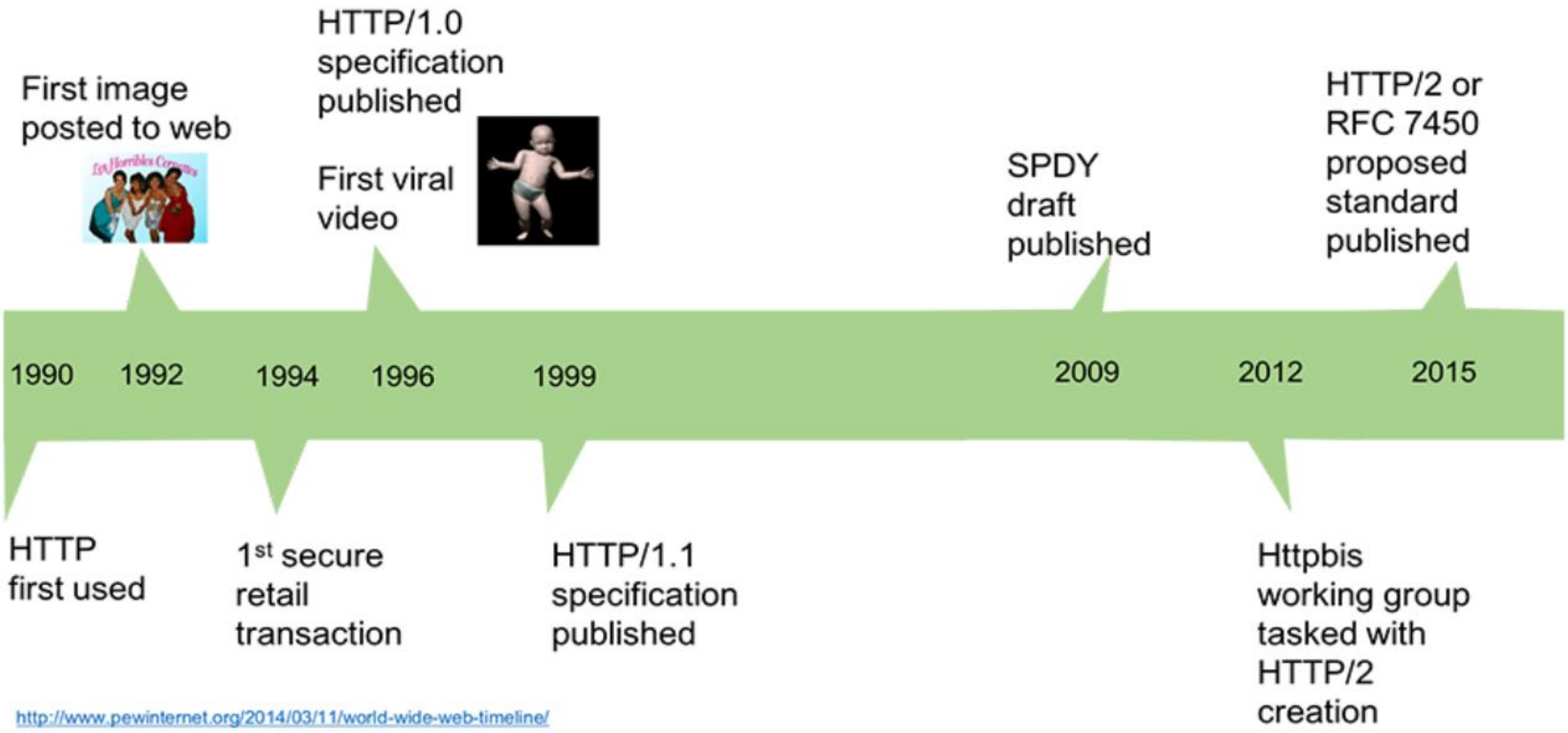
Vir slike: kubernetes.io

Latenca VS prepustnost (throughput)

- **TechEmpower benchmarks**: https://www.techempower.com/benchmarks/
  - Sistemski nivo dogodkov za mrežne vtičnike (sockets): npr. Linux Epoll VS Microsoft IOCP

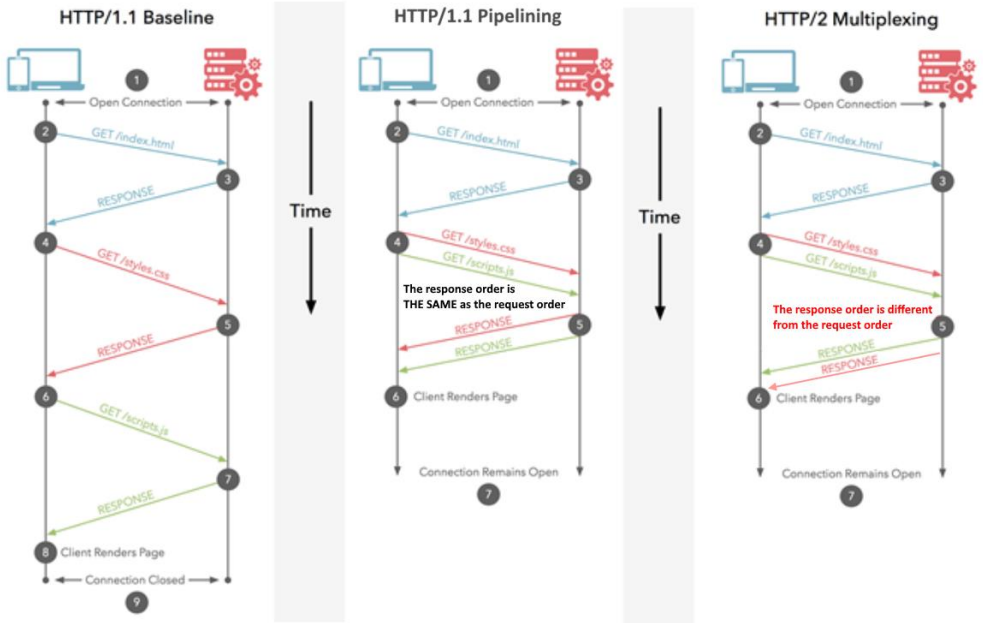
Moderni “HTTP” protokoli ?

- Kratka zgodovina…



HTTP 2.0 vs HTTP 1.1

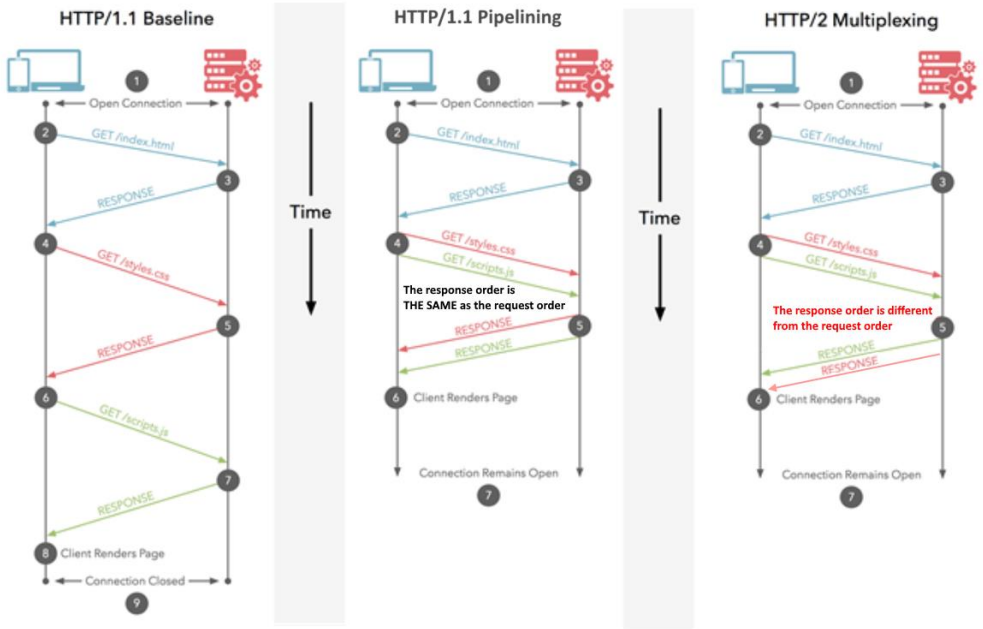
- Demo: https://imagekit.io/demo/http2-vs-http1
- HTTP/1.1 pipelining:
  - Slabosti ?
- HTTP/2.0 multiplexing:
  - Prednosti ?



Vir slike: https://developers.google.com/web/fundamentals/performance/http2

HTTP 2.0 vs HTTP 1.1

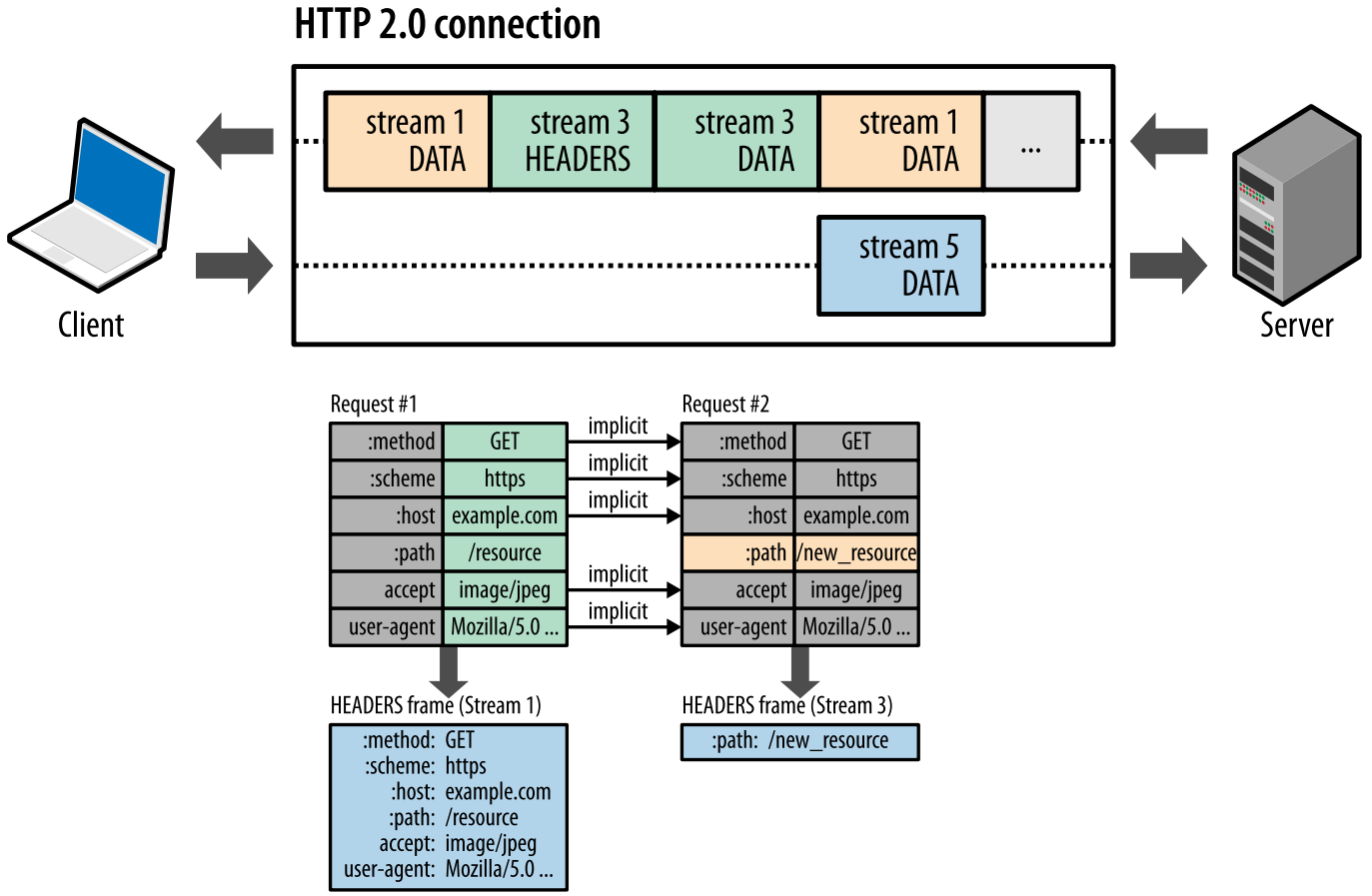
- Demo: https://imagekit.io/demo/http2-vs-http1
- HTTP/1.1 pipelining:
  - Slabosti ? first-in-first-out
  - Potrebno več TCP povezav
- HTTP/2.0 multiplexing:
  - Prednosti ? asinhronost in 1 TCP povezava



Vir slike: https://developers.google.com/web/fundamentals/performance/http2

HTTP 2.0 podatkovni tokovi (streams)

- **Binarni okvirji** vsebujejo **glavo in/ali telo HTTP zahtekov**. En podatkovni tok **(stream) ima lahko več okvirjev**.
- **Asinhrono + bidirektno**
- Podpora tudi **PUSH** mehanizma
- **Huffmanovo kodiranje** glave zahtevkov in
- **Reduciranje podvojenih metapodatkov**
- Danes podpira cca. 50% od top 10 mil. spletnih strani
- Prioritetna lestvica tokov ?



Vir slike: https://developers.google.com/web/fundamentals/performance/http2

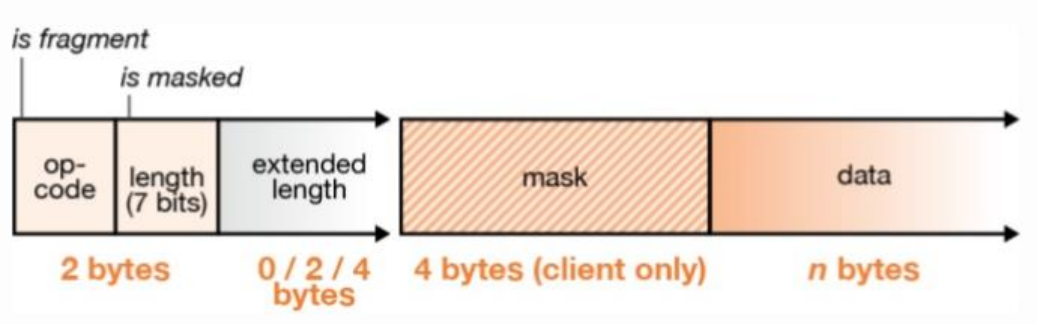
Websockets

- **Full-duplex protokol** (2010), ki je bil razvit namenoma **za realnočasovne front-end aplikacije**. Lahko deluje istočasno preko **obstoječe HTTP povezave**.
- Inicializacija (handshake):

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
Origin: http://example.com
```

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
```

- **Možen binarni prenos**. Vsebina se prenaša v **okvirjih**:
  - Zakaj **maskiramo** podatke?



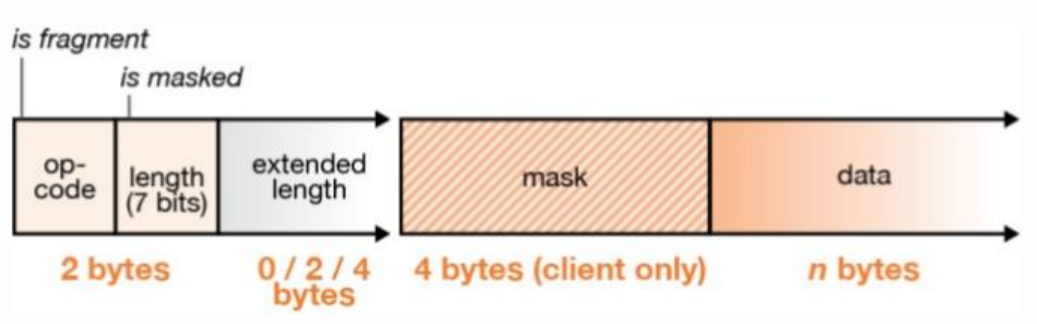
Websockets

- **Full-duplex protokol** (2010), ki je bil razvit namenoma **za realnočasovne front-end aplikacije**. Lahko deluje istočasno preko obstoječe HTTP povezave.
- Inicializacija (handshake):

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
Origin: http://example.com
```

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
```

- **Možen binarni prenos**. Vsebina se prenaša v **okvirjih**:
  - Zakaj **maskiramo** podatke?
    - Preprečimo napade (npr. cache poisoning)



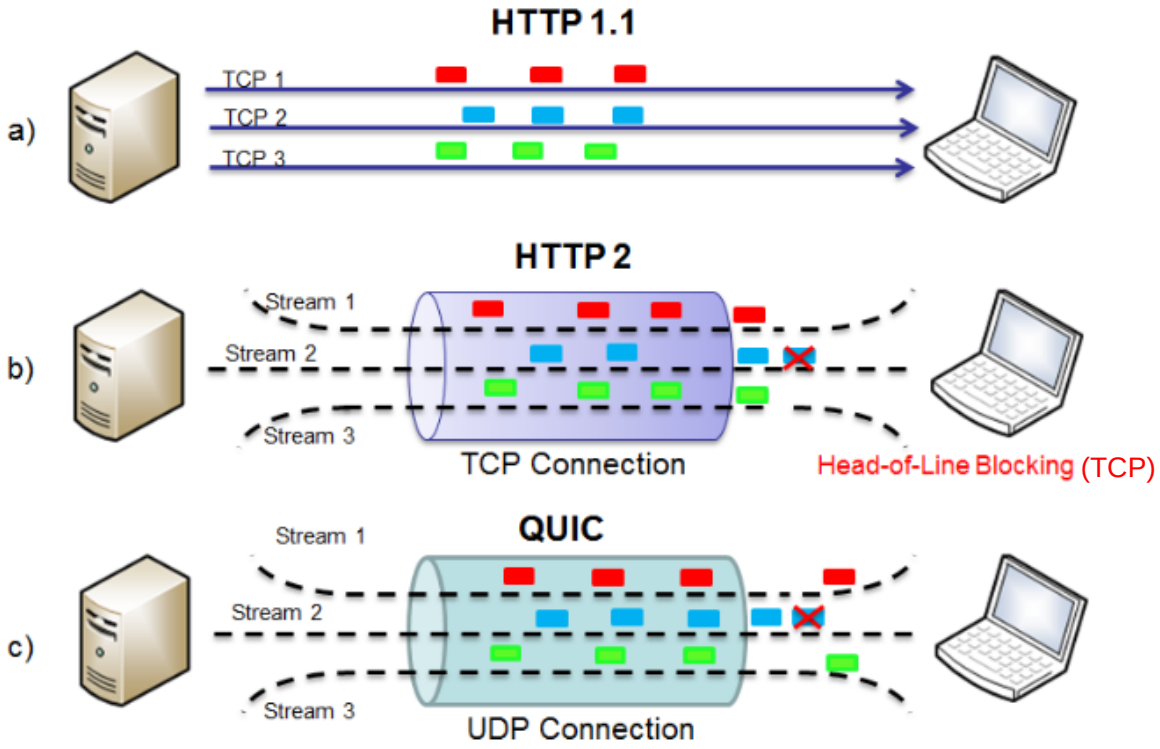
Websockets ali HTTP/2.0

- Zaenkrat **komplementarna**, WS ima manj informacij v glavi ovkirjev
- Uporaba knjižnic: **gRPC** (HTTP/2.0) **ali Socket.IO**, **Deepstream.IO** (WS)
- Full-stack ogrodja: **Meteor**
- Realno časovne NoSQL baze: **RethinkDB**

	HTTP/2	WebSocket
Headers	Compressed (HPACK)	None
Binary	Yes	Binary or Textual
Multiplexing	Yes	Yes
Prioritization	Yes	No
Compression	Yes	Yes
Direction	Client/Server + Server Push	Bidirectional
Full-duplex	Yes	Yes

QUIC

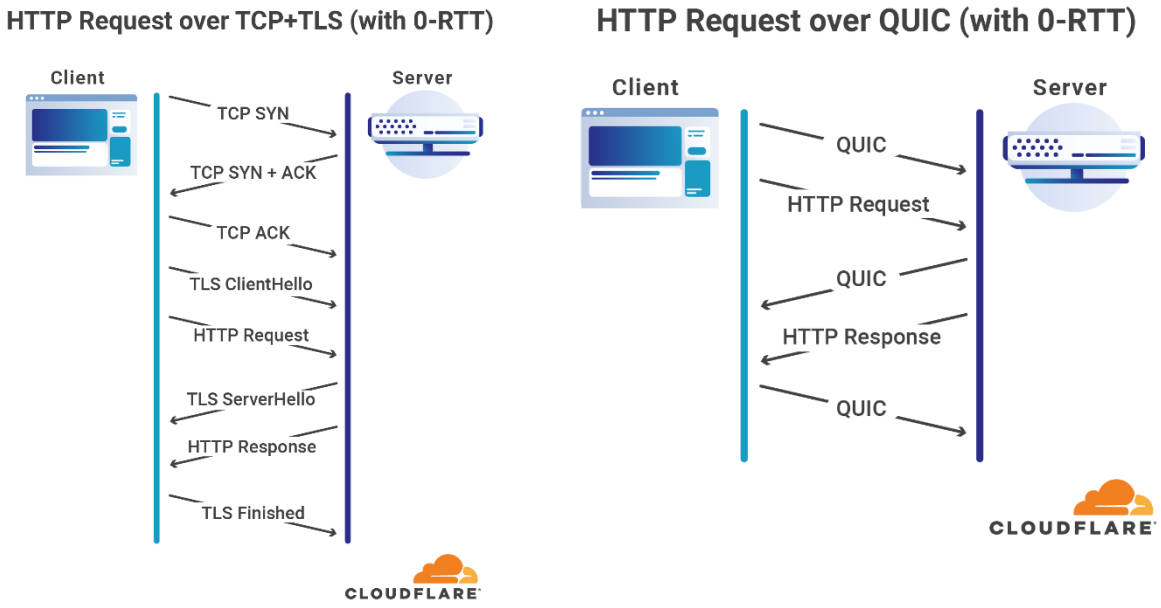
- Novi protokol od Googla (2012+), še hitrejši od HTTP/2, **integracija v HTTP/3**
- **Multipleksiranje preko UDP**, reduciranje **TCP head-of-line (HOL) blokiranja**, kjer se izgubljeni paketki morajo nadomestit

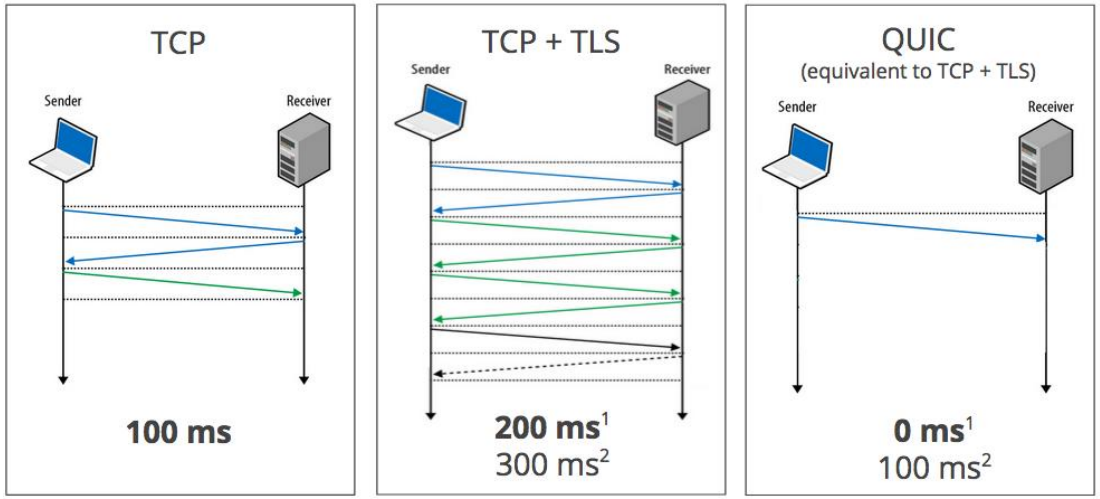


Vir slike: Cui et al., Innovating Transport with QUIC: Design Approachesand Research Challenges

QUIC

- **Reduciranje RTT (Round-trip-delay) pri TLS** z združevanjem **TCP+TLS handshake**

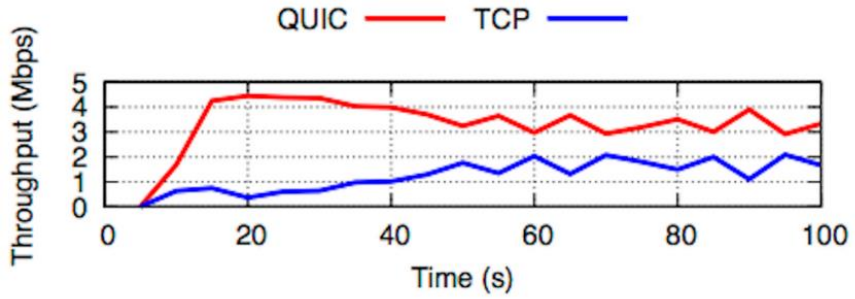




Vir slike: https://blog.cloudflare.com/even-faster-connection-establishment-with-quic-0-rtt-resumption/

QUIC

- **Vedno imamo TLS šifriranje**
- Performansa:



Scenario	Flow	Avg. throughput (std. dev.)
QUIC vs. TCP	QUIC	2.71 (0.46)
	TCP	1.62 (1.27)
QUIC vs. TCPx2	QUIC	2.8 (1.16)
	TCP 1	0.7 (0.21)
	TCP 2	0.96 (0.3)
QUIC vs. TCPx4	QUIC	2.75 (1.2)
	TCP 1	0.45 (0.14)
	TCP 2	0.36 (0.09)
	TCP 3	0.41 (0.11)
	TCP 4	0.45 (0.13)

- Aktivacija v Chromu?
  - chrome://flags/ , ctrl+f "Experimental QUIC protocol“
- Preverimo: https://www.http3check.net/

Vir slike: https://blog.apnic.net/2018/01/29/measuring-quic-vs-tcp-mobile-desktop/

Dobre prakse skalabilnosti

- Za **statične strani** je priporočljiva uporaba samodejnih strenižkov (nginx)
  - **Load balancing** na TCP nivoju
  - Predpomnenje
- **Dinamične strani**: RESTful aplikacije (npr. z node.js)
  - Load balancing na HTTP nivoju (ni primerno za poizvedbene nize – query strings)
  - Predpomnenje
- Če želimo **performanso** lahko zahtevnejše backend operacije prevedemo v strojno kodo
- Ali želimo nižjo latenco ali visoko prepustnost ?
- Ne pozabit na **varnostne vidike** in sinhronizacijo strežnikov (npr. podatkovne baze na večih strežnikih)
- Preverimo kakšen tehnološki sklad drugi uporabljajo ?
  https://builtwith.com/ https://stackshare.io/stacks