Maintaining user/server state: cookies

- جلسه قبل گفتیم HTTP یه پروتکل stateless هست که باعث سادگی سرور میشه و این باعث میشه که سرور بتونه به تعداد زیادی کانکشن TCP سرور بده و state ها لازم نیست توی کلاینت یا سرور ذخیره بشن.

  مفهومی تحت عنوان multi-step exchange نداریم ،یعنی transaction هایی که داریم یک گامی هستن و در قالب یک request و یک response پیام خودمون رو رد و بدل می کنیم.

  اگه ما پروتکلمون multi-step exchange باشه و state ها ذخیره بشن ، اگه قبل از تمام شدن transaction سمت کلاینت یا سرور مشکلی ایجاد بشه ،باید یه مکانیزمی برای ریکاوری تعبیه بشه که باعث پیچیدگی پروتکل میشه.
- مثال اسلاید ۱۸ ، یه پروتکل stateful هست که ابتدا کلاینت از سرور درخواست می کنه که یه رکوردی توی دیتابیسش lock بشه تا کلاینت دیگه ای حق دسترسی و تغییر این رکورد رو نداشته باشه. سرور تایید می کنه . بعد کلاینت مبتنی بر این state ، یه پیامی برای سرور ارسال می کنه که رکورد X به X' تغییر کنه . این X' خودش یه state میانی هست و برای همین کلاینت دوباره

درخواست تغییر وضعیت از X' به X'' می کنه و سرور هم تایید می کنه.در نهایت کلاینت درخواست unlock کردن اون resource رو میده و با تایید سرور، بقیه ی کلاینت ها هم میتونن به این رکورد آپدیت شده دسترسی داشته باشن .

- اگه پروتکل بخواد state ها رو حفظ کنه پیچیدگیش زیاد میشه . مثلا اگه توی همین مثال ، در لحظه ی t' یه تغییری ایجاد کردیم ولی حالت نهایی نیست ، و اگه کلاینت دچار crash بشه،توی بعضی از اپلیکیشن ها کارهایی که تا این مرحله انجام دادیم رو undo کرد، ولی توی بعضی از اپلیکیشن ها این کار ممکنه راحت نباشه !
- توی یه سری از اپلیکیشن ها، شناسایی و رصد کار هایی که کاربران انجام میدن مطلوب و گاها الزامیه .این اپلیکیشن ها از این اطلاعات برای ارائه ی سرویس های شخصی سازی شده و داینامیک استفاده می کنن.

برای این جور اپلیکیشن ها ، توسط پروتکل HTTP یه راهکاری تعیین شده به نام cookies . این تکنولوژی ۴ تا مؤلفه داره :

1 - Cookie header line of HTTP response message
2 - Cookie header line in next HTTP request message
3 - Cookie file kept on user’s host managed by user’s browser

4 - Back-end database at Web site

- این دوتا دیتابیس که یکی سمت browser هست و یکی سمت وب سایت ، می تونن از لحاظ اندازه خیلی متفاوت باشن . مثلا سمت browser در حد چند مگابایت داده در رابطه با یه صفحه وب ذخیره کنه ، ولی سمت وب سرور ، راجب هر کاربر، ممکنه کلی اطلاعات راجع به نحوه ی تعامل با سرویسی که اون وب سرور ارائه می کرده داشته باشیم.

یه مثال مربوط به این قضیه ، راجع به تجارت الکترونیک (e-commerce) در شرکت آمازون هست . این که وقتی یه کاربری به وب سایتشون مراجعه می کنه نحوه ی استفاده از cookie ها به چه شکله.

اگه کاربر قبلا به این سایت مراجعه نکرده باشه ، توی دیتابیسی که سمت کلاینت داریم ،رکوردی متناظر با کوکی سایت آمازون وجود نداره . پس اولین پیامی که کلاینت به وب سرور آمازون می فرسته، یه HTTP request معمولیه که هدرلاین مرتبط با کوکی در اون وجود نداره. وقتی وب سرور آمازون این پیام رو دریافت می کنه و می بینه که هدرلاین کوکی رو نداره، بلافاصله برای این کاربر یه رکوردی به همراه یه ID توی دیتابیس خودش ثبت می کنه که این ID رو هروقت داشته باشیم ما رو به رکورد خاصی توی دیتابیس که برای این کاربر ایجاد شده، می بره . بعد این ID رو در اولین HTTP

response ای که برای کلاینت می فرسته ،قرار میده.(توی قسمت set cookie) . browser هم متناظر با این set cookie در دیتابیس خودش ، یه رکوردی برای آمازون ایجاد می کنه و مقدارش برابر با مقدار set cookie میذاره. ازین به بعد هر وقت که کاربر وارد وب سایت آمازون بشه و کلاینت پیامی برای وب سرور آمازون بفرسته ، توی هدر پیامش ، قسمتی به نام cookie هست که همین مقداری که سرور به عنوان set cookie بهش داده بود، در اون قرار گرفته. به این ترتیب وب سرور آمازون می تونه کاربر رو track کنه و تاریخچه ی کاربر رو ثبت کنه.

- وب سرور آمازون از طریق این تکنولوژی ، می تونه توی recommendation system اش استفاده کنه و از تاریخچه ی صفحات یا اقلامی که کاربر ویزیت کرده ، یه سری پیشنهاد یا تخفیف یا ... به اون کاربر میده . بنابراین این سرویسی که آمازون داره به کاربرش میده یه سرویس شخصی سازی شده هست.

- **کاربرد cookie ها :**

۱-authorization(احراز اصالت) : از ID ای که سرور به کلاینت داده برای احراز اصالتش می تونه استفاده بشه.

۲-shopping carts : در بحث تجارت الکترونیک ، از کوکی ها استفاده می کنن که اقلامی که کاربر انتخاب کرده رو در توی کارتی ذخیره کنن و بعد کاربر بتونه بهش مراجعه کنه و اقدام به خرید کنه.

۳-recommendations : از تاریحچه ی کاربران استفاده می کنن تا محصولات بیشتر برای نیازهایی که دارن بهشون معرفی کنن که باعث سود آوری بیشتر برای اون وب سرویس میشه.

۴-user session state : برای تشکیل جلسه ها، مثل Web e-mail که در اون ها کارهایی که انجام میدیم مبتنی بر اینکه قبلا چه صفحه هایی رو انتخاب کردیم. در واقع کارهایی که الان می تونیم انجام بدیم تابع کارهایی هست که قبلا با اون وب سرور انجام دادیم.

- چطور state کاربر ها رو ذخیره کنیم؟

روشی که توی مثال قبل راجع به کوکی ها گفتیم امکان پذیره. ولی در یه سری اپلیکیشن هایی که نمی خوان به طور سنگین کاربر رو track کنن ، روش دیگه ای به کار می برن ، اینه که از اطلاعاتی که توی هدر لاین کوکی وجود داره ، استفاده کنن ، برای اینکه این اطلاعات و انتخاب هایی که کاربر قبلا انجام داده رو به سرور اطلاع بدن.

برای این منظور توی پروتکل HTTP ،سرور می تونه چندین set cookie برای کلاینت ارسال کنه و ضمن هرset cookie یه name و value ای که متناظر با یه ویژگی هست رو به browser اطلاع میدن و browser همه ی این ها رو توی دیتابیس خودش

ذخیره می کنه . بعد توی تعامل های بعدی ، browser می تونه با استفاده از هدرلاین کوکی ها ، به اندازه محدودی، میتونه از اطلاعات کاربر بهره مند بشه.

- کوکی مکانیزمیه که خیلی از اپلیکیشن ها برای ارائه ی سرویسشون ازش استفاده می کنن.اما این که رفتار کاربران track میشه و ممکنه در جایی که کاربران راضی نیستن استفاده بشه. حالا این اطلاعات یا ممکنه توسط خود وب سرور مورد سوء استفاده قرار بگیره(مثل پروفایلی که از کاربر ها ثبت می کنن و احیانا شماره همراه، شماره ی کارت بانکی و ... هم شاملش هست که ممکنه در جایی که کاربر راضی نیست ازش استفاده بشه.)
- همچنین خیلی از وب سرویس ها از کمپانی هایی مثل third party ها برای کارهای آمارگیری یا تبلیغاتی کمک می گیرن و ما موقع تعامل با اون صفحه ی وب با وب سرورهای third party ها هم ارتباط داریم و خود اون third party هم ممکنه با چندین وب سایت قرارداد داشته بشه و اون ها بتونن حرکت های ما رو track کنن و از کنارهم قرار دادن پروفایل حرکت ما، اطلاعاتی رو استخراج کنه که باز ما راضی به استفاده ازش نیستیم.

- **Web caches** :

- یه Web cache یا proxy server ، سروری هست که در داخل network یا ISP یه سازمان وجود داره، و وظیفه اش اینه که به جای اینکه http request های کاربران به سرور های اصلی برسه و از اونها پاسخ بگیرن، به صورت محلی به این درخواست ها پاسخ بدن.
- نحوه ی عملکردش مثل شکل اسلاید ۲۲ هست. دوتا کلاینت در این شکل وجود داره. Browser کلاینت ها به این صورت کانفیگ شده که http request هاشون رو به Web cache بفرستن. اگه توی Web cache المانی که URL اش توی این http request ها هست و زمان زیادی از نسخه ی اون نگذشته باشه(معتبر باشه)، وجود داشته باشه ،کپی همین نسخه رو به عنوان response برای اون درخواست می فرسته. اما اگه نسخه ای از URL موجود نبود، Web cache به جای کلاینت میاد اون رو از سرور درخواست می کنه و جواب رو می گیره و یه کپی ازش نگه داری می کنه تا احیانا اگر خود اون کلاینت یا کلاینت های دیگه ی اون شبکه ، در خواست URL یکسانی رو کردن ، بتونه برای اون ها بفرسته و یه نسخه کپی هم برای کلاینتی که به صورت اولیه اون URL رو درخواست کرده بود ، می فرسته.

- **هدف استفاده از Web cache** :

۱-زمان response برای request کلاینت کمتر بشه.

۲-باعث میشه ترافیک داخل access link یا core اون شبکه(یا ISP) کمتر بشه.

۳-بعضی از اپلیکیشن هایی که دارن سرویس محتوا ارائه میدن(content delivery network) و نمی تونن سرمایه گذاری زیادی برای ایجاد دیتاسنتر های بزرگ انجام بدن، Web cache ها باعث میشن شبکه های content provider ای که ضعیف هستن هم بتونن سرویس خودشون رو همچنان ارائه کنن.(Web cache در خیلی از مواقع به جای اون ها پاسخ میدن)

- Web cache ها هم نقش کلاینت رو بازی می کنن هم سرور.

جایی نقش سرور رو داره که به صورت اولیه ، درخواست کلاینت رو دریافت می کنه و اگه نسخه ای داره، اون رو برای کلاینت میفرسته.

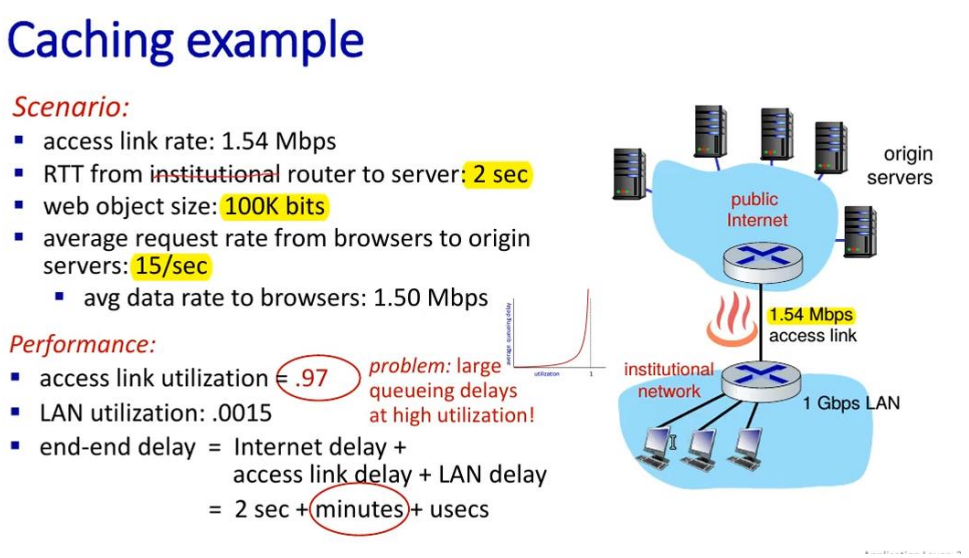
جایی نقش کلاینت رو داره که نسخه ای در داخل خودش وجود نداره یا نسخه ای هست که طبق الگوریتم های خود Web cache ، حدس میزنه که اون نسخه آپدیت نیست و مجبوره با سرور اصلی ارتباط برقرار کنه که اگه نسخه ی آپدیتی داره براش بفرسته.

- داخل پروتکل HTTP ، یه سری هدرلاین برای کارهای کنترلی مربوط به caching تعبیه شدن .

مثل cache-control : ازش استفاده میشه تا مشخص بشه یه نسخه ای از URL که توسط cache دریافت شده تا چه مدتی اعتبار داره و تا اون زمان اگه درخواست دیگه به cache رسید، از همین

نسخه استفاده کنه و اگه مدت زمان از max-age گذشته بود، بره مجددا از original server درخواست نسخه ی آپدیت شده بکنه.

- بعضی محتوا ها هستن که قابلیت cache شدن ندارن . مثلا وقتی یه HTTP request ای به یه سرور ارسال می کنیم ، یه محتوایی داخل سرور ساخته و برای ما ارسال میشه. این محتوا ربط داره به اینکه چه زمانی HTTP request به سرور می رسه ، و طبیعتا قابلیت cache شدن نداره . اون وب سرور هم میتونه در مورد این اطلاعات توسط هدرلاین cache-control اطلاع بده که قابلیت cache شدن ندارن. و اگه مجددا این URL به Web cache رسید دیگه نسخه ای از اون داخل cache نگهداری نشده و برای سرور اصلی تقاضا ارسال خواهد شد.
- توی مثال اسلاید ۲۴ می خوایم تاخیر end-to-end ای رو حساب کنیم که در واقع مدت زمان یه request و یه response هست.



وقتی یه HTTP request ارسال میشه باید از طریق لینک 1Gbps باید به دست روتر توی institutional network برسه. از این تاخیر می تونیم صرف نظر کنیم چون حجم بسته های request کمه و سعت لینک زیاده و طول لینک هم اونقدر زیاد نیست که تاخیر ارسال چشم گیر بشه.

بسته هایی که به institutional network می رسن ممکنه با تاخیر صف مواجه بشن .اما چون تو مسیر uplink حجم بسته هایی که داریم ارسال می کنیم(بسته های request) کمه و چون فرض کردیم در هر ثانیه ۱۵ بسته داره ارسال میشه ، مجددا میانگین نرخ ورودی در مقابل 1.54 Mbps خیلی کم هست و میتونیم از تاخیر صف هم صرف نظر کنیم.

بعد از اینکه HTTP request ها به روتر سمت اینترنت رسیدن ، ۲ ثانیه هم طول می کشه که جوابش رو از سرور دریافت کنن. اینجا دوباره باید بررسی کنیم که چقدر دچار تاخیر صف میشه. اینجا چون به طور میانگین ۱۵ بسته به روتر اینترنت می رسه و حجم هر بسته 100 Kbit هست ، میانگین نرخ ورودی 1.5 Mbps میشه و این ها همه از لینکی قراره استفاده کنن که طرفیتش 1.54 Mbps هست . اگه traffic intensity یا link utilization رو محاسبه کنیم برابر میشه با 1.5/1.54 = 0.97 که خیلی نزدیک یکه و طبق نمودار به

مربوط به تاخیر صف، این تاخیر خیلی زیاده(توی شبکه تاخیر زیاد رو در حد چند دقیقه می تونیم در نظر بگیریم) .

بنابراین تاخیر صفی که HTTP response ها در بافر روتر سمت اینترنت باهاش مواجه میشن ،(وقتی میخوان از access link استفاده کنن)در حد چند دقیقه هست. وقتی نوبتشون میشه و روی لینک قرار می گیرن، تاخیر ارسالشون در حد چند میلی ثانیه ست و وقتی هم به روتر institutional network می رسن تاخیر صفشون زیاد نیست ، چون اگه نرخ ورودی رو 1.5Mbps در نظر بگیریم ، لینک های خروجی ظرفیتشون 1Gbps عه و شدت ترافیک در روتر institutional network در حد 0.0015 هست که خیلی کمه و در حد میکروثانیه ست.

پس کلا تاخیر های قابل ملاحظه ای که در نظر می گیریم ، 2 ثانیه تاخیر مربوط به دریافت پاسخ از سرور ها به روتر های سمت اینترنت و تاخیر صف توی روتر اینترنت هست. این تاخیر end-to-end در حد چند دقیقه هست و برای کاربرای اینترنت اصلا قابل قبول نیست!

- راه اول برای حل این مشکل ، اینه که اون موسسه بیاد access link سریع تری استفاده کنه. مثلا سرعتش صد برابر (154 Mbps) بشه. با این تغییر شدت ترافیک به جای 0.97 ،میشه 0.0097 و تاخیر صف چیزی در حدود چند میکروثانیه میشه.البته استفاده از

این لینک با سرعت بالا، هزینه ی خیلی زیادی داره و موسسه رو با مشکل مواجه می کنه!

- راه حل دوم ، این هست که از همون access link اولیه استفاده کنیم، ولی در کنارش از یه web cache در داخل شبکه ی موسسه استفاده کنیم که از طریق لینک 1 Gbps میتونیم بهش دسترسی داشته باشیم. در این صورت ابتدا HTTP request ها میان سمت web cache ، اگه نسخه ای از request داخل web cache وجود داشت ، همون سرور web cache ، HTTP response رو برای ما ارسال می کنه .

ولی باید ببینیم که چه تعداد ازین درخواست ها hit و چه تعداد miss میشن. منظور از hit اینه که ما نسخه ای ازون URL درخواست شده رو داخل web cache داریم تا ارسال کنیم.

منظور از miss هم اینه که اون نسخه یا قدیمیه یا اینکه اصن وجود نداره ، و سرور های اصلی باید جواب رو ارسال کنن.

hit rate عددی بین 0.2 تا 0.7 هست ولی توی مثال اسلاید ۲۷ ، اون رو در حدود 0.4 (مقدار میانی) در نظر می گیریم. بنابراین 40% درخواست ها در اون شبکه پاسخ داده میشه . چون شبکه محلی اون موسسه پر سرعت (1 Gbps) هست، میتونیم بگیم مواقعی که hit رخ میده، تاخیرمون در حد میلی ثانیه هست.

60% مواقع هم درخواست ها به سرورهای اصلی فرستاده میشن و در این صورت میانگین نرخ بیت ورودی در access link هم 0.6 برابر میشه (برابر با 0.9 Mbps) و مقدار access link utilization هم برابر میشه با : 0.58 = 1.54 / 0.9 . این مقدار توی منحنی مربوط به تاخیر صف چون هنوز به زانوی منحنی :/ نزدیک نشده ، میتونیم بگیم تاخیر صف در حد میلی ثانیه هست.

- پس راجع به میانگین تاخیر end-to-end باید ۲ حالت رو در نظر بگیریم: یا درخواست hit میشه یا miss. اگه miss باید تاخیر رو از origin server حساب کنیم که توی این مثال تقریبا برابر با ۲ ثانیه میشه.

اگر هم hit شد ، تاخیر در حدود میلی ثانیه میشه.

بنابراین کل تاخیر جمع این دو مقدار و حدودا برابر با 1.2 msecs میشه.

این راه حل ، نه تنها صرفه ی اقتصادی داره(چون استفاده از web cache به مراتب ارزون تر از استفاده از access link سریع تره.) بلکه میزان end-to-end delay هم نسبت به راه حل اول بهتر شد.

- Conditional GET

- در پروتکل HTTP ما هدرلاین هایی داریم تحت عنوان If-modified-since و ازون طرف توی http response ها status line هایی داریم که کدشون ۳۰۴ هست به معنای Not Modified .
- وقتی یه نسخه از آبجکتی رو که می خوایم داریم(مثلا در cache browser کلاینت ذخیره شده یا اینکه خودش web cache هست) ، اگه شک کنیم که اون آبجکتی که کاربر درخواست کرده نسخه ی به روزش هست یا نه ، می تونیم یه HTTP request به سرور بفرستیم و از هدرلاین if-modified-since استفاده کنیم و یه تاریخی رو بذاریم توی اون هدرلاین . این تاریخ همون تاریخی هست که اصولا دفعه ی اولی که سرور این آبجکت رو برای کلاینت فرستاده توی هدرلاین last-modified ثبت کرده.

به همچین HTTP request ، Conditional GET میگن . وقتی این درخواست به دست سرور می رسه ، سرور میاد تاریخی که توی هدر این درخواست هست رو با تاریخ last-modified مقایسه می کنه ، اگه نسبت به این تاریخ به روز رسانی جدیدی شکل نگرفته ، سرور یه HTTP response با کد 304 Not Modified می

فرسته و بدنه ی این پیام هم خالیه(دیگه مجددا آبجکت رو برای کلاینت نمی فرسته).

اما اگه تاریخ به روز رسانی بعد از تاریخی باشه که توی هدرلاین HTTP requests ثبت شده(متأخر باشه) سرور در پاسخ، یه HTTP response معمولی با کد 200 OK ارسال می کنه که توی بدنه ی این پیام آبجکت آپدیت شده رو قرار داده و پیام خالی نیست.

- **HTTP/2**

- HTTP/2 در سال ۲۰۱۵ در قالب RFC 7540 معرفی شد که نسبت به HTTP/1.1 سرعت بیشتری داره .
- Method ها ، status code ها ، و هدرلاین ها نسبت به HTTP/1.1 تغییری نکردن و بیشتر تمرکز روی افزایش سرعت انتقال آبجکت های یه web page بوده.
- یه روش این هست که سرور برای فرستادن آبجکت های مختلف یه web page به صورت FCFS(First Come First Serve) لزوما عمل نمی کنه.

- به صورت اولیه توی HTTP/1.1 ، هر درخواستی زودتر سمت سرور دریافت بشه ، اول آبجکت متناظر با اون درخواست ارسال میشه.
ولی توی HTTP/2 لزوما این طور نیست ، و سرور میتونه توی فرستادن آبجکت های یه صفحه ی وب ، اولویت های دیگه ای رو مدنظر قرار بده.
- بعضی از آبجکت ها رو سرور میتونه با مکانیزم push برای کلاینت ارسال کنه حتی قبل از اینکه request اش به سرور ارسال شده باشه.

توی HTTP/1.1 کلاینت اول base html رو درخواست می کرد و متناظر با پاسخی که می گرفت ، می فهمید باید چه آبجکت های دیگه ای که وجود دارن رو هم درخواست کنه.
ولی توی HTTP/2 ، سرور به جای اینکه منتظر بمونه ، تا تقاضای دریافت سایر آبجکت ها رو از کلاینت بیاد، خودش بلافاصله بعد از ارسال base html ، بقیه ی آبجکت ها رو هم ارسال می کنه. به این مکانیزم push میگن که باعث کاهش تاخیر هم میشه.

- توی HTTP/1.1 ارسال آبجکت ها به صورت complete انجام میشه یعنی یه آبجکت ارسال میشه و بعد آبجکت بعدی ارسالش شروع میشه، حالا اگه یه آبجکتی اندازش زیاد باشه آبجکت های کوچکتر که اولویت کمتری دارن متحمل تاخیر زیادی میشن تا اینکه نوبت ارسالشون بشه. به این وضعیت HOL(head Of the Line)

blocking هم میگن . یعنی اول خط بلاک شده توسط آبجکت بزرگتر.

ولی توی HTTP/2 ،برای برطرف کردن این مشکل میان آبجکت ها رو به frame ها می شکنن و به صورت یکی در میان فریم های آبجکت های مختلف رو می فرستن. بنابراین اگه یه سری از آبجکت ها کوچک باشن ، خیلی سریع تر دریافت میشن و معطل آبجکت بزرگتر نمیشن. (مثال های اسلاید ۳۱و ۳۲)

هرچند آبجکت بزرگتر دچار تاخیر میشه تا فریم هاش به طور کامل ارسال بشن ولی آبجکت های کوچکتر زمان دریافتشون کمتر میشه و کاربر حس بهتری نسبت به سرعت عمل شبکه بهش دست میده!