CAP THEOREM:
وقتی سیستم‌های توزیع‌شده باید انتخاب کنند

رضا قمی‌الاصل
دانشجو مهندسی کامپیوتر
دانشکدگان فارابی دانشگاه تهران
rezaqomyalasl@gmail.com

۱۱ دقیقه

فرض کنید دارید یه اپلیکیشن می‌سازید که قراره میلیون‌ها نفر ازش استفاده کنن. از یه جایی به بعد دیگه یه سرور نمی‌تونه جواب‌گو باشه. پس مجبور می‌شید داده‌هاتون رو بین چند سرور پخش کنید. این کار هم عملکرد رو بهتر می‌کنه، هم اگه یکی از سرورها از کار افتاد، بقیه سرورها هستن و نرم افزارتون می‌تونه به کار خودش ادامه بده.

اما اینجا یه سری سوال مهم پیش میاد:

چطور مطمئن بشیم همه‌ی سرورها داده‌ی یکسانی دارن؟

اگه ارتباط بین‌شون یا مثلاً اینترنت قطع بشه، سرور ها باید چی‌کار کنن؟

چطور همیشه و سریع به کاربر ها جواب بدیم؟

جواب کوتاه و ناراحت‌کننده اینه که نمی‌تونیم همه‌ی این‌ها رو با هم داشته باشیم!

اینجاست که پای قضیه CAP به میون میاد. مفهومی که می‌گه در سیستم‌های توزیع‌شده، وقتی مشکل شبکه پیش میاد، همیشه باید بین چند هدف مهم، یکی رو فدای دیگری کنیم.

CAP THEOREM یا قضیه CAP چیه دقیقاً؟

این ایده اولین بار سال ۲۰۰۰، توسط اریک بروئر از دانشگاه برکلی مطرح شد. بعدها به اسم CAP Theorem شناخته شد.

طبق این قضیه، هر سیستم توزیع‌شده در زمان پارتیشن شبکه (اختلال شبکه؛ وقتی ارتباط بین بخشی از سرورها قطع یا مختل می‌شه) فقط می‌تونه دو تا از سه ویژگی زیر رو به‌طور همزمان داشته باشه:

۱. Consistency (سازگاری)[1]

یعنی همه‌ی نودها در یک لحظه، داده‌ی یکسانی رو نمایش بدن. اگه یه کاربر داده‌ای رو تغییر داد، همه باید همون تغییر رو بلافاصله ببینن.

مثال: توی سیستم بانکی، وقتی ۵ میلیون از حسابتون برداشت می‌کنید، موجودی در همه‌ی شعب و اپلیکیشن‌ها فوراً به‌روز می‌شه.

۲. Availability (در دسترس بودن)

یعنی سیستم همیشه جواب بده - حتی اگه بخشی از اون خراب شده باشه. هر درخواست معتبر باید جوابی دریافت کنه، حتی اگه جدیدترین داده نباشه.

مثال: توی توییتر، حتی اگه یکی از دیتاسنترها از کار بیفته، همچنان می‌تونید توییت‌ها رو ببینید(شاید کمی قدیمی).

۳. Partition Tolerance (تحمل پارتیشن)

یعنی سیستم باید بتونه در شرایطی که ارتباط شبکه بین بعضی از سرورها قطع یا مختل شده (پارتیشن شبکه)، به کار خودش ادامه بده.

این ویژگی در دنیای واقعی اجتناب‌ناپذیره. همیشه این امکان وجود داره که کابل شبکه قطع بشه، سوییچ خراب بشه، یا تأخیر شبکه بالا بره. نمی‌تونید فرض کنید که اختلال هرگز اتفاق نمی‌افته.

چرا نمی‌تونیم هر سه رو با هم داشته باشیم؟

فرض کنید یه فروشگاه اینترنتی دارید با دو تا دیتاسنتر - یکی در تهران و یکی در مشهد. موجودی یه کالا روی هر دو سرور «۱۰ عدد» ثبت شده.

حالا یه مشتری در تهران خرید می‌کنه. سرور تهران موجودی رو به «۹» تغییر می‌ده. بعد سعی می‌کنه این تغییر رو به سرور مشهد بفرسته.

تا اینجا مشکلی نیست. اما اگه ارتباط بین تهران و مشهد قطع بشه چی؟

سرور مشهد از تغییر خبر نداره. هنوز فکر می‌کنه موجودی ۱۰ تاست. حالا یه مشتری از مشهد هم همون کالا رو می‌خره.

اینجاست که پارتیشن شبکه رخ داده. سیستم باید یکی از دو انتخاب سخت رو انجام بده:

انتخاب CP (سازگاری + تحمل پارتیشن)

بگید «تا وقتی مطمئن نیستم، جواب نمی‌دم».

داده درست می‌مونه، ولی سیستم موقتاً از دسترس خارجه. یعنی سرور مشهد درخواست رو رد می‌کنه تا وقتی که بتونه با تهران هماهنگ بشه.

انتخاب AP (در دسترس بودن + تحمل پارتیشن)

بگید «فعلاً بفروش، بعداً هماهنگ می‌کنیم».

سیستم فعال می‌مونه، ولی داده‌ها ممکنه موقتاً ناسازگار بشن. یعنی ممکنه هر دو سرور کالا رو بفروشن و موجودی منفی بشه.

نکته مهم: در عمل، سیستم‌هایی که AP رو انتخاب می‌کنن معمولاً از روش‌هایی مثل صف‌های پیام‌رسانی، قفل‌های توزیع‌شده، یا مکانیزم‌های هماهنگی تدریجی (eventual consistency) استفاده می‌کنن تا بعد از برقراری مجدد ارتباط، تعارض‌ها رو حل کنن و از فروش بیش از حد جلوگیری کنن.

۱. این Consistency با C در ACID متفاوته. در CAP منظور «خواندن آخرین داده نوشته‌شده» هست (linearizability). اما در ACID منظور «رعایت قوانین یکپارچگی داده» مثل کلیدهای خارجی و محدودیت‌هاست.

نکته کلیدی: در شرایط عادی (بدون پارتیشن)، سیستم می‌تونه هر سه ویژگی رو داشته باشه. اما وقتی اختلال شبکه پیش میاد، مجبوره بین Consistency و Availability یکی رو انتخاب کنه.

دو رویکرد اصلی: ACID و BASE

برای کنار اومدن با محدودیت CAP، دو فلسفه‌ی اصلی در طراحی پایگاه‌داده‌ها شکل گرفته:

ACID: دقت و نظم در اولویت

پایگاه‌داده‌های سنتی مثل MySQL یا PostgreSQL از مدل ACID استفاده می‌کنن. این مدل چهار ویژگی داره:

Atomicity (اتمی بودن): هر تراکنش یا کامل انجام می‌شه یا اصلاً انجام نمی‌شه. برای مثال در انتقال وجه، پول از حساب یک نفر کم و به حساب دیگری اضافه می‌شه. اگر مشکلی پیش بیاد، تراکنش به‌طور کامل لغو می‌شه و این‌طور نیست که پول از حساب مبدأ کم بشه اما به حساب مقصد واریز نشه.

Consistency (سازگاری منطقی): داده‌ها همیشه قوانین مشخص رو رعایت می‌کنن (مثل کلیدهای خارجی: مثلاً اگه توی دیتابیس، یه فیلد user_id داری که به تیبل users اشاره می‌کنه، حتماً باید user با همون id وجود داشته باشه).

Isolation (جداسازی): تراکنش‌های هم‌زمان روی هم اثر نمی‌ذارن. مثلاً اگر دو نفر به‌طور هم‌زمان بخوان موجودی حساب رو تغییر بدن، سیستم طوری عمل می‌کنه که انگار هر کدوم جداگانه اجرا شدن و هیچ تداخلی ایجاد نمی‌شه.

Durability (پایداری): داده‌ها بعد از ثبت شدن، دیگه از بین نمی‌رن حتی اگه برق دیتاسنتر به‌طور ناگهانی بره، باز هم داده‌ها به شکل صحیح و سالم سرجاشون هستن.

این نوع سیستم‌ها معمولاً سمت CP هستن. یعنی سازگاری براشون مهم‌تر از در دسترس بودن آنی هست. اگه شبکه اختلال داشته باشه، ترجیح می‌دن موقتاً جواب ندن تا داده اشتباه بدن.

BASE: انعطاف و مقیاس بالا

پایگاه‌داده‌های مدرن و توزیع‌شده مثل DynamoDB، Cassandra یا Couchbase معمولاً از مدل BASE استفاده می‌کنن. این مدل سه ویژگی اصلی داره:

Basically Available (تقریباً در دسترس): سیستم تقریباً همیشه پاسخ می‌ده.

Soft State (وضعیت نرم): وضعیت سیستم ممکنه موقتاً تغییر کنه. مثلا در یه وب‌سایت فروش آنلاین، وقتی یک کالا به تازگی موجود شده یا موجودی اون تغییر کرده، ممکنه تا مدتی در همه سرورها و کش‌ها یکسان نشون داده نشه و سیستم یه حالت موقت ناپایدار داشته باشه تا همه سرورها به‌روز بشن.

Eventually Consistent (سازگاری نهایی): اگه نوشتن جدیدی انجام نشه، در نهایت همه‌جا داده‌ها یکی می‌شن.

این نوع سیستم‌ها معمولاً سمت AP هستن. یعنی حتی در زمان قطعی یا تأخیر، سیستم همچنان پاسخ می‌ده. بعداً داده‌ها رو بین نودها هماهنگ می‌کنه. مثلاً در DynamoDB، داده ممکنه بلافاصله روی همه‌ی سرورها همگام نباشه. اما سیستم تضمین می‌کنه که در نهایت همه به حالت یکسان می‌رسن.

انتخاب در دنیای واقعی: CP یا AP؟

وقتی داده‌ها برامون خیلی مهم و حساسن، سیستم‌ها طوری طراحی می‌شن که دیتاشون همیشه درست باشن. مثل بانک‌ها، رزرو بلیط یا مدیریت موجودی کالا. تو این حالت، اگه شبکه اختلال داشته باشه، سیستم ترجیح می‌ده نوشتن رو متوقف کنه تا هیچ اطلاعات غلطی ثبت نشه.

ولی گاهی مهمه که سیستم همیشه در دسترس باشه، حتی اگه اطلاعات کمی ناهماهنگ باشن. مثل شبکه‌های اجتماعی یا وضعیت آنلاین کاربران. تو این حالت ممکنه وقتی کسی آنلاین می‌شه، چند لحظه طول بکشه تا همه جا درست به‌روز بشه و بعضی جاها هنوز «آفلاین» نشون داده بشه. بعد از مدتی همه چیز درست می‌شه و داده‌ها هماهنگ می‌شن.

اما خبر خوب اینه که لازم نیست برای کل سیستم یه انتخاب ثابت داشته باشیم. می‌تونیم ترکیبی طراحی کنیم؛ یعنی برای بخش‌های حیاتی (مثل تراکنش‌ها) سازگاری رو ترجیح بدیم و برای بخش‌های کم‌اهمیت‌تر (مثل آمار، اعلان‌ها یا وضعیت آنلاین) در دسترس بودن رو اولویت بذاریم.

نتیجه نهایی

CAP محدودیت نیست، بلکه یه راهنماست. وقتی بفهمیم چه چیزی برای سیستم و کاربرانمون واقعاً مهمه، می‌تونیم سیستمی طراحی کنیم که هم مقیاس‌پذیر و قابل اعتماد باشه، هم هوشمندانه تصمیم بگیره که کِی و چی رو فدا کنه. در واقع، مهندسی نرم‌افزار یعنی همین: انتخاب‌های سنجیده و هوشمندانه، به‌جای تلاش برای کامل بودن در همه چیز به‌طور همزمان.

منابع

[1] “Design in data intensive application” and “A Critique CAP Theorem” by Martin Chapman

[2] Werner Vogels. Eventually consistent. ACM Queue, 6(6):14-19. October 2022. doi: 10.1145/1466443.1466448.

[3] “Architecture overview” in aerospike docs