
جلسه ۲ - tanenbaum کتاب cover

Distributed system:

1. History:

سیستم های توزیع شده قائل بر سابقه و تاریخچه هستند / خیلی از فعالیت ها منبع از وجود پیشینه هست / همه چی store و retrieve می شوند.

2. Geography:

سیستم های توزیع شده گسترده جغرافیایی وسیعی دارند / به منظور ارسال یک request به یک موجودیت در یک فضای توزیع شده از naming برای هویت بخشیدن به موجودیت و address دادن درخواست استفاده می کنند.

3. Atomic multicast:

نوعی ارتباط برای به اطلاع رساندن هست که تنها گروه های خاصی قادر به دریافت اون هستند.

4. Broadcast communication:

تا جایی که می شود گسترده دریافت سیگنال را می خواهد زیاد کنند تا به دست افراد بیشتری برسه.

5. Point to point communication:

ارتباط P2P که دو موجودیت با هم ارتباط گرفتند و قرار نیست توسط دیگران شنود بشوند.

6. Wide-area communication:

به طور کلی عوامل مختلف در یک سیستم توزیعی نیاز به برقراری ارتباط با هم دارند / سیستم توزیعی باید برای هر سامانه ای قابلیت communication را در نظر بگیرد مثل UFO.

7. Clock synchronization:

در سیستم توزیعی زمان واحد بین سیستم ها وجود نداره / clock سیستم ها با هم synch نیستند / در سیستم توزیعی timing و ترتیب زمانی اتفاقات مورد نیازه و این یک چالش است.

8. Mobile agent:

عامل کسی که به جای موجودیت کاری رو انجام میده / عامل متحرک عاملیه که جا به جایی هم داره و ثابت نیست و کیفیت کارش باید در عین تحرک حفظ شه.

9. Passive communication:

مثل یک برچسب و اطلاعیه ست که در مکان ثابتی قرار داره و اطلاعات رو به این صورت منتقل میکنه.

10. Replication:

نیازه که از یک عنصر گاهی بیش از یک نسخه وجود داشته باشه / عضو تکثیر شده از یک global space به یک local space او مده و اگر نسخه اصلی تغییری که باید به تبع اون تغییر کنه و در سیستم توزیع شده چالش coherence رو ایجاد میکنه.

11. Transaction / client / server:

یک client از طریق یک میان افزار به server هزینه پرداخت میکنه تا یک خدمتی از server دریافت کنه و یک تراکنش رخ بده / ممکنه client و server در یک locality نباشند / history دارای history هست و فهرست خدماتی که در فرکانس قبلی خرید توسط client هست رو در میاره و بر اساس تحلیل رفتاری مشتری که از history به دست آورده سفارش انبارش رو برای ارائه خدمات آینده میده.

12. Fault tolerant:

در سیستم های توزیع شده سامانه هاش توسط developer باید تحمل پذیری خطا توشون در نظر گرفته شده باشه تا هر سامانه بتوانه ماموریت خودشو انجام بده.

13. Legacy system:

سامانه موروثی یک موجودیت نامنوس در سیستمه (عجق و جقه) / در سیستم های توزیعی نمیشه این موجودیت ها رو کنار گذاشت و باید باهشون کنار اومد / در سیستم توزیع شده با توجه به سابقه قبلی و چیز هایی که در آینده رخ میده، باید وضعیت جاری رو manage کرد.

14. stream:

دنیای بیرون analog و پیوسته هست اما دنیای کامپیوتر digital و گسسته و ۰ و ۱ هست. در دنیای واقعی داده ها دارن به صورت stream و پیوسته تولید میشن و بحث زمان هم درشون مطرحه و باید در نظر گرفته بشه.

معرفی (introduction)

بیشتر توضیحات این فصل توضیحات عام هستن که با تکید حل تمرین در سوالات و تمارین نمیان.

سوال: سیستم‌ها برای ذخیره‌سازی و پردازش داده‌ها چطور مقیاس‌پذیر میشون؟

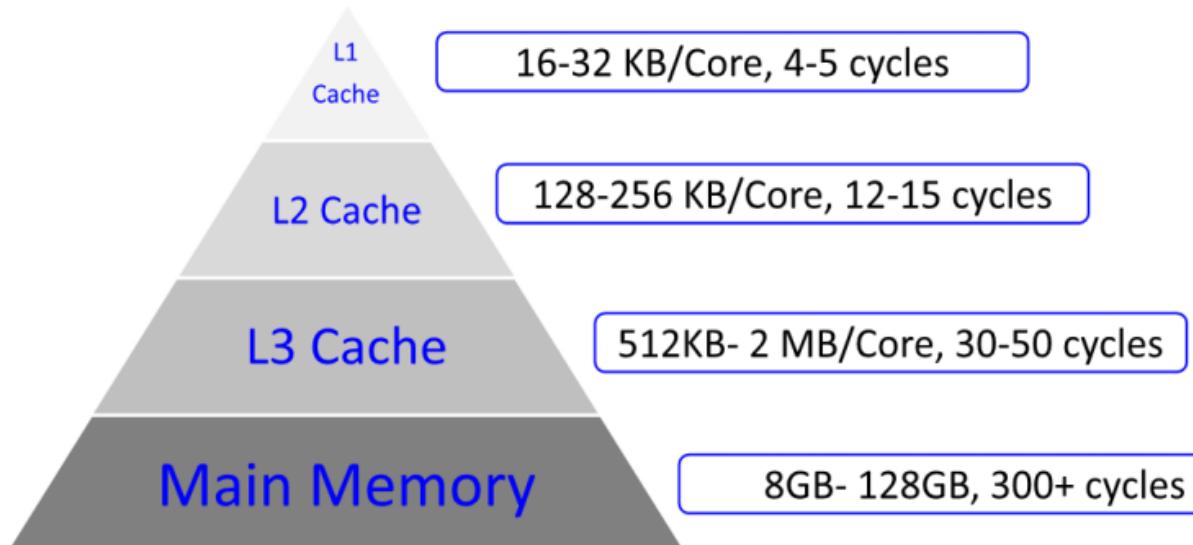
مقیاس‌پذیری در دو سطح رخ میده:

- ۱) عمودی: در این حالت سخت‌افزار یک سیستم کامپیوتری ارتقا پیدا میکن (مثلا cpu سریع‌تر یا حافظه بیشتر میشه و ...) اینجا با توجه به معماری و توان‌های پردازشی محدودیت داریم.
- ۲) افقی: در این روش اضافه شدن ماشین‌های مختلف می‌تواند مقیاس‌پذیر بشه. اینجا هم تا حدودی محدودیت داریم.

مشکلات افزایش مقیاس عمودی (vertically)

۱) محدودیت **cache** و سلسله مراتب حافظه

تو سلسله مراتب حافظه ما کش‌های L1 و L2 و L3 داریم که در صورت نبود داده در هر کدام به سراغ بعدی میریم و بعد از اینا میریم سراغ memory و اگر اونم نبود میریم سراغ disk تقبل و کند تا به داده دسترسی داشته باشیم.

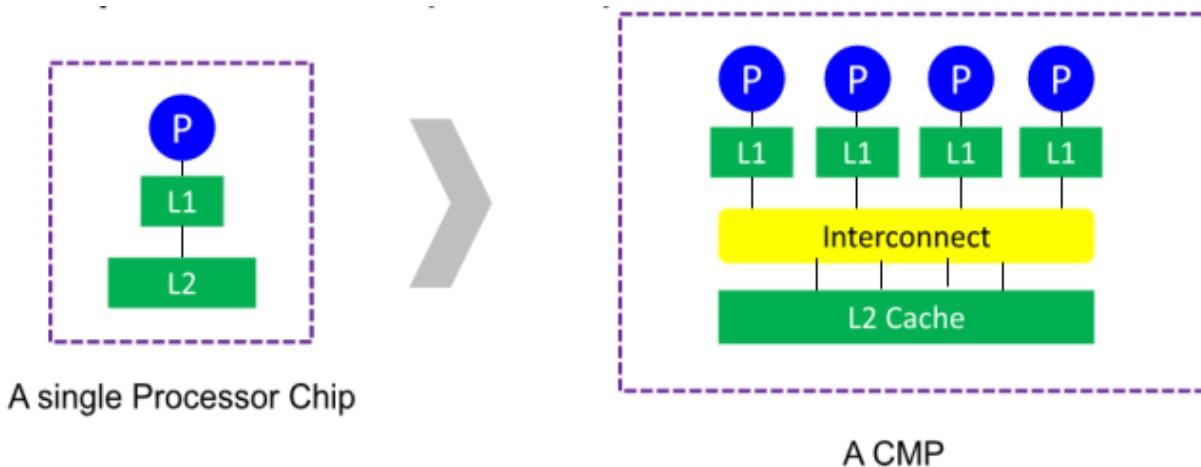


۲) محدودیت و کند بودن دیسک‌ها

- محدودیت ظرفیت
- محدودیت تعداد کانال‌ها
- محدودیت پهنه‌ای باند

۳) پردازنده (processor)

- محدودیتی به نام قانون moore داریم که اجزا نمیده سیستم‌ها را کیلویی زیاد کنیم.
که البته تلاش شده از طریق چیپ‌های چند پردازنده‌ای یکم این مشکل رو از پیش رو برداریم.

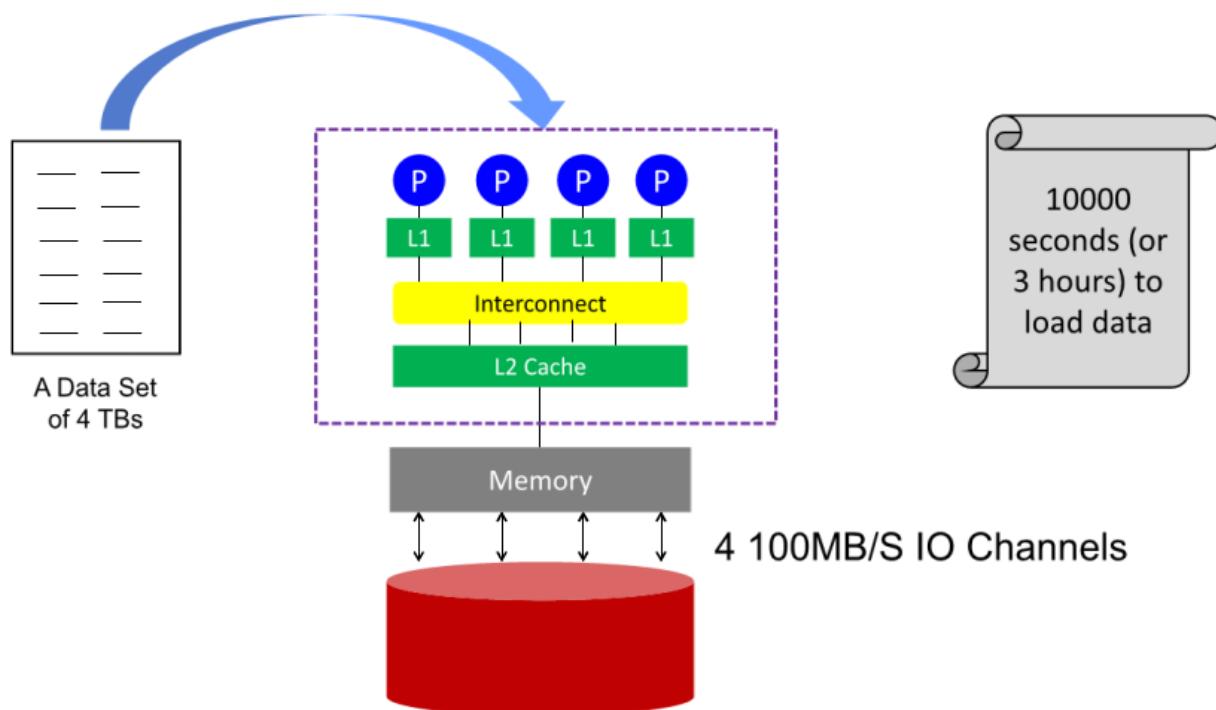


سوال: میتونم این ۴ تا پردازنده رو بکنم ۸ تا؟

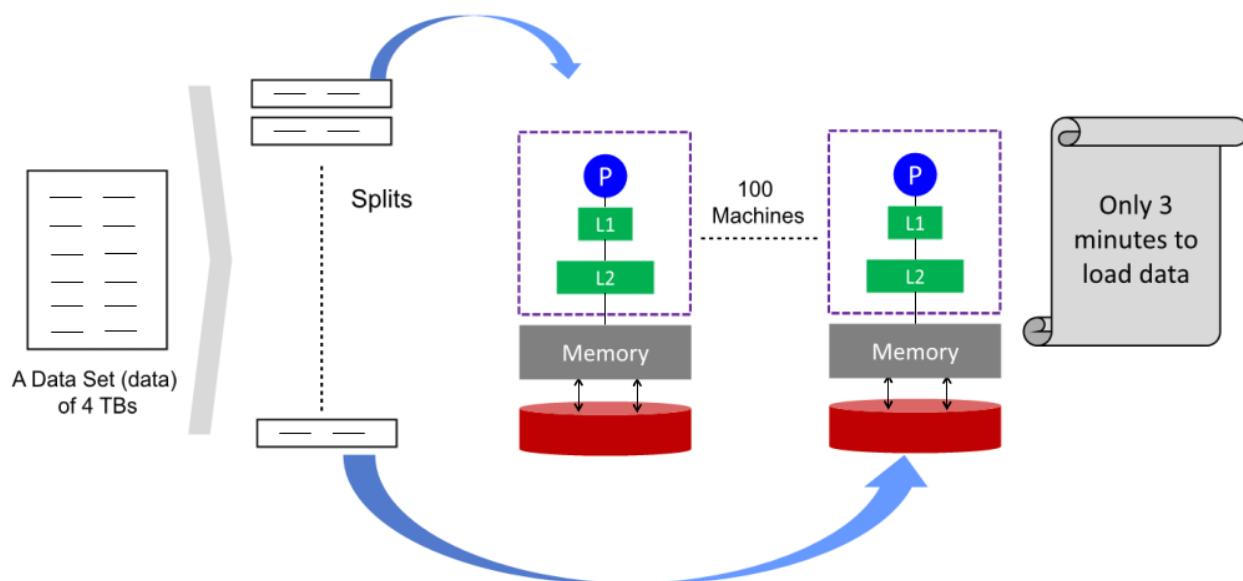
ممکنه بشه ولی باید معماری سختافزاری رو برای این مدل مقیاس‌پذیری‌های عمودی تغییر بدم.

- محدودیت سرعت تو پردازنده‌ها هم هست. برای مثال الان سرعت CPU‌ها ۵۵ درصد زیاد شده ولی سرعت حافظه فقط ۷ درصد بهبود پیدا کرده... اینجاست که میگیم تو یه سیستم هر چقدر هم که همه سریع باشن اگر یه قسمت گند باشه میشه bottleneck و همه به پای اون میسوزن.

با توجه به شکل زیر اگر بخوایم ۴ تراپایت داده رو داخل سیستم سمت راست لود بکنیم، زمانی بالغ بر ۱۰۰۰۰ ثانیه یا ۳ ساعت نیاز داریم که تو این دوره زمونه خیلی افتضاحه.



حالا با همهٔ این تفاسیر به جای اینکه بیایم عمودی مقیاس بدیم، مشابه زیر افقی این کار رو انجام بدیم. مثل سناریوی زیر که همون سناریو بالا رو او مده بین ۱۰۰ تا کامپیوتر تقسیم کرده:



همونطور که میبینیم همون کار قبلی داره تو سه دقیقه انجام میشه.

نیازمندی‌های این سیستم‌های **horizontally** یا افقی چیه؟

۱) یه مدلی برای پردازش موازی یا همروند بین ماشین‌های مختلف تعریف کنیم (Programming and Concurrency models)

۲) روشی برای دسته‌بندی پردازه‌ها (architecture)

۳) پردازه‌ها نیاز داشته باشن که با هم ارتباط برقرار کنن (communication)

۴) راهی برای مکان‌یابی و به اشتراک‌گذاری منابع (Naming Protocols))

۵) این داده‌ها زمانی معنا پیدا میکنن که پردازه‌ها بتونن اطلاعاتشون رو با هم سینک کنن و همکاری داشته باشند (Synchronization))

۶) اگر سیستم من تو علموص داره با یه سیستم توی توکیو ارتباط برقرار میکنه بالاخره latency هم این وسط دخیله. پس برای دفعات بعدی ارتباط با همون سیستم برای گرفتن همون اطلاعات میتونم از caching استفاده کنم. برای اینکه اتکاپنیری سیستم رو بالا ببریم میایم از replication استفاده میکنیم. در واقع میگیم از یه سیستم‌مون چند تا clone دیگه هم درست میکنیم ولی خب بازم بین consistency یا سازگاری بین سیستم‌هامون باید نظر بدیم که به چه صورت این داده‌ها کش بشن یعنی آیا نیاز هست داده‌ها روی همه‌ی سیستم‌ها به یک شکل و سطح باشه یا جای تغییر و جابه‌جایی وجود داره؟ (Caching, Replication and Consistency))

۷) نیاز دارم تا بار کاری خودم رو که بین همه سیستم‌هایی که افقی مقیاس‌پذیری کردم یکسان داشته باشم و اینطوری نباشه که یکی over utilize شده و اون یکی خالیه و کاری نداره. حالا این نکته هم هست که تو سیستم‌های مقیاس‌پذیر افقی همه سیستم‌ها ناهمگن. یعنی یکی گوشیه منه یکی سرور مایکروسافته و یکی سوپر کامپیوتره چرا باید لودی که رو گوشیه منه رو اون سوپر کامپیوتر هم باشه... صحبت اینه که لود روی همه سیستم‌های portable و منعطف باشه و مثل کش باشه. (Virtualization))

همه اینا رو گفتیم که برسمیم به تعریف سیستم‌های توزیع شده

سیستم‌های توزیع شده:

مجموعه‌ای از کامپیوترهای مستقل از هم که از دید کاربر به عنوان یک سیستم منسجم به نظر می‌رسد.

ویژگی اول  مجموعه‌ای از کامپیوترهای مستقل از هم

- ◀ منظور از کامپیوترهای مستقل از هم = **node** / یک پردازه = **autonomous computing element**
- ◀ نرم افزاری / یک دستگاه سخت افزاری، که نیاز به تعامل با هم دارند.
- ◀ برنامه ریزی میشن تا به هدف مشترکی دست پیدا کنند
- ◀ از طریق **message passing** با هم ارتباط می‌گیرند
- ◀ **Global clock synchronization** بین همه اجزا وجود نداره و چالش وجود دارد
- ◀ چون مجموعه‌ای از **node**‌ها هستند، چالش مدیریت عضویت (managing Group membership) و سازماندهی مجموعه هم موجود
- ◆ **Open Group** = هر **node** ای قابلیت اضافه شدن داره / میتوانه به هر **node** دیگه ای در سیستم پیام بفرسته
- ◆ **Close Group** = مکانیسمی برای اضافه و حذف شدن به گروه وجود داره / هر **node** فقط یه اعضاي گروهش میتوانه پیام ارسال کنه
- ◀ نیاز به طراحی مکانیسمی، برای احراز هویت **node**‌ها موجوده که اگر خوب طراحی نشه با **scalability bottleneck** مواجهیم
- ◀ سیستم توزیع شده به عنوان شبکه پوشانی (overlay network) در نظر گرفته میشه که همیشه باید در حالت متصل باشه / یه نوع شناخته شده این اتصال، شبکه P2P هست
- ◆ **Structured overlay** = هر گره کاملا مجموعه همسایگانشو میشناسه و با اونها در ارتباطه. مثل ساختار درخت یا logical ring
- ◆ **Unstructured overlay** = هر گره با گره هایی که تصادفی انتخاب شدن ارتباط میگیره ویژگی دوم از دید کاربر یک سیستم منسجم به نظر می رسه
- ◀ کاربر نهایی نباید متوجه این شود که پردازه ها، داده ها و کنترل بین اجزا پراکنده شده
- ◀ کاربر نباید تشخیص بدے که پردازه دقیقا در کدام کامپیوتر در حال اجراست، و متوجه انتقال task ها بین اجزا هم نشه
- ◀ جایی که داده در اون ذخیره میشه و وجود **replication** از داده ها هم به کاربر ارتباطی نداره (transparency)
- ◀ این نکته در سیستم منسجم غیر قابل اغماض هست که ممکنه در اون بعضی از **node**‌ها fail بشن. اگرچه خرابی جزئی از هر سیستم پیچیده ای هست اما در سیستم های توزیع شده پنهان کردنش دشواره

ویژگی‌های اصلی سیستم‌های توزیع شده:

۱) توزیع شدگی جغرافیایی دارن

۲) ساعت فیزیکی یکسانی ندارن

۳) حافظه فیزیکی مشترکی ندارند

۴) ناممکن هستن

نکته: حواسمن باشه که سیستم‌های توزیع شده رو با سیستم‌های موازی یکی در نظر نگیریم. سیستم‌های موازی ویژگی‌هایی نظیر:

- همبستگی خیلی محکمی دارن (به اصطلاح twisted) ولی سیستم‌های توزیع شده همبستگی شُلکی دارن.
- ساعت فیزیکی مشترک دارن.
- یک حافظه فیزیکی به اشتراک گذاشته شده.
- همگن هستن

اصول سیستم‌های توزیع شده: (اهداف طراحی)

۱. Connectivity

Supporting resource sharing : اتصال remote به منابع توزیع شده برای کاربران آسونه

منابع میتوونن هر چیزی باشن و تمایل به اشتراک گذاری اونها میتوونه به دلایل اقتصادی باشه همکاری و تبادل اطلاعات رو برای کاربران آسون میکنه

مثال موفقی از به اشتراک گذاری منابع در سیستم‌های توزیع شده BitTorrent هست که file sharing هستش P2P

۲. Transparency

توزیع پردازه‌ها و منابع از دید کاربر مشخص نیست

Openness : سیستم‌های توزیعی الزاماً باید open باشن. یعنی میرایی کمتری داره چون میتوونه به محیط خودش تاثیر بذاره و تاثیر هم بپذیره.

Open distributed system = سیستمی هست که اجزایی رو ارائه میده که میتوونن به راحتی توسط سیستم‌های دیگر مورد استفاده قرار بگیرند یا با اون‌ها ادغام بشند و در عین حال خودش شامل اجزایی هست که از جای دیگری originate شدن.

Open بودن به این معناست که اجزا باید قوانین استانداردی که semantic syntax و سرویس‌هایی که ارائه میدن رو توصیف میکنه، توافق کنن. یک رویکرد تعریف سرویس‌ها با استفاده از IDL یا زبان

تعريف interface هست که همیشه فقط syntax را شامل میشه و semantic به صورت غیررسمی و با زبان های طبیعی داده میشه. دو پردازه از طریق interface ها با هم ارتباط میگیرن.

مشخص میکنه که ۲ پیاده سازی مجزا یا دو جز با سازنده متفاوت با تکیه بر سرویس هایی که توسط استاندارد مشترک مشخص شده با هم کار کنند. ◀

مشخص میکند که app ای که در سیستم توزیع شده A توسعه پیدا کرده میتوانه بدون تغییری در سیستم توزیع شده B که همون interface سیستم A را پیاده سازی میکنه، اجرا بشه یا خیر. Extensibility = یعنی افزودن اجزای جدید آسون و همچنین جایگزینی اجزا بدون تاثیر بر سایر اجزای موجود باشه. یعنی قابلیت تغییر پیکره بندی رو داشته باشه. ◀

۴. Scalability: از لحاظ جغرافیا، administration و سایر

Size scalability = سیستم باید کاربران و منابع بیشتر رو پشتیبانی کنه. اگر یک سرویسی تنها در یک ماشین پیاده سازی شده باشه به ۳ علت میتوانیم با گلوبال مواجه شیم:

CPU يا همون محدودیت در computational capacity ◆

I/O شامل نرخ Storage capacity ◆

network between the user and the centralized service ◆

Geographical scalability = از دلایل دشوار بودن مقیاس پذیری سیستم های توزیع شده ای که برای LAN طراحی شدن اینه که مبتنی بر synchronous communication هستند. در این حالت client به حالت block در میاد تا جواب از سرور برسه که برای LAN که ارتباط ۲ دستگاه در بدترین حالت در حد چند میکرو ثانیه هست مناسبه، اما برای wide-area system ارتباط کنده و نیاز به دقت بیشتری برای این ارتباطات وجود داره. همچنین در WAN برای پیاده سازی سیستم توزیع شده reliability هم کمتره و یک چالش هست. علاوه بر این پهنای باند محدود هم یک چالش در توزیع شدگی جغرافیایی گستردس.

Administrative scalability = مسئله ای که در این بخش مطرحه سیاست های متقاض در مورد استفاده از منابع و پرداخت هزینه اونها، مدیریت، امنیت منابع توزیع شده هست. معمولاً اجزایی که در یک domain قرار دارند برای کاربرای اون domain قابل اعتمادن. اگر یک سیستم توزیع شده به دیگری extend بشه ۲ اقدام امنیتی باید انجام بشه. سیستم توزیع شده باید از خودش در برابر حملات مخرب domain جدید محافظت کنه و بالعکس.

۵. High availability: یه سیستم توزیعی میخوام که بتونه به من خدمت برسونه.

Networking or Connectivity

برای اینکه بتونم سیستم توزیع شده‌ای ایجاد کنم، ابتدا نیازه که فسلفه‌ی شبکه‌بندی و اتصالات بین سیستم‌ها رو درک کنیم.

در این جلسه موارد زیر بحث خواهد شد:

- انواع شبکه
- اصول شبکه‌سازی: لایه‌بندی و کپسوله‌سازی

سوال: چرا برنامه‌نویس‌های سیستم‌های توزیع شده باید درباره شبکه اطلاعات داشته باشند؟

مشکلات شبکه‌بندی معمولاً روی کارایی، تحمل پذیری خطأ و امنیت سیستم‌های توزیع شده تاثیر می‌ذاره.

یه نقل قول جالبی از یه سخنران گوگل بود که می‌گفت:

«اونی که تو گوگل نشسته بوده فک میکرده این سیستم توزیع شده‌ای که دارن می‌تونه جواب‌گو باشه ولی دیده تعداد request هایی که داره به سمت گوگل میاد به قدری زیاده که روتراها دیگه overload شدن و نمی‌تونن پاسخ‌گو باشن. در واقع منطق برنامه من درسته ولی شبکه و روتر من نمی‌تونه این حجم رو مدیریت کنه که بتوونه خوب پاسخ‌گوی کاربر باشه.»

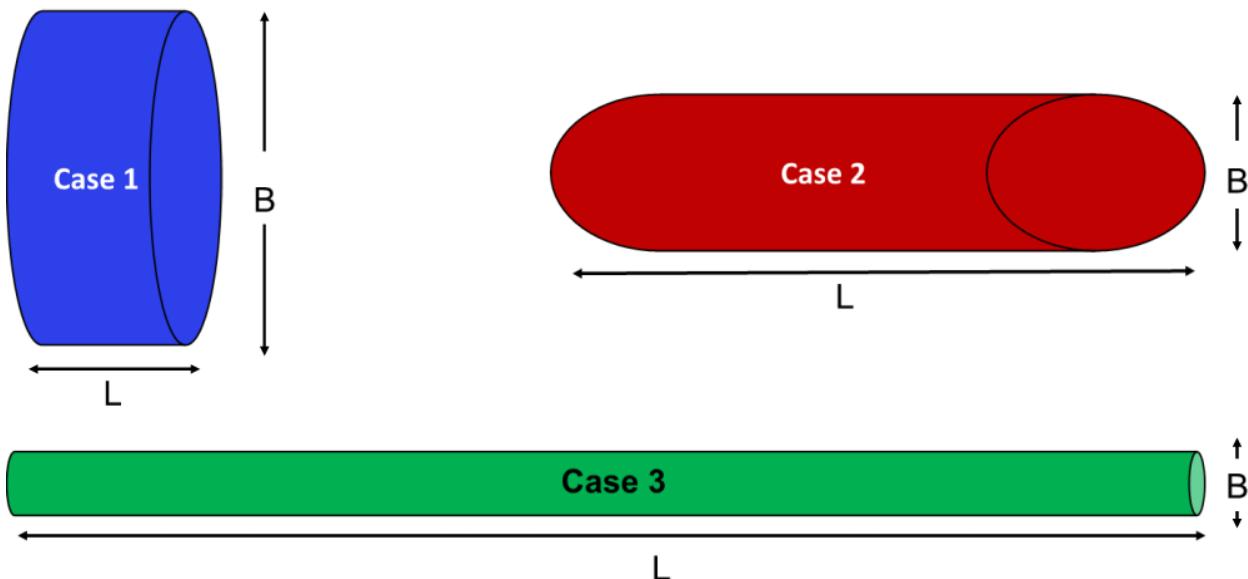
در کل برای ارتباط بین اجزای مختلف سیستم‌های توزیع شده به دلیل ماهیت توزیع شدگی جغرافیایی‌شون به شبکه‌ای که اونا رو بهم ارتباط بده نیاز داریم.

هر شبکه به صورت کلی دو متریک مشخص به نام‌های:

- تاخیر یا Latency

- پهنای باند یا Bandwidth

داره که شبکه رو باهش توصیف می‌کنم. در شکل زیر سه نمونه شبکه با پهنای باند و تاخیر متفاوت رو مشاهده می‌کنیم.



ما فکر میکنیم که هر چقدر $B \times L$ بیشتر باشه در واقع سرعت انتقال داده بیشتری رو تجربه میکنیم. ولی در واقع هر چقدر حاصلضرب این دو بیشتر بشه، **عدم قطعیت** تو شبکه بالا میره یعنی ممکنه داده بیشتری تو شبکه گم بشه. معمولاً تو شبکه برای اینکه بتونم داده‌ها رو کنترل کنم که گم نشن یه بافر هم در نظر میگیرم. ولی با بالا رفتن مقدار $B \times L$ تو شبکه ما (به مرز ترکیبی رسیدن) رو هم تجربه خواهیم کرد.

ملاحظاتی که در شبکه‌ها میشه تعریف کرد:

نظرات در مورد طراحی سیستم‌های توزیع شده	ملاحظات شبکه‌ها
میتوانه روی تأخیر و نرخ ارسال داده تو پیام‌ها تاثیر داشته باشه	Performance
اندازه اینترنت داره بزرگتر میشه و باید انتظار ترافیک بیشتر در آینده رو داشته باشه	Scalability
خطاهای ارتباطی رو شناسایی و یه چک لیست خطایی در لایه اپلیکیشن ترتیب بدیم	Reliability
نصب فایروال‌ها. قرار دادن مازول‌های احرار هویت انتها به انتها، حریم خصوصی و امنیتی	Security
اگر من متوجهم پس به این معنیه که ارتباط من پایدار نیست و باید دید شبکه من این متوجه بودن رو چطور پشتیبانی میکنه	Mobility

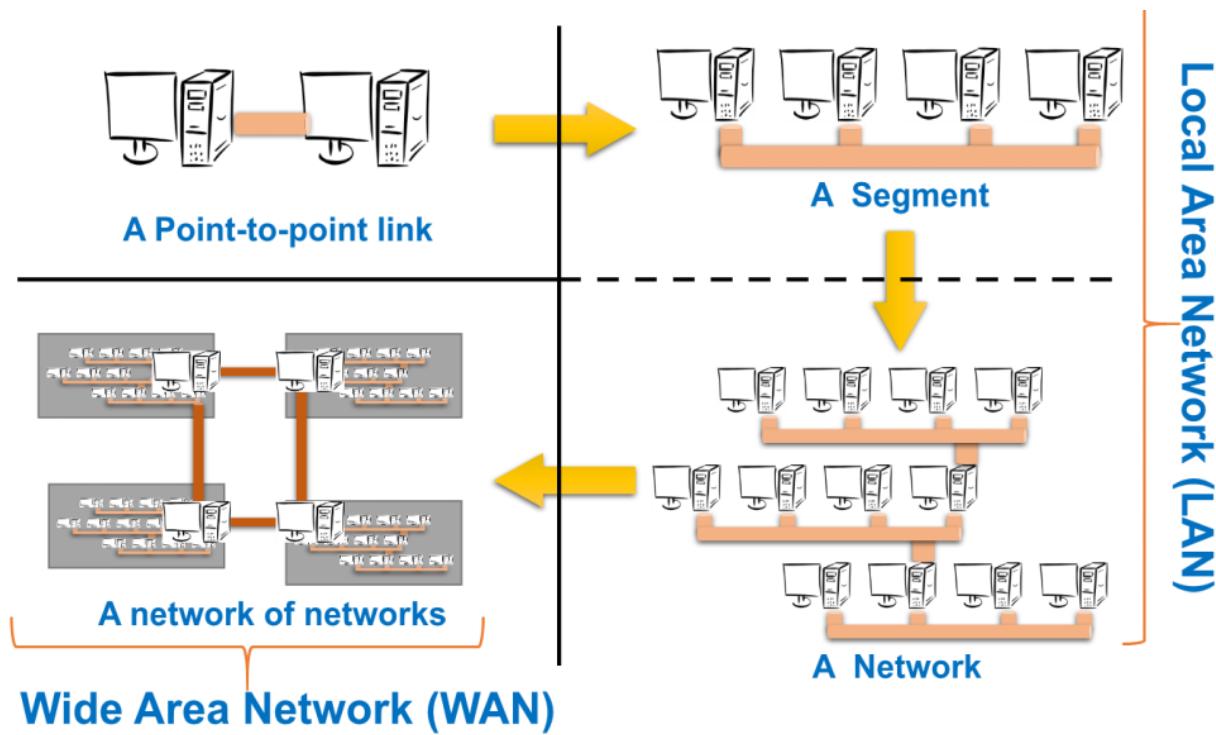
<p>آیا QoS رو شبکه‌ی من میتوانه مدیریت کنه یا نه؟ اگر داریم multimedia تو سیستم‌های توزیع‌شده کار میکنیم آیا شبکه میتوانه مدیریت‌شون کنه یا نه.</p>	<p>QoS</p>
--	-------------------

دسته‌بندی شبکه‌ها:

معمولًا شبکه‌ها را از دو بُعد میشه دسته‌بندی کرد:

۱. بر اساس اندازه

- Body Area Network (BAN)
 - یکی یه سری سنسور به خودش بسته که اطلاعاتی رو درباره خودش دریافت میکنه. معمولًا این انرژی و هزینه پایینی دارن.
- Personal Area Network (PAN)
 - وسایلی مثل موبایل و لپ تاپ و تبلت و غیره که متعلق به یه فرد رو به هم با بلوتوثی یا wifi ای وصل باشن.
- Local Area Network (LAN)
 - یه رسانه ارتباطی تا چند صد متره، نرخ اطلاعاتم بالا، تاخیر خیلی کم، نیازی به مسیریابی هم نیست.
- Wide Area Network (WAN)
 - نمونش میشه همون اینترنت، ابعادش متفاوته، روتینگ نیاز داریم. اینجا پهنای باند و تاخیر با شبکه‌های قبلی فرق میکنه.



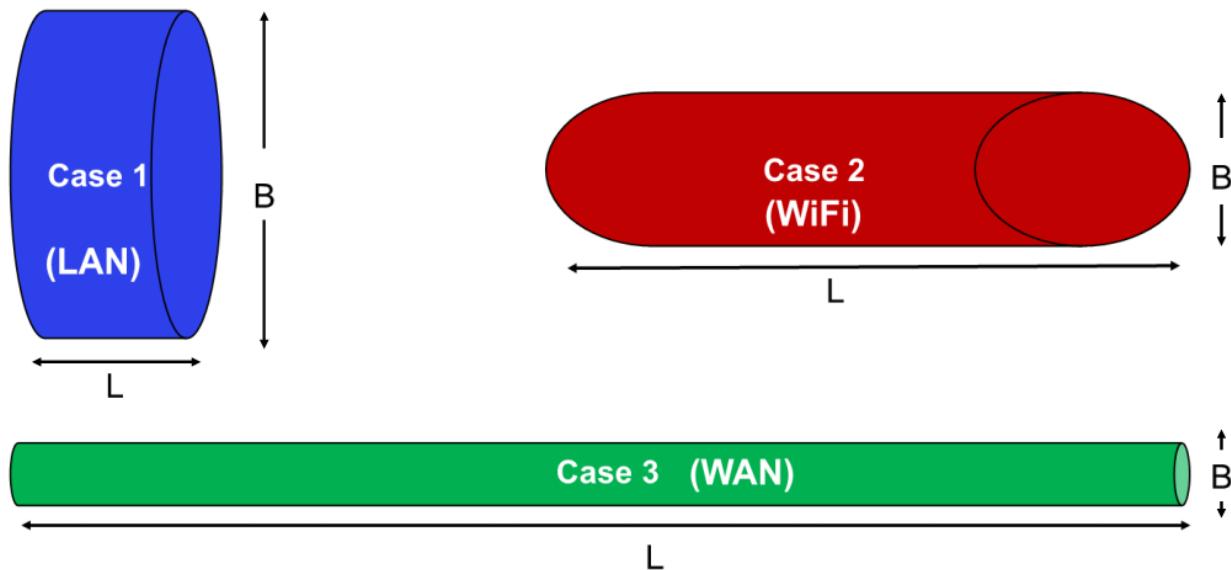
۲. بر اساس توپولوژی

- Ethernet networks
- Wireless networks
- Cellular networks

خلاصه دسته‌بندی شبکه‌های مختلف را تو جدول زیر می‌توانیم بینیم

Network	Example	Range	Bandwidth (Mbps)	Latency (ms)
Wired LAN	Ethernet	1-2 km	10 – 10,000	1 – 10
Wired WAN	Internet	Worldwide	0.5 – 600	100 – 500
Wireless PAN	Bluetooth	10 – 30 m	0.5 – 2	5 – 20
Wireless LAN	WiFi	0.15 – 1.5 km	11 – 108	5 – 20
Cellular	2G – GSM	100m – 20 km	0.270 – 1.5	5
Modern Cellular	3G	1 – 5 km	348 – 14.4	100 – 500

در شکل پایین دید بهتر نسبت به اون شکل تاخیر و پهنه‌ی باند بدست میاریم:



تعريف پروتکل:

به تفاهمنامه و استانداردسازی که برای ارتباط میان مولفه‌های مختلف شبکه مورد نیاز خواهد بود.

مثلًا:

- چقدر بیت لازمه تا بشه سیگنال ، یا ۱ فرستاد؟
- گیرنده از کجا متوجه آخرین بیت ارسالی بشه؟
- گیرنده از کجا میتونه بفهمه که پیام آسیب دیده یا نه؟

هر جز تو شبکه ممکنه نظر و سلایق مختلفی داشته باشن (ناهمگن) ولی وقتی در یه بستری مثل اینترنت میخوان با هم ارتباط برقرار کنن باید از یه سری قوانین استاندارد و شناخته شده تبعیت کنن.

هر داده رو میایم تو شبکه برای راحتتر فرستادن و جلوگیری از گم شدن اطلاعات در قالب‌هایی با سایز مشخص ارسال کنیم (MTU) مثل بازی زو

لایه‌بندی شبکه:

لایه‌بندی طراحی مقیاس‌پذیر و مازولار برای نرم‌افزارهای پیچیده‌ست.

در شکل زیر عملکرد مرسوم نرم افزار شبکه را بررسی می کنیم:

Functionality	Layer
Transmits bits over a transmission medium	Physical
Coordinates transmissions from multiple hosts that are directly connected over a common medium	Data link
Routes the packets through intermediate networks	Network
Handles messages – rather than packets – between sender and receiver processes	Transport
Satisfies communication requirements for specific applications	Application

این تعریف کلی لایه بندی شبکه بود ولی یه مدل OSI یا Open Systems Interconnection که مدل مرجع هست داریم که به شکل زیر نشان داده می شود:

Functionality	Layer	Example Protocols
Satisfy communication requirements for specific applications	Application	HTTP, FTP
Transmit data in network representation that is independent of representation in individual computers	Presentation	CORBA data representation
Support reliability and adaptation, such as failure detection and automatic recovery	Session	SIP
Handle messages – rather than packets – between sender and receiver processes	Transport	TCP, UDP
Route the packet through intermediate networks	Network	IP, ATM
Coordinate transmissions from multiple hosts that are directly connected over a common medium	Data-link	Ethernet MAC
Transmit bits over a transmission medium	Physical	Ethernet

کپسوله سازی بسته ها در شبکه:

لایه application من تو مبدا می خواهد با لایه application مقصد ارتباط برقرار کنه. مطابق شکل زیر در هر لایه از شبکه داده ای به بسته اطلاعاتی اضافه شده و توسط هدر اون لایه خاص کپسوله شده و تحويل لایه بعدی در مبدا پیام می شود.

در نهایت این بسته به مقصد رسیده و از لایه فیزیکی مقصد به صورت لایه لایه باز میشه تا به لایه application مقصد برسه.

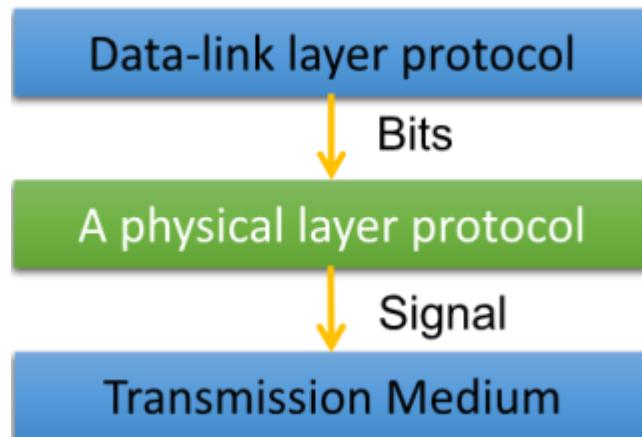
انتهای جلسه ۴

جلسه ۵

تو این فصل شروع به توضیح و شرح لایه به لایه شبکه خواهیم کرد:

لایه فیزیکی

پروتکل‌های لایه فیزیکی دنباله‌ای از بیت‌ها روی یک رسانه‌ی انتقالی هدایت خواهند کرد. این لایه برای انتقال اطلاعات، بیت‌ها رو به سیگنال مدوله می‌کنه. پس در واقع لایه فیزیکی ما میشه همون رسانه واقعی فیزیکی که دیتا روش منتقل میشه. اینجا دیگه خبری از فریم و پکت و اینا نیست و فقط با یه سری بیت و سیگنال سروکار داریم.



لایه پیوند داده

این لایه روی لایه فیزیکی میشینه و به عنوان اولین لایه کپسول‌مسازی شناخته میشه. اینجا با مفهومی به اسم فریم (frame) سروکار داریم. بر اساس یک سری پروتکل‌ها، این لایه دو نوع عملکرد برآش تعریف میشه:

(۱) نحوه هماهنگی (coordination) () بین فرستنده‌ها که frame ها چطوری با موقتیت دریافت بشن.

۲) نحوه شناسایی host ها در شبکه محلی (addressing over local networks))

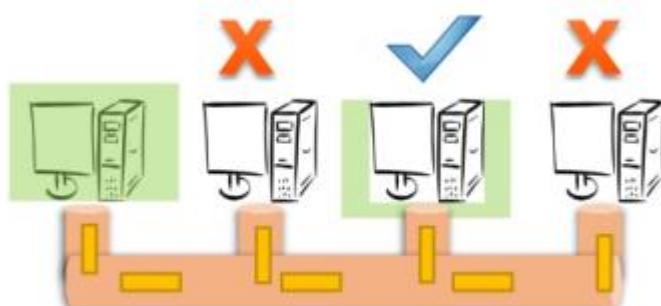
حال به تشریح این دو وظیفه به صورت دقیق‌تر خواهیم پرداخت:

۱. Coordination .

- در واقع ارسال و دریافت سالم یک frame در شبکه به عهده‌ی این لایه هست.
- ارسال همزمان دو بسته در شبکه توسط دو host مختلف به صورت همزمان بر روی یک لینک باعث ایجاد تصادم یا collision می‌شود. پس لایه پیوند داده باید این تصادم را از بین ببره یا به کاری کنه اصن رخ نده.

۲. Addressing over local networks .

- به منظور شناسایی host ها در شبکه محلی، هر دستگاه متصل به شبکه، با یک آدرس منحصر به فرد فیزیکی به نام MAC (Medium Access Control) صدا زده می‌شود.
- مثال: byte long ۶ که A:D4:AB:FD:EF:8D است.



رویکرد لایه پیوند داده به این صورته که می‌باید بسته را روی اون رسانه ارتباطی broadcast می‌کند و گیرنده‌های پیام میان اون header بسته را می‌خونند و اگر گیرنده خودشون بودن اون پیام را ضبط می‌کنند.

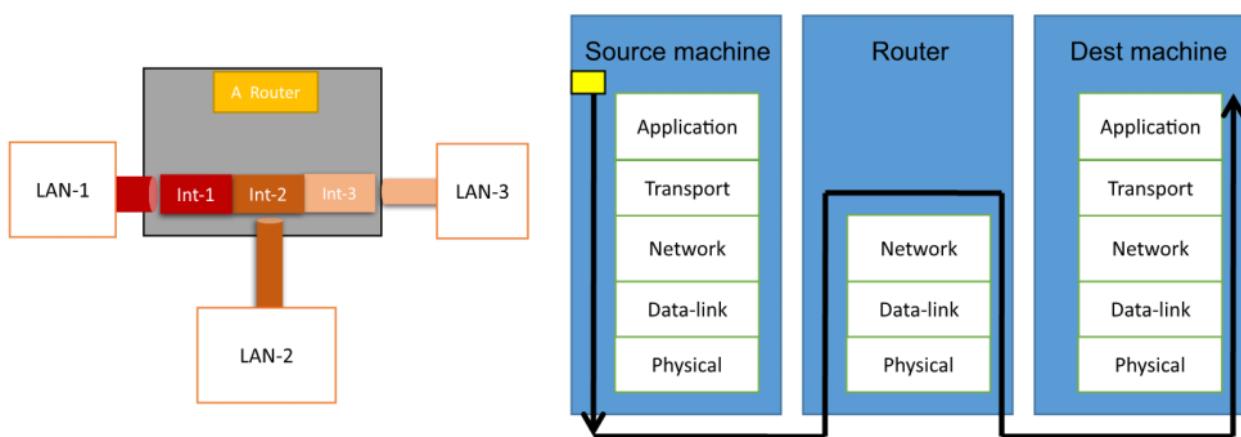
لایه شبکه

- پروتکل‌های لایه شبکه نقش مسیریابی یا routing را به عهده دارند. یعنی بباید یه مسیر از مبدأ به مقصد برای اون بسته‌ای که می‌خواهد منتقل بشه پیدا کنه.
- این مسیر هم می‌توانه ویژگی‌های نظری کوتاه‌ترین، کم هزینه‌ترین، مطمئن‌ترین و غیره را داشته باشد.

- این لایه پروتکل معروفی به نام Internet protocol یا IP رو هم داخل خودش داره که به صورت بسیار گسترده در سطح انواع شبکه مورد استفاده قرار می‌گیره. این پروتکل معمولاً به منظور شناسایی ماشین‌ها در سطوح بالاتر و یا خود شبکه محلی (local network) مورد استفاده قرار می‌گیره.

Router

دستگاهی هست که وظیفه مسیریابی بسته‌ها بین شبکه‌های مختلف رو به صورت پویا انجام میده. قبیماً به صورت سخت‌افزاری ساخته می‌شود ولی الان به صورت نرم‌افزاری - سخت‌افزاری یا کامل نرم‌افزاری انجام می‌شه.



همینطور که از تو شکل هم مشخصه روترا تا لایه سوم رو میتوانه باز کنه و به بالاترش کاری نداره.

الگوریتم‌های مسیریابی

حالا اینکه این فرآیند مسیریابی به چه نحوی صورت بگیره هم خودش مبحث الگوریتم‌های مسیریابی رو به میون میاره. اینکه بر طبق چه سیاست‌ها و تصمیم‌گیری‌هایی این انتقال بسته بین شبکه‌های مختلف صورت بپذیره.

مفهومی طراحی این الگوریتم‌های مسیریابی در اینترنت هم چالش‌های خاص خودش رو داره:

- کارایی (performance): ترافیک در شبکه‌های مختلف با هم متفاوته و ممکنه در رسیدن یه بسته خلی ایجاد بشه.
- Router failures: روترا ممکنه تو اینترنت دچار اختلال بشن که در فرآیند مسیریابی تاثیرگذار خواهد بود و نیازمند تهییه مسیر مناسبی می‌باشد.

الگوریتم‌های مسیریابی دو کار مهم باید انجام بدن:

(۱) بعدی که هر packet بخواهد بهش دست پیدا کنه رو باید مشخص کن.

الگوریتم باید سریع و بهینه باشه (البته میتوانیم هر معیار دیگه‌ای رو مطابق خواست و سیاست خودمون داشته باشیم).

(۲) به صورت پویا اطلاعات اتصالات شبکه رو بروز کنه.

وظیفه نگهداری و بازسازی اطلاعات شبکه به واسطه monitoring روترا و ترافیک شبکه

فعالیت‌های بالا به صورت توزیع شده در سطح شبکه قرار می‌گیرد.

- تصمیمات مسیریابی به صورت hop by hop انجام می‌شه.

- اطلاعات روتراهای احتمالی hop بعدی به صورت محلی در هر روتر ذخیره می‌شه.

- این اطلاعات هم باید به صورت دوره‌ای بروز بشه.

یکی از انواع الگوریتم‌های ساده‌ی مسیریابی «Distance vector» هست. این نوع الگوریتم‌ها از مباحث تئوری گرافی

برای پیدا کردن مناسبترین مسیر در شبکه استفاده می‌کنن. در واقع از الگوریتم کوتاه‌ترین مسیر (shortest path) یا

(bellman-ford) ممکن بهره‌مند می‌شه.

چرا بالا گفتیم مناسبترین مسیر و چرا نگفتیم بهترین؟

چون کلمه بهترین وزن سنگینی با خودش حمل می‌کنه و وقتی به چیزی بگیم بهترین یعنی از هر جهتی بهتر هستش ولی وقتی بگیم مناسبترین یعنی یه سری فاکتور مدنظرمون بود و با توجه به اوها این قید تناسب رو مطرح کردیم. لزوماً تو همه چی بهترین نبوده.

دو فعالیت الگوریتم‌های DV:

(۱) مشخص کردن بهترین hop بعدی در هر روتر

(۲) بروزرسانی پویای اطلاعات اتصالات در همه روترا

در این نوع الگوریتم‌ها هر روتر یه جدول مسیریابی نگهداری می‌کنه:

• مقصد: آدرس IP مقصد بسته

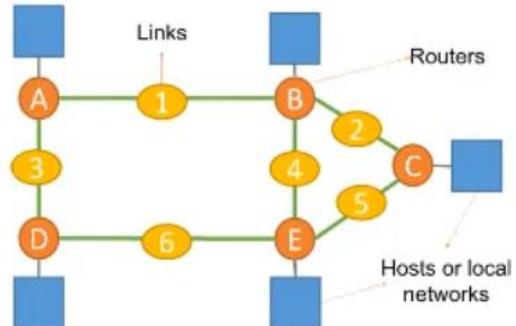
• پیوند (link): لینک یا اینترفیس خروجی که بسته باید به سمتش فوروارد بشه.

• هزینه (cost): فاصله‌ی بین روتر و مقصد

در شکل زیر یه سناریو جامع از مطالبی که در بالا شرح داده شد رو می‌شه دید.

Routings from A			Routings from B			Routings from C		
To	Link	Cost	To	Link	Cost	To	Link	Cost
A	local	0	A	1	1	A	2	2
B	1	1	B	local	0	B	2	1
C	1	2	C	2	1	C	local	0
D	3	1	D	1	2	D	5	2
E	1	2	E	4	1	E	5	1

Routings from D			Routings from E		
To	Link	Cost	To	Link	Cost
A	3	1	A	4	2
B	3	2	B	4	1
C	6	2	C	5	1
D	local	0	D	6	1
E	6	1	E	local	0



یکی از انواع الگوریتم‌های DV، میشه به RIP یا Router Information Protocol اشاره کرد. این الگوریتم کارای زیر رو انجام میده:

- ۱) ارسال جداول مسیریابی به روترهای همسایه (به صورت دوره‌ای یا هر وقت جدول بروز شد)
- ۲) وقتی جدول مسیریابی از همسایه‌ی خودش دریافت کنه، کارای زیر رو انجام میده:

Case	If the received routing table ...	Updates to the local routing table
1	Has a new destination that is not in the local routing table	Update the Cost and Link
2	Has a better-cost route to a destination in the local routing table	Update the Cost
3	Has a more recent information	Update the Cost and Link

شبه کد الگوریتم RIP

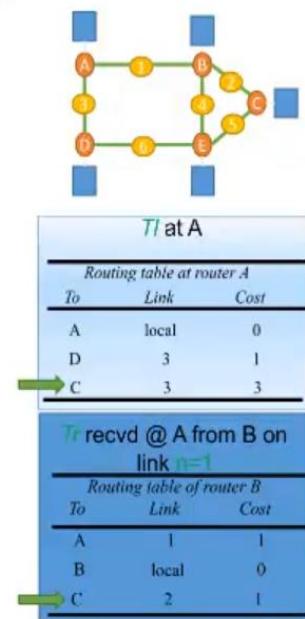
Send: Each t seconds or when Tl changes, send Tl on each non-faulty outgoing link

Receive: Whenever a routing table Tr is received on link n :

```

for all rows  $Rr$  in  $Tr$  {
    if ( $Rr.link \neq n$ ) {
         $Rr.cost = Rr.cost + 1$ ; // Update cost
         $Rr.link = n$ ; // Update next-hop
        if ( $Rr.destination$  is not in  $Tl$ ) {
            add  $Rr$  to  $Tl$ ; // add new destination to  $Tl$ 
        }
    }
    else for all rows  $Rl$  in  $Tl$  {
        if ( $Rr.destination = Rl.destination$ ) {
            //  $Rr.cost < Rl.cost$  : remote node has better route
            //  $Rl.link = n$  : information is more recent
            if ( $Rr.cost < Rl.cost$  OR  $Rl.link = n$ ) {
                 $Rl = Rr$ ;
            }
        }
    }
}
}

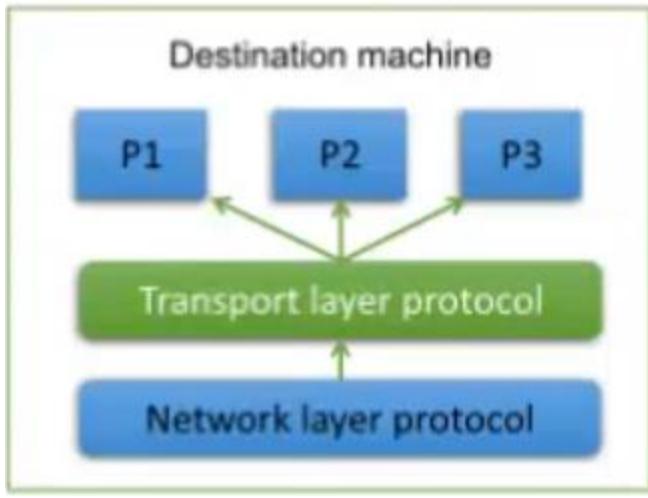
```



لایه Transport یا ترابری

اگر فرض کنیم وزارت راه ما متصدی لایه شبکه (network) باشند، اما اینکه از این مسیرها چه چیزهایی می‌شنوند بشه يعني مثلًا از آزادراه تهران - شمال تریلی و کامیون سنگین و اینا نمی‌توانه عبور کنه دیگه وارد حوزه transportation می‌شیم که به لایه انتزاعی روی لایه شبکه اضافه می‌کنند.

- پروتکل‌های لایه transport یک ارتباط end-to-end را برای برنامه‌ها مهیا می‌کنند. (مثلًا به یکی می‌گیم به باری رو از تهران ببر شیراز و اصن کاری نداریم چطوری و با چی می‌بریم، فقط می‌گم من اینجا اینا رو بهت سالم تحويل دادم اونورم ازت همینطوری تحويل می‌گیرم.)
- اینجا پایین‌ترین لایه‌ای هست که پیام‌ها (به جای بسته‌ها) مدیریت می‌شون.
- پیام‌ها به جای اینکه با آدرس جابه‌جا بشن با یه سری پورت‌های ارتباطی که به پردازه‌های مختلف اختصاص داده شده، ارتباط برقرار می‌کنند.
- لایه transport هر port دریافتی رو به مخصوص خودش ارتباط میده.



پروتکل‌های **ساده Transport** شامل سرویس‌های زیر هستند:

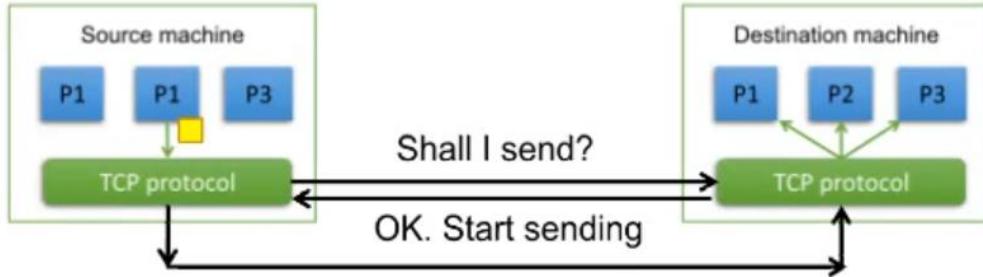
- (۱) سرویس **Multiplexing**
- (۲) ارتباط **connection-less**: پردازهای فرستنده و گیرنده، قبل ارسال پیام‌ها هیچ ارتباطی را ایجاد نمی‌کنند. میشود **connectionless** مثل UDP.

- در این لایه هر پیام در داخل **datagram** کپسول‌سازی می‌شود.
- پیام‌هایی که تو سمت گیرنده دریافت می‌شوند ممکن است با ترتیب ارسالی فرستنده متفاوت باشند.

اگر بخوایم **پیشرفتمند** به این سرویس‌هایی که توسط پروتکل‌های لایه Transport ارائه می‌شوند، اشاره کنیم، به پروتکل TCP می‌پردازیم. که این پروتکل:

- (۱) ارتباط **connection oriented** را در اختیار میدارد (three way handshake)
- (۲) اطمینان‌پذیری (reliability)
- (۳) مدیریت congestion (که در UDP نیازی نبود بهش چون اطمینان‌پذیری مهم نبود)

- در این مدل ارتباطی قبل از ارسال پیام بین فرستنده و گیرنده، حتماً یک دست دادن سه طرفه استفاده می‌کنند.



- برخلاف connection less، مقصد داده‌ها را به همون ترتیبی که مبدأ ارسال کرده دریافت می‌کند.
(مقصد یه بافری درست می‌کنه)

Reliability

بسته‌های تو شبکه به دلایلی مثل buffer overflow یا خطا در حین ارسال ممکن است بین برن. به همین دلیل مکانیزم‌های مختلفی برای ایجاد اطمینان‌پذیری یا reliability داشته باشیم. مثل:

- دوباره بفرست که در این سیستم فرستنده و گیرنده با ack و syn این دریافت را تایید می‌کنند. کنار هر ack هم یه عدد یا sequence می‌ذاریم که مشخص باشه بسته داره دوباره فرستاده می‌شود یا هر چی.

Congestion control

ظرفیت شبکه‌ها به واسطه‌ی لینک‌های مجازی ارتباطی و روتورها محدود شده. حال چه اتفاقی می‌وقتی اگر مبدأ بسته‌ها را با نرخی بیشتر از ظرفیت شبکه ارسال کند؟

- بسته‌ها توی روتورهای میانی drop بشوند.
- مربوط به اون پکت تو مبدأ دریافت نشوند.
- مبدأ مجدد پیام را ارسال می‌کند.
- بسته‌های بیشتری در صف روتور ایجاد می‌شوند.
- شبکه به فنا میره.

برای جلوگیری از congestion دو رویکرد قابل انجام است:
(۱) شناسایی congestion در سطح روتورها

اگر روتور انتظار buffer overflow داشت، به صورت عادی دو رویکرد زیر را اتخاذ می‌کند:

- یه بسته ECN برای ورودی ارسال میکنه.
- یه سری پکت رو drop کنه و به مبدا یه جوری بگه با مشاهده بسته‌های از دست رفته ورودی رو تنظیم کنه.

۲) خود مبدأ ورودی رو سمت خودش تنظیم کنه.

اگر فرستنده TCP باعث ایجاد congestion بشه (مثلًا پکت congestion دریافت کنه)، بعدش میاد نرخ ورودی رو تنظیم میکنه.

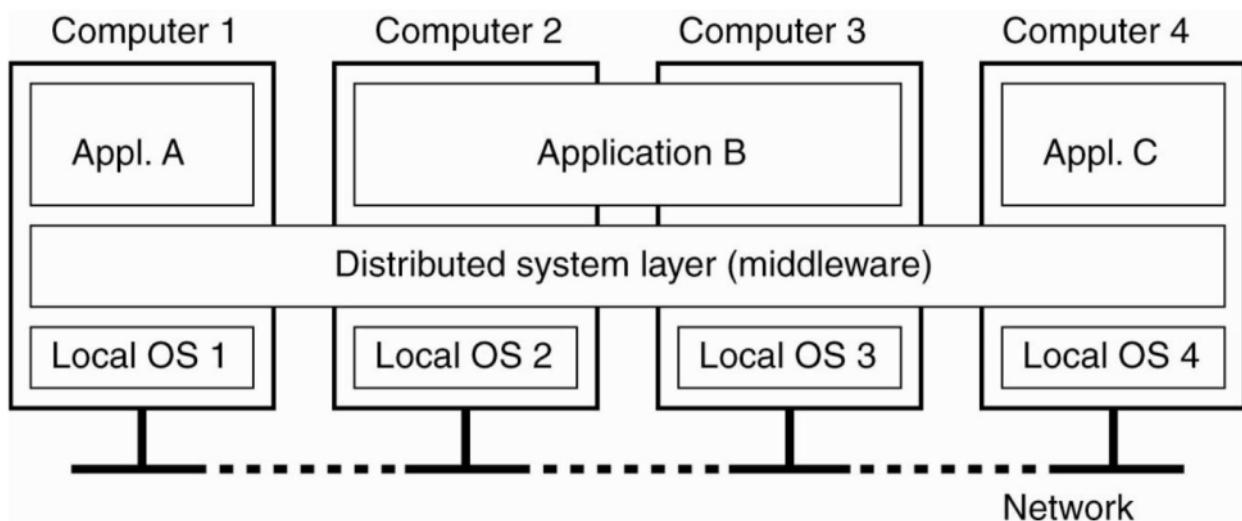
انتهای جلسه ۵

جلسه ۶:

در این جلسه مروری بر مفاهیم سیستم‌های توزیع شده خواهیم داشت و در ادامه به مفهوم RPC پرداخته خواهد شد.

یادآوری تعریف سیستم‌های توزیع شده:

مجموعه‌ای از کامپیوترهای مستقل از هم که از دید کاربر به عنوان یک سیستم منسجم به نظر می‌رسد.



این شکل بیان‌گر یک سیستم توزیع شده است. همانطور که مشاهده می‌شود هر کامپیوتر ممکن است مربوط به خودش رو داشته باشد. بر روی این سیستم‌های عامل یک لایه middleware قرار گرفته که هدفش پنهان سازی پیچیدگی‌ها از دید کاربر. همچنین مشخص است که این سیستم‌های کامپیوتری از طریق شبکه با همیگه در ارتباط هستند. در این شماتیک

هر کامپیوتر برنامه مخصوص خودش رو داره، فقط برنامه B بین سیستم‌های ۲ و ۳ تقسیم شده که مجاز هم هست و به این معنیه که کد برنامه در هر دو وجود داره.

OS = نقشی که Middleware برای یک کامپیوتر بازی میکنه رو برای یک سیستم توزیع شده بازی میکنه

مدیریت منابع به اشتراک گذاشته شده و استقرار کارآمد منابع / مدیریتی که ارائه میده شامل:

Facilities for interapplication communication ◀

Security services ◀

Accounting services ◀

Masking of and recovery from failures ◀

مروری بر اهداف سیستم‌های توزیع شده

(۱): بتونم به مردم خودم رو به منابع در دسترس متصل کنم. Connecting users and resources

(۲): بعضی وقتاً میگیم transparent یعنی همه چی واضح و شفافه ولی اینجا به این معنیه که پیچیدگی‌های لایه‌های زیرین از دید کاربر مخفی میمونه. Transparency

(۳): اجزا داخل شبکه به راحتی بتونن با هم اطلاعات تبادل کنن. مثلاً ما تو ایران کلی قومیت و فرهنگ داریم که با هم متفاوت‌تر ولی در نهایت به هم‌شون میگیم ایرانی. حالا اگر ما بخواهیم با یکی تو یه کشور دیگه ارتباط داشته باشیم، این ارتباط زیاد open در نظر گرفته نمیشه و یه سری قواعد رو میطلبه.

میتونیم به تعبیری بهش interoperability هم بگیم.

سیستمی openness هست که انtrapوپی کمی داشته باشه یا به عبارتی دیگه میرایی پایینی داشته باشه. یعنی بتونه خودش رو با تغییر و تحولات تعديل بکنه.

(۴) مقیاس‌پذیری: سه نوع مقیاس‌پذیری داریم. اندازه، گسترده جغرافیایی، گسترده administration

(۵): بتونم دسترس پذیری به مردم‌ها رو با ۴ هدف بالا به راحتی برقرار کنم. Enhanced availability

شفافیت در سیستم‌های توزیع شده

توضیحات	شفافیت
پنهان کردن بیان داده‌ها و دسترسی به منبع	دسترسی (Access))
پنهان کردن مکان منابع (میتوانه ازشون استفاده کنه، فقط قرار نیست بدونه کجان)	مکانی (Location))
پنهان کردن احتمال اینکه یه منبع جابه‌جا بشه	Migration

پنهان کردن اینکه منبع حین استفاده جایه‌جا بشه.	Relocation
ویژگی‌های Location، Migration هر سه دارن می‌گن کاربر کاری نداره منبع کجاست؟ جایه‌جا می‌شه یا نه؟ یا حتی مکان این منبع از یه جایی بره جای دیگه. (منبع می‌تونه سخت‌افزار یا یک VM روی یک سخت‌افزار باشه.)	
پنهان کردن احتمال اینکه ممکنه چند تا کپی از یه منبع وجود داشته باشه. (به جهت اطمینان و دسترس‌پذیری)	Replication
ممکنه این سرویس که من می‌گیرم اختصاصی برای خودم نباشه و با کس دیگه‌ای هم به اشتراک‌گذاشته شده باشه... اینم پنهان کنش	Concurrency
پنهان کردن fail یا recovery منابع	Failure

مشکلات مقیاس‌پذیری

مثال	مفهوم
یک سرور تنها برای همه‌ی کاربران	سرویس‌های مرکزی
یک دفترچه تلفن آنلاین	داده‌های مرکزی
مسیریابی حول یه سری اطلاعات کامل رخ میده	الگوریتم‌های مرکزی

اگر سیستم رو توزیع شده در نظر بگیرم اونوقت می‌تونم سرویس‌ها، داده‌ها و الگوریتم‌ها را پخش و پلاکم و دیگه با این چالش‌های بالا مواجه نشم.

تعريف الگوریتم: توالی از یه سری دستورالعمل از پیش تعریف شده که بر روی کامپیوتر تعریف شده و هدفش حل یه سری مسائل باشه.

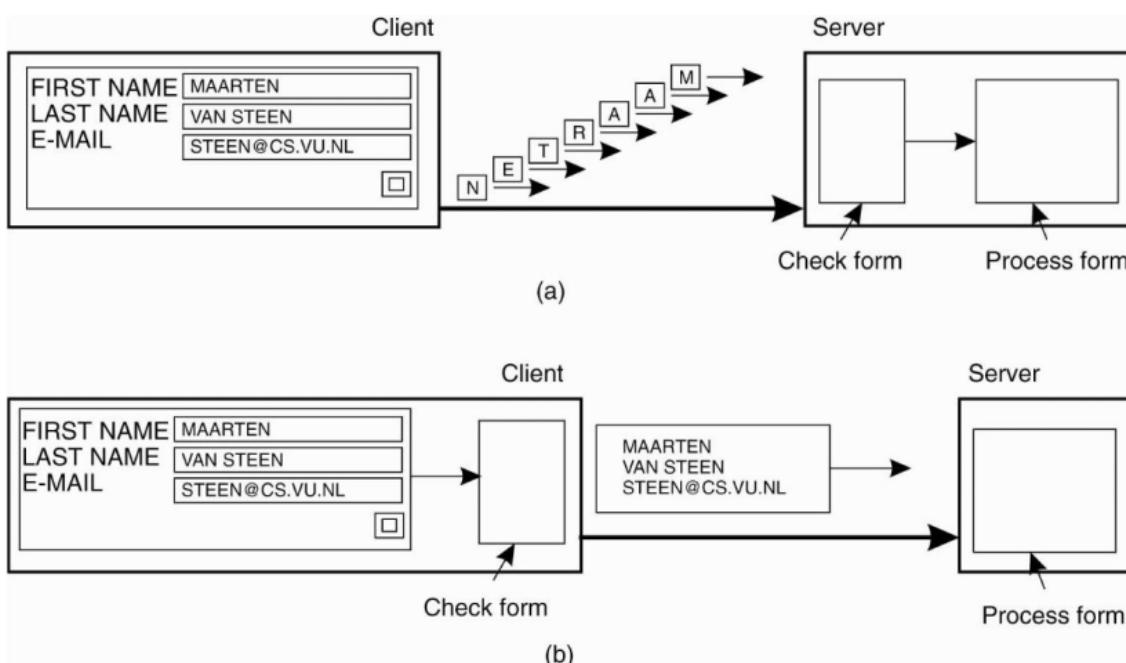
سیستم شبکه یا همون شبکه بانکی ایران رو می‌توانیم یه سیستم غیر مرکزی یا Decentralized خطاب کنیم ولی نمی‌توانیم بگیم که این سیستم Distributed هستش. حالا می‌گن به جای اینکه بیایم توزیع شده سیستم رو مقیاس‌پذیر بکنیم بریم غیر مرکز مقیاس‌پذیری رو انجام بدیم که باید گفت اونم مشکلات خوش رو داره:

- هیچ ماشینی اطلاعات کامل درباره وضعیت سیستم رو نداره.
- ماشین‌ها فقط بر اساس اطلاعات محلی تصمیم‌گیری انجام میدن.

- اگر یه ماشین این وسط fail بشه، بقیه کارا خراب نمیشه.
- فرض میکنیم که هیچ ساعت جهانی وجود نداره.

تکنیک مقیاس‌پذیری (۱)

تو شکل زیر داریم تفاوت بین اینکه یه کاربر چک کردن فرم رو انجام بدنه یا یه سرور رو بررسی میکنیم.



اگر هر کاربر بخواهد دونه دونه بخواهد فرم رو پر کنه و برای سرور بفرسته که سرور چک کنه ببینه آیا این فرم رو درست پر کرده یا نه، سربار خیلی زیادی روش میزاره. به همین خاطر میگه من میام بار اینو میزارم سمت کلاینت و میگم تو اینو انجام بدنه و وقتی کامل شد و درست بود، اینو یهو بفرست سمت من. (یه روش **Scale up** کردن سیستم) کاملا به عقل شعور ربط داره و نیاز نیست هزینه برای هرجی بکنیم.

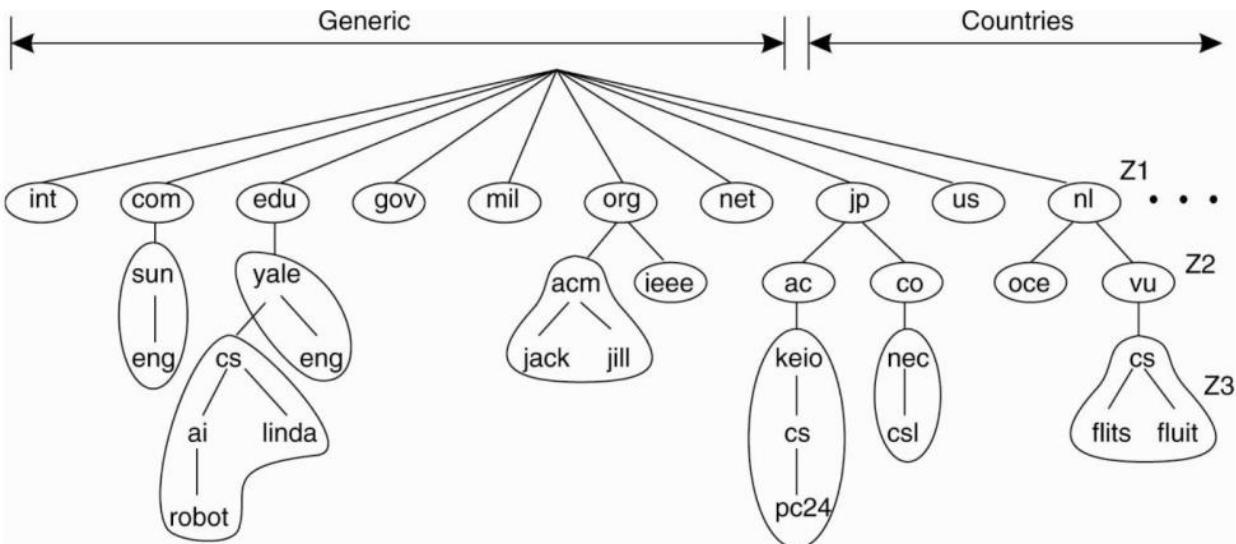
یا مثل سیاست کوکی سمت کردن سایت‌ها برای کاربرا به جای سایت روی خود وب بروزr کلاینت ذخیره میکنه.

پایان جلسه ۶

جلسه ۷

ارتباطات (RPC) یا Remote Procedure Call

در جلسات قبل در رابطه با اهداف سیستم‌های توزیع شده بحث کردیم و اولین هدف رو ارتباط بین منابع و کاربران عنوان کردیم. در جلسه قبل مقدماتی در رابطه با این هدف و این جلسه هم به صورت ریزتر به ابزارهای این هدف خواهیم پرداخت. جلسه قبل به یک تکنیک مقیاس‌پذیری در تشخیص صحت اطلاعات پر شده در فرم‌ها توسط سرور و کلاینت پرداخته شد و تکنیک دیگری هم وجود دارد که با مثال زیر برای سیستم آدرس دهی دامین‌ها در وب کاربرد دارد:



در این سیستم سلسله مراتب و ایزوله بودن هر موجودیت نیز رعایت شده است. به راحتی می‌شود به این سیستم چیزی اضافه کرد و هر کس در دامین خودش می‌توانه از accessing protocol های خودش استفاده کند. این ساختار scalable هست.

توسعه دادن و program کردن سیستم‌های توزیعی نسبت به سیستم‌های مرکزی پیچیده و سخت است. این سخت بودن به دلیل فرضیه‌ها و عادات غلطی هست که در ابتدای کار در نظر گرفته می‌شون:

- شبکه مطمئنه
- شبکه امنه
- اجزای شبکه همگن
- توپولوژی تغییر نمی‌کند
- تاخیر صفره
- پهنای باند نامحدوده
- هزینه‌ی انتقال صفره
- فقط یک administrator وجود دارد که سیستم را مدیریت می‌کند.

- همه اطلاعات دقیق و کامل است و همه عملیات‌ها درست هستند

RPC

در سیستم‌های توزیعی موجودیت‌ها (entities) تمایل دارند تا با همیگه ارتباط برقرار کنند. این ارتباط بین موجودیت‌ها میتواند در دو دسته طبقه‌بندی بشود:

□ موجودیت‌های مبتنی بر سیستم = system-oriented entities

□ Process

□ Thread

□ Node

□ موجودیت‌های مبتنی بر مسئله = problem-oriented entities

□ اشیاء (در روش‌های مبتنی بر برنامه‌نویسی شیگرا)

نکته: پارادایم مجموعه‌ای از روش‌هاست.

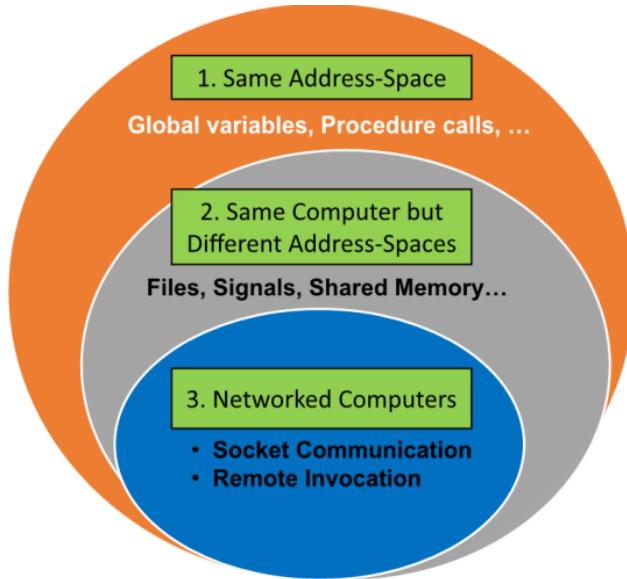
پارادایم‌های ارتباطی: مجموعه‌ای از روش‌هایی که به واسطه‌ی اونا «موجودیت‌ها میتوانند با همیگه تعامل داشته و داده تبادل کنن» را توصیف و طبقه‌بندی می‌کند.

با توجه به اینکه موجودیت من در چه گسترده‌ی جغرافیایی قرار دارد، میتوانیم این پارادایم‌های ارتباطی را در سه دسته مطابق شکل زیر طبقه‌بندی کنیم:

۱) همهی موجودیت‌ها در یک فضای آدرسی به سر می‌برن. مثل سیستم‌های تک پردازنده‌ای که همهی پردازه‌ها روی یه ماشین اجرا می‌شون (مثل متغیرهای global یا procedure call ها).

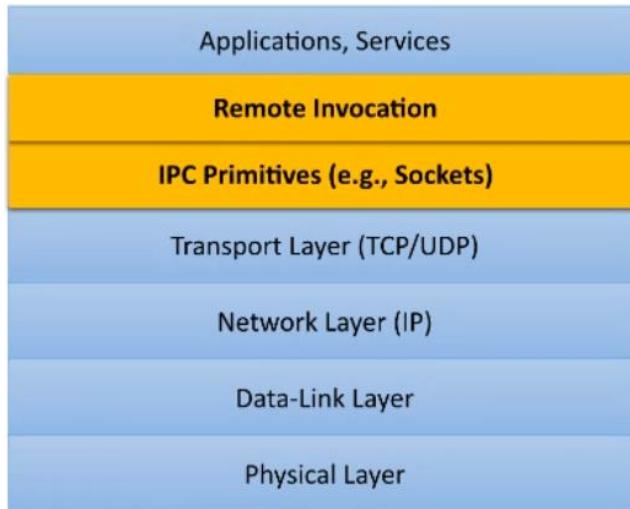
۲) همهی موجودیت‌ها در یک ماشین ولی با فضای آدرسی متفاوت باشند (چند تا فایل روی یک سیستم دارم که هر کدام فضای مخصوص خودشون را دارند). Shared memory در جاهایی که address space های مختلف داریم تعریف می‌شوند.

۳) موجودیت‌هایی که از طریق یک شبکه و سوکتی با هم در ارتباط و میخوان از امکانات همیگه استفاده کنند و امکانات مورد نیاز خودشون را فراخوانی بکنند.



امروزه مطالعه می‌کنیم که چگونه موجودیت‌ها در کنار شبکه‌های کامپیوتری با استفاده از socket و remote invocations در سیستم‌های توزیع‌شده ارتباط برقرار می‌کنند.

در سیستم‌های توزیعی ما زیاد به لایه‌های پایینی کاری نداریم. پس من قرار میان افزاری بالای لایه transport داشته باشم و به application (ممکن‌هه برنامه باشه یا developer) این امکان رو بده که بتونه با لایه‌های زیرین بدون اطلاع از پیچیدگی‌های اون لایه‌ها ارتباط برقرار کنه (مطابق شکل زیر).



سوکت

دو نوع ارتباط سوکت وجود دارد: TCP و UDP

۱) UDP

یک ارتباط connectionless را مهیا می‌کند و هیچ اطلاعاتی از وضعیت شبکه یا نیاز به باز ارسال بسته‌ها در شبکه ندارد. نداره retransmit کار می‌کنه، سرعت عمل بالایی داره اما نیست. مکانیسم ارتباطی در این نوع سوکت نیز به شرح زیر می‌باشد:

- سرور یک سوکت UDP به نام SS بر روی پورت شناخته شده sp باز می‌کند.
- سوکت SS منتظر دریافت یک درخواست می‌باشد.
- کلاینت‌ها یک UDP سوکت CS روی پورت رندوم CX باز می‌کنند.
- کلاینت سوکت CS یک پیام به آدرس ServerIP و پورت sp ارسال می‌کنند.



برای استفاده از UDP یه سری ملاحظات طراحی هم مدنظر قرار گرفته که خیلی تو دیوار نباشد:

- فرستنده باید به صورت صریح قبل از ارسال یک پیام بزرگ به قطعات کوچکتر تقسیم کنه (مدیریت پیام‌های طولانیشون انجام بد)
- ماکریم اندازه ۵۴۸ بایت برای هر انتقال پیشنهاد می‌شه.
- پیام‌هایی ممکنه بدون ترتیب به مقصد برسن
- اگر لازم باشه، برنامه‌نویس باید ترتیب بسته‌ها رو تغییر بد و out of order رو خودشون in order کنه.
- این ارتباط قابل اطمینان نیست
- پیام‌ها ممکنه حین فرآیند چک کردن check-sum یا buffer-overflow سمت روتر گم بشن
- گیرنده باید یک بافر به اندازه کافی برای پیام‌های فرستنده اختصاص بد

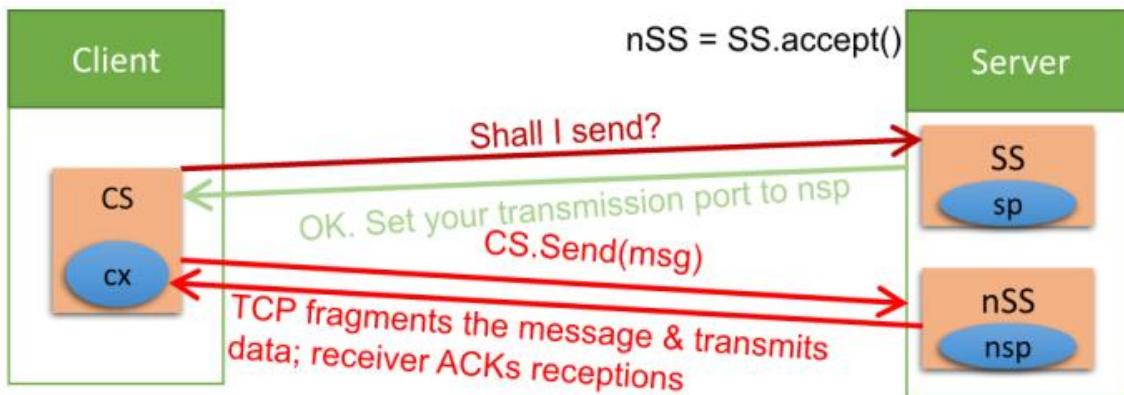
۲) TCP

در برابر UDP پروتکل TCP که **تحویل به ترتیب**, **اطمینان‌پذیری**, **کنترل ازدحام** رو داره, ارائه شد. البته این روش **هزینه‌بره**, **تاخیر بیشتری** داره و کنترل هست و ... مکانیسم ارتباطی نیز به نحو زیر است:

- سرور یک سوکت سرور TCP به نام SS بر روی پورت شناس sp باز می‌کنند.
- سرور منتظر می‌باشد تا درخواستی بهش برسه (با استفاده از accept call)
- کلاینت یه سوکت TCP روی یه پورت رندوم CX باز می‌کنند.

- سوکت CS یه پیام آغاز ارتباط (connection initiation message) به آدرس ServerIP بر روی پورت sp ارسال میکنه.

- سرور سوکت SS یه سوکت جدید NSS روی پورت رندوم nsp به کلاینت اختصاص میده.
- و به این ترتیب CS میتوانه به NSS داده ارسال کنه.



توی UDP میگفتیم که برنامنویس باید مسئلی که براش مهمه رو هندل کنه ولی اینجا این مسئولیت به عهده سوکت خواهد بود و به عهده application developer نیست و از دید اون transparent هست.

مزایای اصلی TCP

- TCP دریافت به ترتیب پیام رو تضمین میکنه.
- برنامه ها میتوانن پیام با هر اندازه ای رو ارسال کنن.
- TCP ارتباط قابل اطمینان رو با استفاده از ack و باز ارسال پیام تضمین میکنه.
- مکانیسم کنترل ازدحام یا congestion control نرخ ارسال فرستنده رو به منظور کاهش سربار شبکه، تنظیم میکنه.

پس یادمون نره کجاییم، الان میخوایم به ارتباط بین دو ماشین ایجاد کنیم و بدون اینکه برنامه‌نویس مجبور باشه به صورت صریح بدون نیاز جزئیات ارتباطی، یک رویه رو بر روی یک کامپیوتر دیگه اجرا کنه.

- میان‌افزار میانی از ارتباطات-خام مراقبت خواهد کرد.
- برنامه‌نویس می‌توانه به صورت شفاف (transparent) با یک موجودیت از راه دور ارتباط برقرار کنه که در locality من نیست و ممکنه naming و accessing با من فرق داشته باشه.

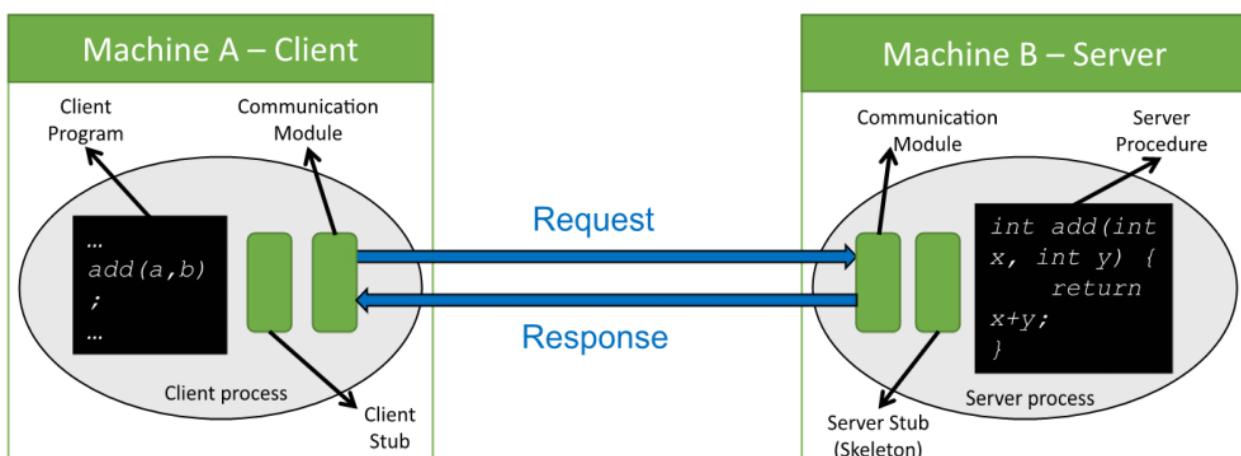
دو نوع میان‌افزار برای **remote transparent communication** وجود داره:

1. (Remote Procedure Call) یا RPC

- RPC این امکان رو به فرستنده میده تا از طریق یه procedure call با گیرنده ارتباط برقرار کنه.
- در این مدل هیچ ارتباط یا message-passing ای برای برنامه‌نویس آشکار نیست (هستش ولی مخفی یا transparent شده).

طبق شکل زیر ما یه تابعی به اسم add داریم که در ماشین B یعنی server پیاده‌سازی شده. حالا فرض کنیم بخوایم این تابع رو تو ماشین A یعنی Client اجرا کنیم. اگر تابع add داخل سیستم A باشه که خودش از تو برنامه یا لاپیراری‌های سیستم این کار رو انجام میده ولی اگر نباشه باید از طریق یه میان‌افزار یا middleware بره سراغ ماشین B تا اونجا این تابع رو فراخوانی کنه و نتیجه رو برگردونه.

مطابق شکل زیر میان‌افزار کلاینت و سرور به ترتیب Server Stub و Client Stub هستش که این دو از طریق یک communication module که در هر دو سمت وجود دارد با یکدیگر ارتباط برقرار می‌کنن.



وظیفه‌ی Client Stub چیست؟

- کدهای کاربر رو مثل یک procedure محلی اجرا می‌کنه.
- پارامترهایی که بهش پاس می‌شه (مثلاً تو شکل بالا a و b که تو تابع add بهش پاس شدن) رو به اصطلاح مارشال یا سریالی‌شون تبدیل به یک message server stub می‌کنه. یعنی می‌دان به request packet می‌گه بین این

- ای (request-pkt) توجه بشه که این بسته IP نیست و صرفا دینتا رو داخل يه بسته کپسوله کرده) که از سمت من میاد پیشت این پارامترها رو داره و ترتیبیشون به این صورته و ...
- یک رویه transport سمت کاربر رو هم فراخوانی میکنه مثل:
 - makerpc(request-pkt, &reply-pkt) - زمانی که پاسخ خودش رو از سمت سرور دریافت کرد میاد به اصطلاح بسته رو unpack یا unmarshal یا de-serialize میکنه به پارامترهای خروجی در انتها هم به user code بر میگردونش.

حالا تو Server Stub چه خبره؟

- درخواست‌ها را از کلاینت‌ها می‌گیره (())getrequest
- آرگومان‌هایی که از سمت کلاینت‌ها دریافت کرده رو unmarshal میکنه، opcode ای که باید ران بشه رو پیدا میکنه و به server code محلی ارسال میکنه.
- پاسخ خودش رو داخل يه سری آرگومان مارشال میکنه و رویه server side transport را فراخوانی میکنه
- پاسخ خودش رو داخل يه سری آرگومان مارشال میکنه و رویه (())sendresponse) و به حلقه‌ی سرور بر میگردونه.
- حلقه اصلی سرور مرسوم:

```
while (1) {
    get-request (&p); /* blocking call */
    execute-request (p); /* demux based on opcode */
}
```

۱. ارسال پارامتر به واسطه‌ی مارشال (Parameter passing via marshaling)

- a. پارامترهای رویه و نتیجه باید روی شبکه به صورت بیت جابه‌جا بشه.
 - b. این ارسال پارامتر به دو صورت قابل انجامه:
- i. Call by Value: مقدار اون پارامتر رو ارسال میکنه (حجم کوچکی از داده به این صورت ارسال میشه نه یک database). اطلاعات کامل راجع به اون پارامتر در اختیار میداره و میشه به صورت مستقیم داخل پیام کد بشه. همچنین تغییراتی که رویه فراخوانده شده (callee) انجام می‌دهد بر روی رویه فراخوان کننده (caller procedure) تاثیر نخواهد داشت.

- ii. مشکل: انواع داده مثل int یا float در هر دو سمت یه جور برداشت میشه یا خیر
- Call by Reference: در این حالت مواردی مثل یک دیتابیس یا حجم دادهای بزرگ رو میخوایم جابه‌جا کنیم، استفاده میشه و وقتی پاس میشه گیرنده ادرس فیزیکی یا مجازی مبدأ رو داره.

۱. مشکل: عدم اعتبار پارامترهای رفرنس در سرور.

a. راه حل: پاس کردن reference parameter با کپی کردن دادهای که

رفرنس بهش داده شده

۲. مشکل: تغییرات پارامترهای رفرنس بر روی کلاینت بی تاثیر خواهد بود. تغییراتی

که در server side انجام میشه لزوما در locality خودمون نداریم.

a. راه حل: "Copy/Restore" داده

i. کپی داده که توسط پارامتر بهش رفرنس داده شده.

ii. Copy-back کردن مقدار در سرور به کلاینت.

۲. بیان داده (**Data Representation**): کامپیوترها در سیستم‌های توزیع‌شده معماری‌ها و سیستم

عامل‌های متفاوتی دارن.

a. اندازه‌ی انواع داده‌ها متفاوته (برای مثال نوع long چهار بایت در سیستم‌های یونیکس ۳۲ بیتی و ۸ بایت

در سیستم‌های یونیکس ۶۴ بیت اشغال میکنه)

b. فرمت داده‌ای ذخیره شده متفاوت است.

i. اینتل داده رو به فرمت big-endian و SPARC به فرمت little-endian انجام میشه.

c. بیان داده باید به صورت یک شکل و uniform باشد و کلاینت و سرور باید برای یک بیان مشترک

ساده داده در پیام‌ها به توافق برسن.

i. البته ممکنه معماری ماشین‌های فرستنده و گیرنده متفاوت باشه.

۳. شکست مستقل (**Failure Independence**)

a. کلاینت و سرور ممکنه به صورت مستقل از هم fail بشن.

b. در حالت محلی، سرور و کلاینت مرگ و زندگی‌شون با همه

c. در حالت ریموت، با انواع مختلف fail شدن ممکنه مواجه بشه.

Network failure .i

Server machine crash .ii

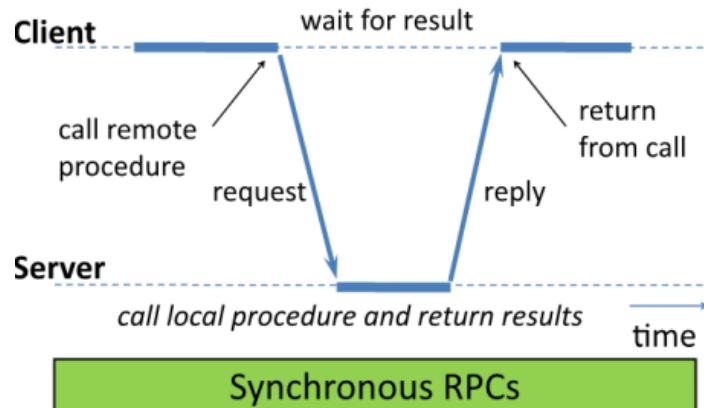
Server process crash .iii

d. بنابراین، کد مدیریت کننده شکست باید کامل‌تر باشد (و لزوما پیچیده‌تر)

RPC انواع

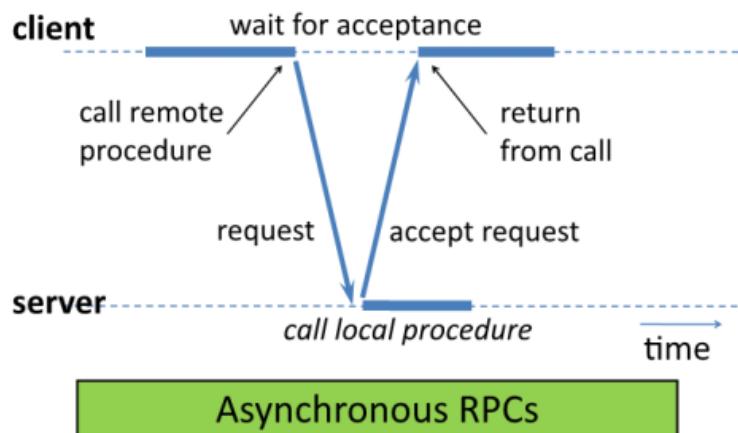
۱) یک RPC با بلوک‌ها محدود request-reply که کلاینت منتظر میمونه تا سرور جواب

بدم. این روش بلاکینگ هم منابع رو هدر میده. اما safety بالایی داره.

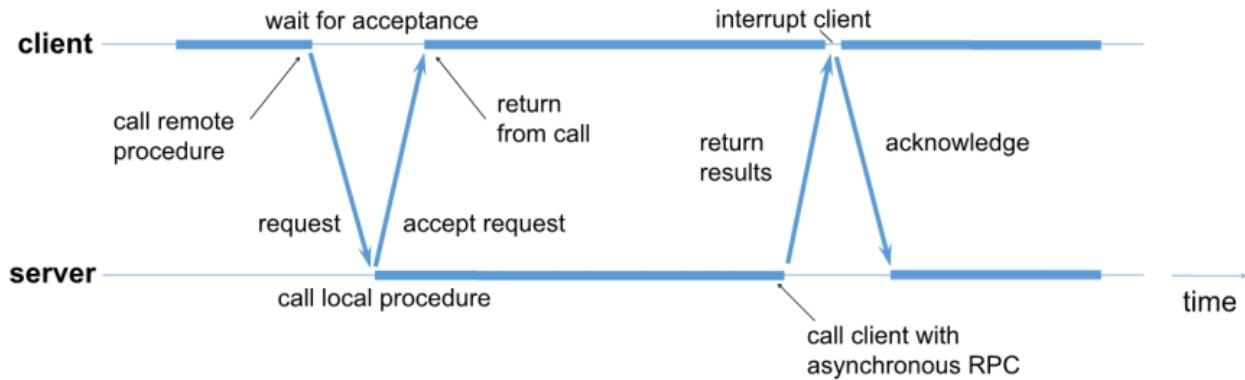


۲) این مدل RPC زمانی به کار برده میشه که کلاینت نیازی به جواب سرور نداشته باشه. مثلا ایمیل یه مثالی از این حالته، چون کارش رو انجام میده و ایمیل رو ارسال میکنه و بعدا نتیجش رو هم میگیره که محموله رسید یا نه. البته مفهومی هم با عنوان **pure asynchronous** هستش که همین مقداری که در شکل پایین صبر کرده رو هم نداره.

- سرور فورا یه ACK رو به سمت کلاینت ارسال میکنه.
- کلاینت هم بعد از دریافت ACK از سرور به اجرای خودش ادامه میده.



Deferred Synchronous: توضیحات تکمیلی در مورد Async RPC زمانی مفیده که کلاینت نتایج رو بخواهد اما نخواهد تا زمانی که **call** تمام بشه بلاک بشه تا منبع هدر بره. کلاینت از این حالت یعنی Deferred sync RPC ها استفاده میکنه. در واقع یک RPC به دو request-response مطابق شکل زیر تقسیم میشه:



وقتی سیستم جواب داد و کلاینت هم ack داد اونوقت سرور میتوانه به کارش ادامه بده.

Remote Method Invocation ۲. رمی (RMI)

این روش (RPC) مشابه است با این تفاوت که در دنیای اشیای توزیع شده به سر میبره. وقتی object پاس کنه در واقع داره object id رو پاس میکنه و این یه جور call by reference هست.

برنامهنویس میتونه از قدرت بیان کامل برنامهنویسی شیگرا استفاده کنه.

RMI نه تنها اجازه pass کردن object reference رو میده بلکه امکان کردن object reference رو هم در اختیار میدارد.

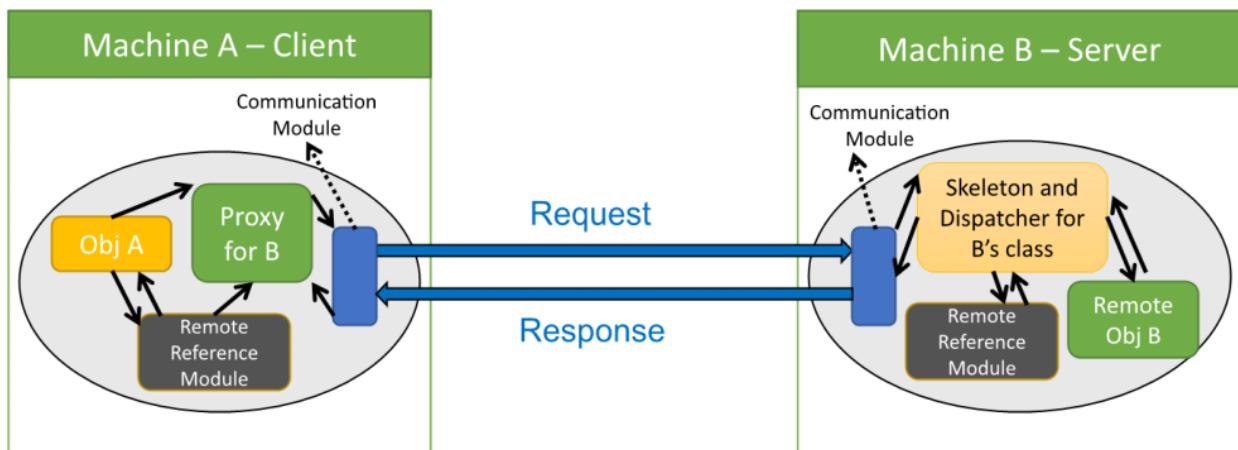
تو RMI، اشیایی که متدهاشون به صورت ریموت فراخوانده میشن به عنوان «remote objects» شناخته میشن.

- ashiyai ریموت remote interface ها رو پیادهسازی میکنن.

- در حین فراخوانی مت، سیستم باید مشخص کنه که اون مت روی یک شی محلی یا ریموت فراخوانی شده.

- فراخوانی محلی باید روی local object ها انجام بشه.

- فراخوانی های ریموت هم متدهای ریموت رو فراخوانی میکنه.



در شکل بالا جریان کنترلی در RMI را مشاهده می‌کنیم. در این شکل obj A را می‌بینیم که میخواهد obj B یا یکی از متدهای obj B را فراخوانی کنه، اگر پراکسی وجود داشته باشه پس این از قبل کد شده است. یه موجودیتی هم به نام **remote reference module** داریم که تصمیم می‌گیره این شی‌ای که از سمت A obj فراخوانی شده محلی هستش یا نه. اگر محلی باشه که به صورت محلی بهش دسترسی پیدا می‌شه و اگر محلی نباشه به obj A می‌گه برو سراغ proxy و از اون به بعد obj A با proxy به صورت مستقیم با پروکسی در ارتباط خواهد بود.

این پروکسی مانند RPC یک communication module داره که در سمت ماشین مقصد هم متضادش یه communication module برای ارتباط دستگاهها وجود داره. این مازول‌های ارتباطی همیگه رو یه جوری می‌شناسن و تو لایه‌های پایین شبکه با هم handshake داشتن و سوکت باز و بسته کردن.

در سمت سرور هم اون مازول ارتباطی با یه موجودیتی به نام skeleton ارتباط می‌گیره و اونم با remote reference module در ارتباط خواهد بود. در انتهای درخواستی که شی A از شی B داشته رو پاس میده به B و اونجا کارش انجام می‌شه و نهایتاً این شی از طریق skeleton و روند قبلی به دست A obj برمی‌گردد.

۷ انتهای جلسه

جلسه ۸

تعريف **Liveness**: یعنی اینکه همونطور که ازش انتظار داریم کار می‌کنه و اتفاق خوبی هم حین کارش می‌وقته، یعنی کار بیهوده نمی‌کنه (working as expected).

مثلاً ما می‌خوایم از اینجا بريم قم و انتظار داریم به سلامت راه رو طی کنیم، پلیس تو راه برای کنترل باشه، چاله و چوله نباشه تو راه، هر چند کیلومتر به جاهایی باشه که بتونیم استراحت کنیم. کامپیوتری بخوایم بگیم در واقع وقتي از مسیر A تا B بخوایم بريم، می‌ایم با دو تا هاپ به راحتی عبور می‌کنیم. در صورتی که ممکنه این مسیر رو از يه جای افتضاح بريم که کلی loss packet و congestion داشته باشه و واقعاً اذیتمون کنه.

حال	به	سراغ	شكل	Flow	Control	RMI	میریم:
-----	----	------	-----	------	---------	-----	--------

فرض کنیم اگر دفعه‌ی بعد obj A از ماشین A سراغ obj B را از ماشین B می‌گیرد، و این شی از این ماشین به ماشین دیگه‌ای migrate یا relocate کرده باشه چه اتفاقی می‌وقته؟

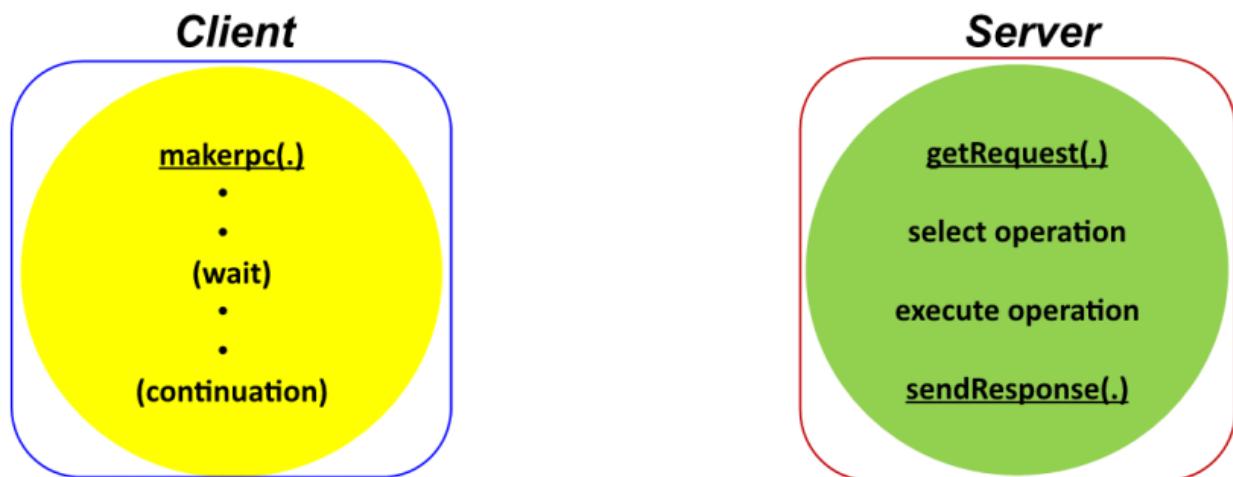
ظاهرا هیچ اتفاقی نباید بیفته و میان‌افزار RMI باید این مسئله رو مدیریت کنه. این روند از A تا communication obj می‌باشد که این نبودن obj B می‌شه. پس بهتره یه registry برای اینکه بدونیم کدام شی کجا قرار داره داشته باشیم که خب فعلاً بهش ورود نکردیم.

Transport primitives

لغت primitive رو با يه مثال توضیح میدیم:
مثلما میگیم نوع داده‌ای های int, float, double و غیره رو من يه کاری کردم که تو سخت‌افزار هم قابل فهم هستش
بهشون میگیم primitive ولی من او مدم به شی تعریف کردم به نام student که اصن این سخت‌افزار من نمیدونه چیه که.

در مازول ارتباطی RPC (بدوی) ارتباطی اصلی به شرح زیر وجود دارد:

- makerpc()
- getRequest()
- sendResponse()



نکته: در زبان‌های برنامه‌نویسی ثابت شده که هر چقدر primitive هاش کمتر باشه خیلی راحت‌تر قابل برنامه‌نویسی کردنه تا اونی که کلی primitive داره و همه رو باید به یاد بسپارم تا بخواه حرکت باهاش بزنم یا جایگشت‌های بکارگیری این primitive ها رو به خاطر داشته باشم.

انواع failure

توضیح	انواع Failure
یه سرور داشتم که کار میکرده و handshake باهاش داشتم ولی به هر دلیل متوقف شد و دیگه کار نکرد.	Crash Failure
• یک سرور در جواب به درخواست‌های ورودی دچار fail میشه.	• Omission Failure

<ul style="list-style-type: none"> ○ سرور در دریافت پیام‌های ورودی دچار fail میشے. ○ سرور در ارسال پیام دچار fail میشے. 	<ul style="list-style-type: none"> ○ Receive Omission ○ Send Omission
<p>ممکنه یه جواب از سمت سرور در یه زمان مشخص برای من قابل قبول باشه ولی سرور به هر دلیلی دچار مشکل شده و این پیام رو سر تایم نفرستاده که دیگه به درد منم نمیخوره</p>	<p>Timing Failure</p>
<ul style="list-style-type: none"> ● سرور جواب نادرست ارسال کرده. ○ مقدار جواب نادرست بوده. ○ سرور از مسیر جریان کنترلی درست منحرف شده. 	<ul style="list-style-type: none"> ● Response failure <ul style="list-style-type: none"> ○ Value failure ○ State Transition failure
<p>سرور ممکنه در زمان‌های دلخواه پاسخ‌های دلخواه تولید کنه (مورد اعتماد نیست).</p>	<p>Byzantine failure (بدترین نوع)</p>

مکانیسم‌های Timeout

برای موقعیت‌هایی که یه سری پیام request یا reply گم میشن، makerpc (.) میتونه از این مکانیسم‌های timeout استفاده میکنه. راهکارهای مختلفی که makerpc بعد از timeout میتونه بکنه:

- یه درخواست فورا به کلاینت برگردونه که «بین پیامت timeout خوردش.»
- خودم به عنوان میان‌افزار درخواست رو باز ارسال بکنم تا زمانی که یا reply از سمت سرور بیاد یا سرور فرض کنه که fail رخ داده.

نحوه انتخاب مقدار timeout به چه صورتیه؟

- در بهترین حالت، بیام از آمارهای تجربی یا تئوری استفاده کنم.
- در بدترین حالت، هیچ مقدار مناسبی وجود نداشته باش.

اگر در مکانیسم‌های timeout خود میان‌افزار اقدام به باز ارسال بکنه اونوقت با چالشی به نام **Idempotent side effect** داشته باشند. دوست داریم **Operations** روبه‌رو خواهیم شد. هیچ قابلیت idempotent operation بودن رو بدون هیچ effect داشته باشند.

هشدار: هیچ عملیاتی که بتونه بیش از یکبار اجرا بشه و هر بار هم نتیجه یکسان بده وجود نداره.

Idempotent Operations: اگر پیام درخواست باز ارسال بشه، سرور ممکنه بیشتر از یکبار دریافتش کنه. در واقع side effect same نداشته ولی effect داشته باشن.

به منظور جلوگیری از **Duplication** سرور باید:

- پیام‌های موقتی‌آمیز از کلاینت‌های «بکسان» رو شناسایی کنه.
- به صورت پکنواخت seq number رو افزایش بده.
- تکراری‌ها رو فیلتر میکنه.

هنگام دریافت یک درخواست «تکراری»، سرور میتونه:

- عملیات رو دوباره اجرا و reply بزن.
- فقط برای عملیات‌های idempotent ممکنه.
- یا از باز اجرای دوباره‌ی عملیات با نگهداری خروجی در یک فایل log غیر فرار و ماندگار جلوگیری کنه.
- ممکنه transactional semantics ضروری باشه.

انتخاب‌های پیاده‌سازی

RPC transport میتونه به روش‌های مختلف به منظور تامین روش‌های مختلف delivery پیاده‌سازی بشه. انتخاب اصلی شامل:

- (۱) **سمت کلاینت** retry request service) - : کنترل میکنه که آیا سرویس درخواستی را تا زمانی که پاسخی دریافت بشه یا سرور از کار افتاده باشه، مجدداً منتقل میکنه.
- (۲) **سمت سرور** (duplicate filtering) - : هنگام استفاده از انتقال مجدد و فیلتر کردن درخواست‌های تکراری تو سرور رو کنترل میکنه.
- (۳) **سمت سرور** (حفظ نتایج) - : کنترل میکنه که آیا سابقه پیام‌های نتیجه رو نگه میداره تا بتونه پاسخ‌های از دست رفته رو بدون اجرای مجدد عملیات در سرور مجدد ارسال کنه.

RPC Call Semantics

Call Semantics	معیارهای Fault Tolerance		
	Retransmit Request	Duplicate filtering	Re-executing

	Message		procedure or retransmit reply
احتمالاً تعریف نشده	نداره	نداره	نداره
حافظ یک بار (۱ یا بیشتر)	داره	نداره	Re-execute Procedure
حافظ یک بار (۰ یا ۱)	داره	داره	Retransmit Reply

نکته: در حالت ایدهآل ما حالت دقیقاً یک بار semantic را میخوایم.

اینکه من کدوم قسمت‌ها را توی میان‌افزار و کدوم را توی برنامه مدیریت کنه بستگی داره که UDP روی RPC بنا شده باشه یا TCP.

(۱) اگر RPC روی UDP بالا اومده باشه:

- باز ارسال باید / میتوانه توسط RPC مدیریت بشه.

(۲) اگر RPC روی TCP بالا اومده باشه:

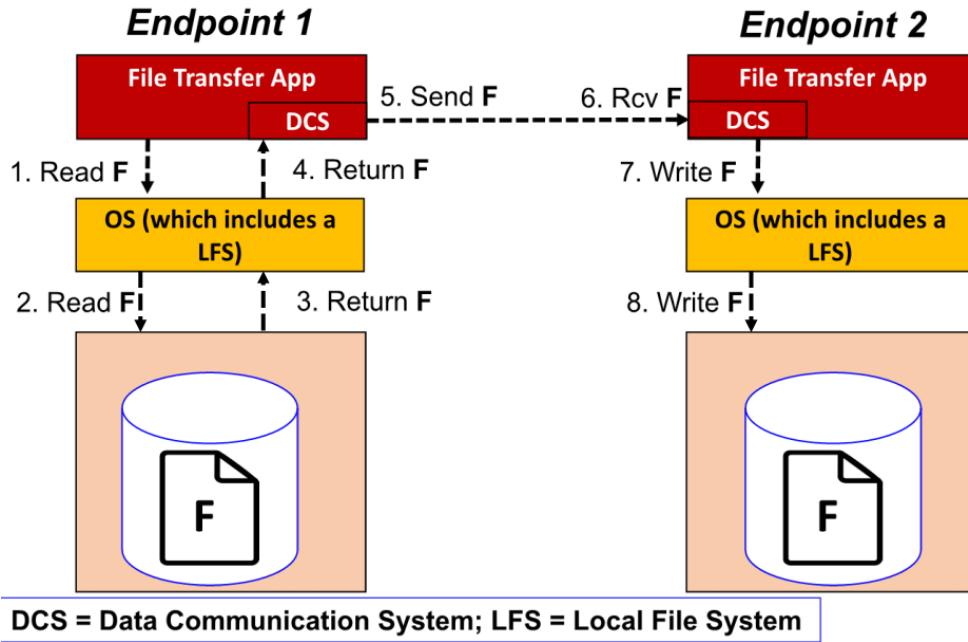
- باز ارسال می‌توانه توسط TCP مدیریت بشه.

البته بازم یه سری کارها رو خود RPC باید انجام بد. آیا ضروریه که رسیدگی به معیارهای fault-tolerance ●

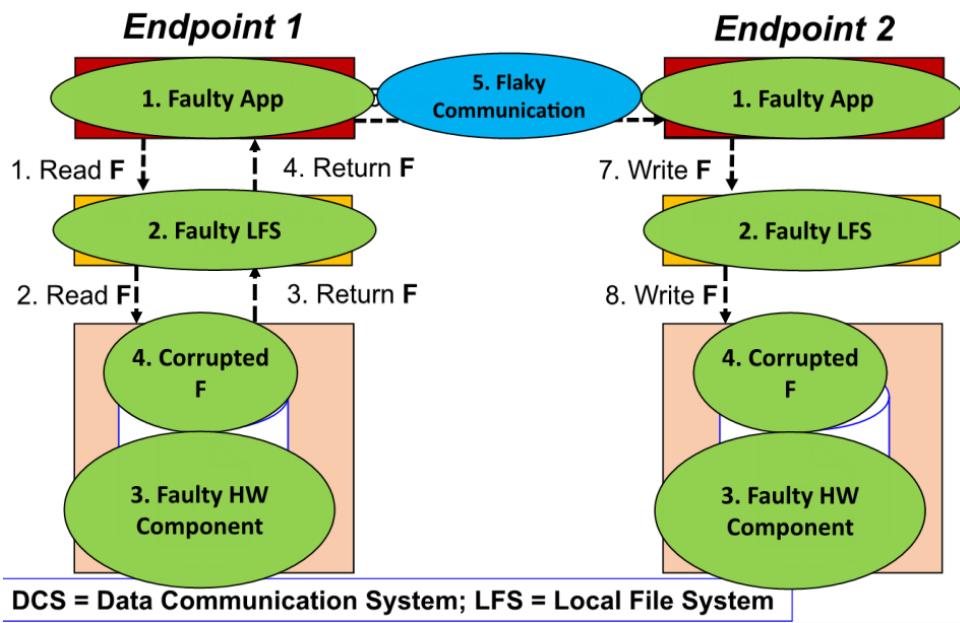
داخل RPC صورت بگیره؟

- بله

شكل پایین مثال خوبی از توضیحات هستش. فرض کنیم دو تا endpoint میخوان با هم ارتباط برقرار کنن و هر کدوم هم به یه همون communication module DCS دارن. حال endpoint1 میاد درخواست یه فایل رو به OS میده که اونم از LFS درخواست کنه که بگیره و برای endpoint2 ارسال کنه. اونم تو DCS خودش دریافت میکنه مینویسه توی OS و سیستم عامل هم روی LFS خودش مینویسه.



ولی اگر همه‌چی انقدر خوب باشد که عالیه. مشکل اینجاست که همه‌چی به این خوبی نیست و ممکنه هر جزوی از ما دچار **fault** بشه. توجه بشه که برنامه من داره درست کار میکنه درباره‌ی حالاتی صحبت می‌کنیم که ممکنه خطاهایی خارج از برنامه برای سیستم رخ بده.



حالا با چه سازوکاری میتوانیم جلوی این fail‌ها رو بگیریم:

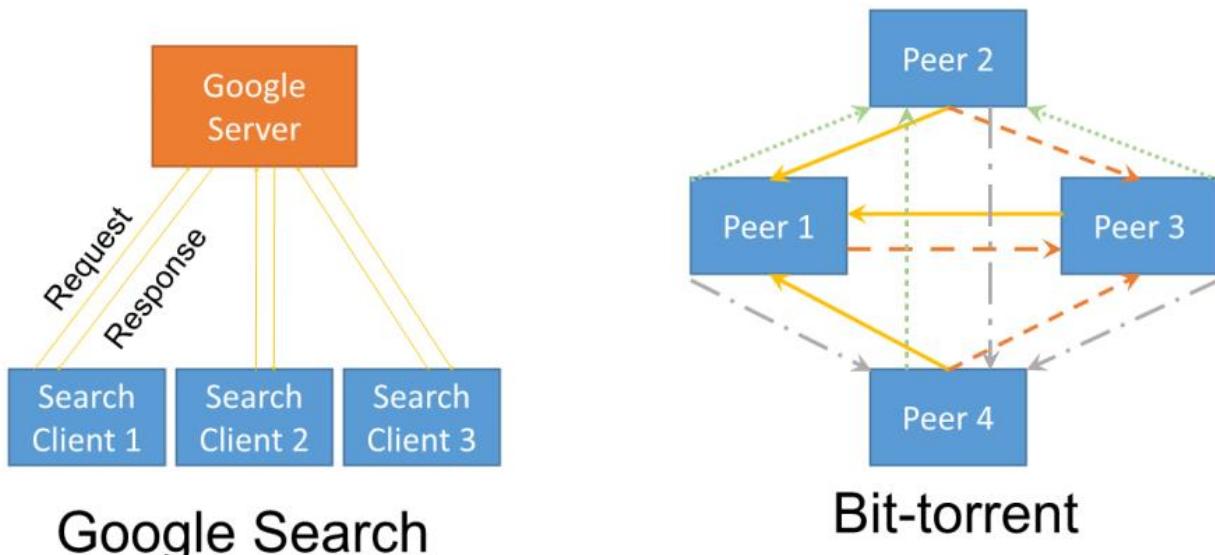
- در این حالت endpoint1 میتوانه یک checksum از اون فایل F رو بگیره و با نام Ca ذخیره میکنه.
- بعد از اینکه endpoint2 فایل رو دریافت کرد، میاد فایل رو مجدد از حافظه خودش میخونه و checksum ش رو محاسبه میکنه و برای endpoint1 ارسال میکنه (Cb).

- بعد از Ca و Cb endpoint میاد رو مقایسه میکنه:
 - اگر $Ca=Cb$ بودش انتقال رو commit میکنه.
 - در غیر این صورت دوباره ارسال میکنه. چند بار این ارسال میشه؟
 - معمولش یکباره ولی تا ۳ بار نشون میده که یه قسمتی از سیستم نیاز به تعمیر داره.
- چی میشه اگر DCS از TCP استفاده کنه؟
 - فقط تهدیدهای packet loss به جهت نامناسب بودن ارتباطات ممکنه برطرف بشه.
 - اگر علت fault سیستم ارتباطی باشه، فرکانس تلاشها کاهش پیدا میکنه.
 - کنترل ترافیک بیشتر خواهیم داشت و قسمت های گمشدهی F نیاز به باز ارسال خواهد داشت.
 - ولی برنامه file transfer هنوز به معیار های اطمینان پذیری انتهاء انتها نیاز داره!
- اگر UDP از DCS استفاده کنه چی؟
 - تهدیدهای ارتباطی برخلاف TCP به قوت خودشون باقین. در صورت عدم اقدام برای رفع این تهدید، برنامه باید F را مجدد باز ارسال کنه.
 - فرکانس تلاشها افزایش پیدا میکنه.
 - بدترین کارایی روی ارتباطات خراب
 - بازم برنامه file transfer به یه سری معیار های قابلیت اطمینان end-to-end نیاز خواهد داشت.

پایان جلسه ۸

جلسه ۹

تعريف معماری در علوم کامپیوتر: اجزا و ارتباط ساختاری بین شون.



در شکل بالا دو معماری مختلف برخی سیستم‌های توزیع شده رو مشاهده می‌کنیم. در معماری سمت چپ ارتباط میان مونتور جستجوی گوگل (فارغ از پیچیدگی‌های درونش) با کلاینت‌های (مثل سیستم رزرواسیون Expedia). در سمت راست هم معماری شبکه نظیر به نظیر Bit-torrent (رو مشاهده می‌کنیم و اینکه هیچ تمرکزی وجود ندارد و هر peer به صورت مستقیم می‌تواند با باقی ارتباط برقرار نماید (مثل برنامه skype).

مشخصه‌های ساده‌ی سیستم‌های توزیع شده

چه موجودیت‌هایی در سیستم‌های توزیع شده با یکدیگر ارتباط دارند؟

- موجودیت‌های ارتباطی (مبتنی بر سیستم (مثل پروسس و ترد و ...) و مبتنی بر مسئله (مثل شی یک runtime سیستم سطح بالاتر))

این موجودیت‌ها چطوری ارتباط برقرار می‌کنند؟

- پارادایم‌های ارتباطی (مثل سوکت و RPC - بعدا با پارادایم‌های بیشتری آشنا خواهیم شد)

موجودیت‌ها چه نقش‌ها و مسئولیت‌هایی دارند؟

- سازماندهی‌های مختلفی برای این نقش‌ها وجود دارد.

معماری‌ها

دو معماری اصلی عبارتند از:

• معماری master-slave

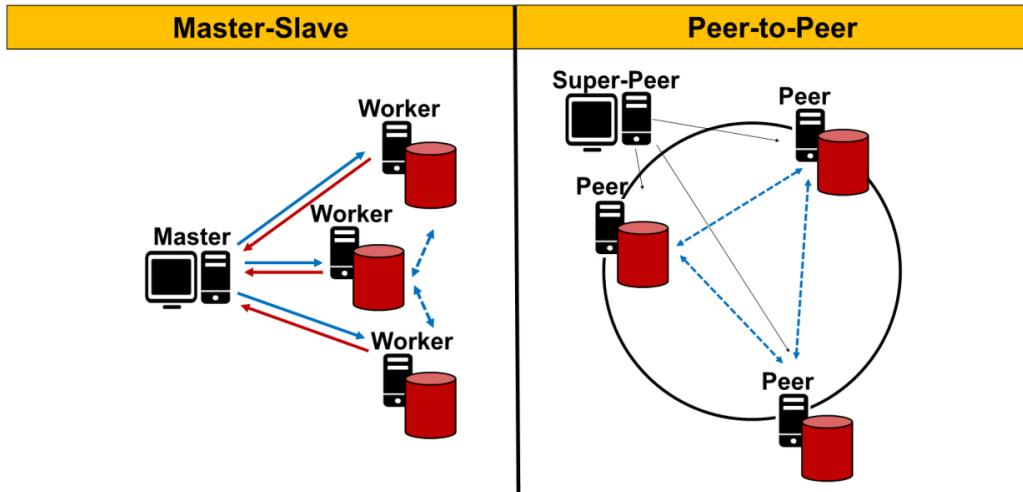
- نقش‌های موجودیت‌ها نامتقارنه. بعضی به نقش خاص دارن که بعضی‌ها ذاتاً نمی‌تونن اونا رو بازی کنن.
(سلسله مراتب وجود داره)

■ در برابر SPOF آسیب‌پذیر هستند.

- گره master به عنوان یک coordinator مرکزی نقش ایفا می‌کنه.
■ تصمیم‌گیری راحت خواهد بود.
- به راحتی نمی‌شه به سیستم زیرین گره اضافه کرد (یا به اصطلاح scale out کردش).
- با افزایش تعداد workerها گره master دچار یک performance bottleneck خواهد شد.

• معماری همتا-به-همتا (p2p)

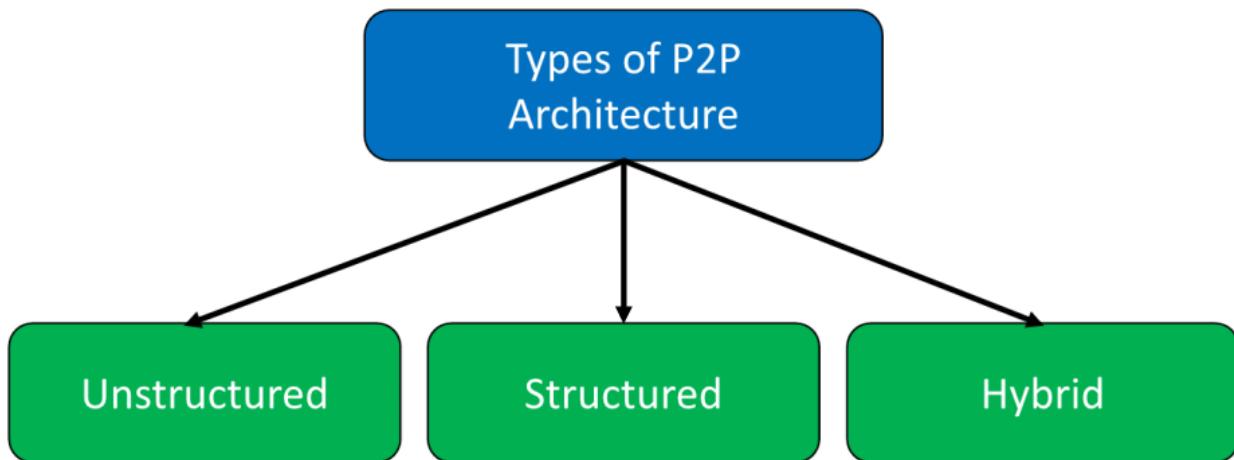
- نقش‌های موجودیت‌ها متقارنه. یعنی هر موجودیت مقام یکسانی نسبت به دیگری داره (سلسله مراتبی نیست).
- دیگه SPOF نداریم.
- نیازی به coordinator مرکزی نیست
 - ولی خب تصمیم‌گیری سختتر میشه.
- سیستم زیرین رو میشه scale out کرد چون تمرکزی نداریم و مشکلی نیست.
 - در نتیجه performance bottleneck نخواهم داشت.
- هم‌تاهای می‌تونن با هم به صورت مستقیم ارتباط برقرار کنن، تشکیل گروه بدن و محظا به اشتراک بذارن (یا به هم‌دیگه سرویس پیشنهاد بدن).
 - حداقل یک هم‌تا باید دیتا رو به اشتراک بذاره و این هم‌تا باید در دسترس باشه (اینجا چون گفتن حداقل یک هم‌تا در واقع بازم همون SPOF ممکنه پیش بیاد ولی خب میتوانیم فراتر از یک هم‌تا تو شبکه داشته باشیم که این مشکل پیش نیاد).
 - داده‌ها یا هر موجودیت مشهوری highly available خواهند شد (چون توسط خیلی‌ها به اشتراک گذاشته خواهد شد).
 - و دیتا‌های غیرمعروف ممکنه در نهایت ناپدید یا غیرقابل دسترس بشه (چون کاربرای بیشتری دیگه به اشتراک نمی‌ذارند).
- هم‌تاهای میتوانن بالای یک توپولوژی شبکه فیزیکی یک شبکه پوششی (overlay) مجازی را تشکیل میدن. شبیه simulation هست.
 - مسیرهای منطقی معمولاً به مسیرهای فیزیکی match نمیشن (تأخیر بالا به همراه خواهد داشت). مثلاً ممکنه تو اون شبکه همپوشان مجازی گره‌ها تو یه شبکه باشن ولی از لحاظ فیزیکی یکی‌شون توی کابله و یکی هم توی آمریکا به همین دلیل تأخیر زیادی ممکنه به همراه داشته باشه.
 - هر هم‌تا نقشی رو در ترافیک روتینگ در شبکه overlay ایفا میکنه.



نکته: در معماری بالا تنها فرق با توضیحات وجود یک Super Peer هستش که معماری رو مثل master-slave نمیکنه ولی شانیت peer رو هم نداره و یه چیز بینابینی هستش.

تفاوت emulator و simulator: یه وقتی به یکی میگیم $2 + 2$ چند میشه، مثل این بچهها که کلی روش و تکنیک تو ذهنشون برای رسیدن به جواب انجام میدن بهش برسن، این میشه simulator تحلیل هاش زیاد قابل انتکا نیستند. یه وقت هست میریم توی سختافزار یه چیزی رو با جزئیات میسازیم و به جواب میرسیم که این میشه emulator. فرض کنیم ما میخوایم کامپیوتر کوانتمی بسازیم و سیستم عاملش رو هم بنویسیم. منتظر نمیمونیم تا یکی بعد سال‌ها سختافزارش رو با مشقت بسازه. میریم سراغ یک API که مبتنی بر کامپیوترهای کوانتمی و Qbit ها هستش و از اون برای ساخت سیستم استفاده میکنم و به جزئیات هم دقت میکنیم (این میشه emulator).

P2P انواع



۱. **Unstructured P2P**: هیچ ساختاری خاصی رو تو این شبکه overlay تحمیل نمیکنه.

- a. مزایا:
 - i. راحت ساخته میشه
 - ii. در برابر نرخ بالای Churn مقاومه (یعنی اگر تعدادی زیادی از کاربرها به صورت متناسب متصل و منفصل بشن دچار مشکل نمیشه و دچار افت کارایی نمی شیم.)
- b. عیب اصلی:
 - i. همتاها و محتواها به هم وابستگی محکمی ندارن و به اصطلاح loosely-coupled هستن.
 - 1. سرچ کردن دیتاها سخته و ممکنه به broadcasting نیاز داشته باشه و سرباز زیادی داره چون حواس همه نودها پرت میشه و باید مدام گوش بدن.

۲. **Structured P2P**: معماری بعضی ساختارها رو بر روی توپولوژی overlay قبول میکنه.

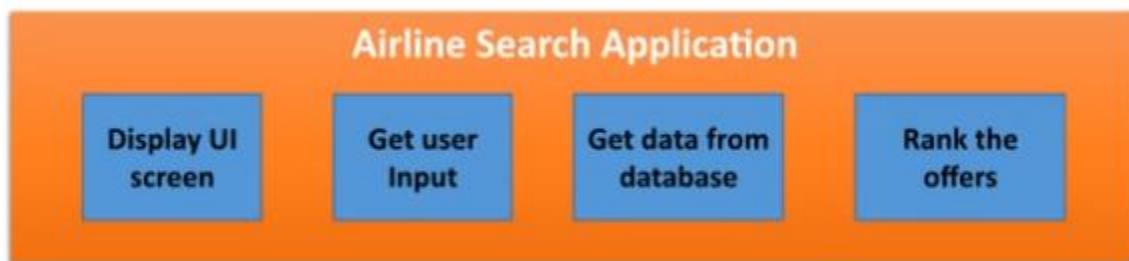
- a. مزایا:
 - i. همتاها و محتواها به هم وابستگی قوی دارن (tightly-coupled).
- b. معایب:
 - i. ساختنش سختتره
 - ii. برای data location های بهینه شده باید متادیتای اضافی نگهداری بشه (لیستی از همسایهها که یک شاخص خاص رو ارضا میکن) و space بیشتری باید صرف بکنیم.
 - iii. قدرتمندی کمی رو در برابر نرخ بالای churn رو داره.

۳. **Hybrid P2P**: این معماری برای برخی سرورهای مرکزی برای کمک کردن به peer ها به منظور پیدا کردن همیگه کمک کنه

- a. یه تلفیقی از P2P و مدل‌های master-slave هستش. یعنی مصالحه‌ای بین عملکرد مرکز مهیا شده توسط مدل master-slave و برابری گره‌ها که توسط مدل‌های P2P پیشنهاد می‌شوند رو برقرار کنند.
- i. به عبارت دیگه مزایای P2P و master-slave رو می‌باره و سط ولی معایب‌شون رو کنار میدارند.

الگوهای معماری

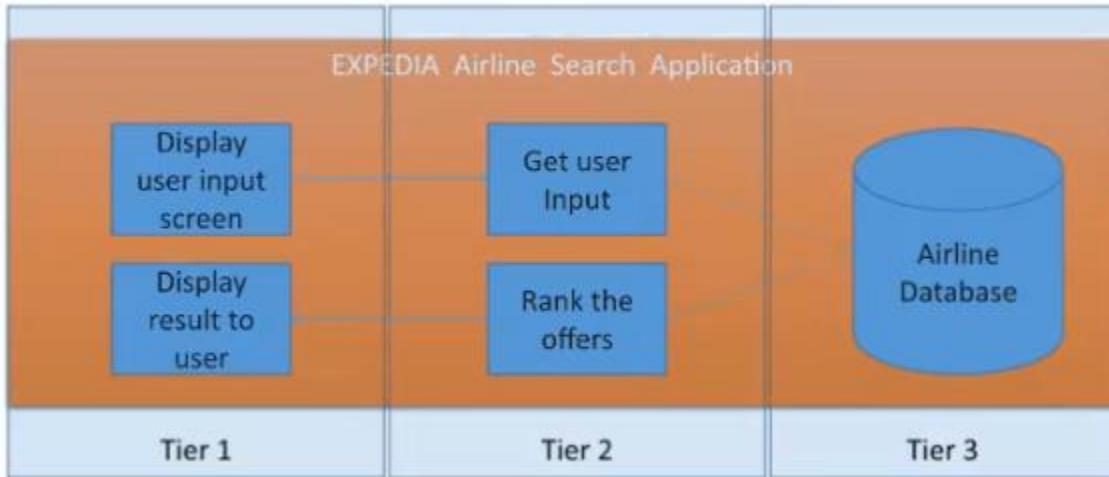
جدا از معماری، عناصر معماری بدوى را می‌توان با هم ترکیب کرد و الگوهای مختلفی را از طریق: **Tiering** • سرویس‌ها و خدمات و functionality ها رو به صورت افقی بشکونم و در اختیار ملت بذارم.



به جای این که یک application را monotonic و یک‌به‌یک نگاه کنم، break down کنم و افقی کنار هم در نظرش بگیرم. برای مثال در این application می‌خواه ۴ تا کار بکنم. بخش اول ارتباط با UI هست که روی هر دستگاهی بشه اجرا بشه. بخش دوم این است که روی هر دستگاهی بتونه ورودی بگیره و خروجی بده. بخش بعدی بر اساس داده‌های ورودی و منطق برنامه داده‌ها رو manipulate کنم. بخش آخر هم خروجی هارو rank کنم.



میشه بخش display کردن ورودی و خروجی رو کرد یک tier و باقی بخش ها رو کرد tier دوم. این دو میتونه هر کدام روی یک سرور باشه. اگر کسی tier اول رو هک کرد احتمالش ضعیفه بتونه tier بعدی رو هم بتونه هک کنه نسبت به حالتی که tier بندی نکرده باشیم.



اگر بطلبه، گرفتن ورودی از کاربر و فرآوری خروجی به شکل مورد نیاز بهره بردار رو هم در tier جدایی قرار داد. با این کار database ام که بخش مهمی از سیستمه به عنوان **Data Logic** رو داخل mainframe قرار بدم. بخش **Presentation** رو هم داخل یک سرور بهتر قرار بدیم و در نهایت **Application Logic** رو میتونم توی یک سرور داغون قرار بدم. این ۳ تا tier یکسری منافع برآش وجود دارد و معماری **Logic** در نظر بگیریم و ۳ بخش رو از هم جدا کنم.

- مزایا:

- هر کدام رو جدا جدا میتونم maintain کنم و نیازی نباشه به بخش های دیگر دست بزنم. مثلا فقط در بخش presentation logic صفحات رو مبتنی بر هر دستگاه بهره بردار از device responsive بیام ای خودش رو adjust بکنه.
- Single point of failure

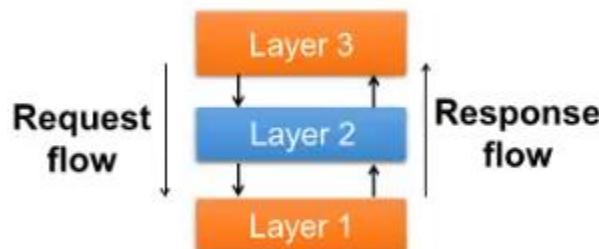
- معایب:

- این تقسیم بندی ها میتونه پیچیدگی ایجاد کنه بابت مدیریت چندین سرور و روی کارایی هم تاثیر منفی بازه
- افزایش ترافیک شبکه به واسطه ارتباط بین چند سرور که هر کدام یک tier هستند
- افزایش تاخیر به واسطه انفصل بین tier ها
- Multi point of failure دارم (هر کدام از این قسمت ها میتوانه fail بکنه و این هم خوبه و هم بد) (میتونم ارزون تر به دست بیارم، فضا و پاور کمتری مصرف کنم اما ایرادش cost میباشد)
- over performance هست ما به ازای چه هزینه ای چه کارایی نیاز داریم و باید مصالحه کنیم تا به تصمیم مناسب برسیم).

مثال: در هوانوردی برای جا به جایی مسافر دسته بندی های مختلفی انجام شده مثلا یک airbus میاد ۷۰۰ نفرو منتقل میکنه و بلیت‌ش گرونه، خود هواپیما هم گرونه و تعدادش در جهان کمه و سامانه هاش هم فوق العاده پیچیدس. خوبیش اینه که ۷۰۰ نفرو یکجا جا به جا میکنه. ولی به درد جاهای دور میخوره.

حالا اگر ۷۰۰ نفرو به ۷ تا گروه ۱۰۰ نفره تقسیم کنیم و با هواپیماهای ۱۰۰ نفره بفرستیم کمتر میشه safety بیشتر میشه چون در حالت قبل اتفاقی بیوقته ۷۰۰ نفر تلفات دادیم. یعنی **single point of failure** به **multiple** تبدیل شد اینکه هر ۷ تا هواپیما دچار سانحه بشن احتمالش ضعیفه. اما این نکته هم هست که احتمال **failure** هر کدام از این ۷ تا خیلی بیشتر از اون ۱ هواپیمای قوی هست.

Servicing: سرویس‌ها رو به صورت عمودی بشکنم و در اختیار ملت بذارم. لایه لایه سرویس‌ها روی هم قرار گرفتند. در **tier** جایگشت افقی بود چون هم ارز همیگر بودند اما اینجا جایگشت عناصر لایه لایه هست.



به لایه‌های پایین platform میگن که متشکل از سخت افزار و سیستم عامل هست. لایه میان افزار بین application و application هاست. هر کدام از این لایه‌ها قابلیت اینو داره که ریزتر بشه مثل بحث‌های SaaS و .cloud computing و .. در PaaS



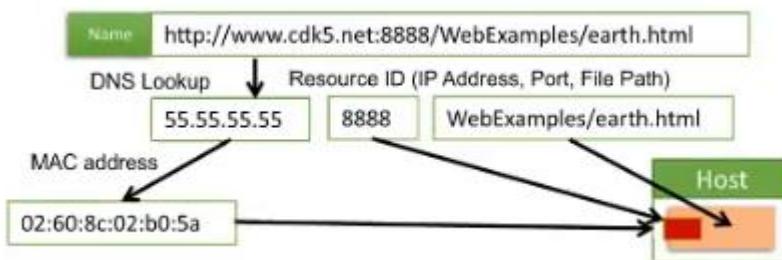
در cloud به قسمت platform این شکل میگن IaaS و به میان افزار میگن Platform as a Service و به app‌ها میگن SaaS. این دو الگو مکمل همیگه هستن.

Naming:

اسم گذاری برای address کردن موجودیت ها در سیستم های توزیع شده یا تک هست و در سیستم توزیعی چالش برانگیزتر هست.

موجودیت میتوانه پردازه، ریسمان، شیء، newsgroup یا فایل یا هر چیزی باشد.

از رهگذر نام گذاری بتنم محلیت جغرافیایی موجودیت را پیدا کنم. **Name resolution** : در کاربردهای وب بخوایم فایل f.html را resource id address کنیم باشد از id استفاده کنیم که از آدرس ip و port یا مسیر مورد نظر اون فایل تشکیل شده است. خود ip address آدرس منطقی هست و باید map بشه به یک unique MAC address که یک آدرس فیزیکی برای اون ماشینی هست که میزبان اون منبع من یعنی اون فایل هست.



۳ نوع نام گذاری را باید از هم تفکیک کنیم:

NAME . ۱

- یک sequence ای از یک سری کار اکثر هست که میتوانه human friendly هم باشد. (مثال: محمد، علی و ...)
- میتوانه semantic داشته باشه یا نداشته باشه. (محمد: مسلمان و مذکور هست اما نمیتوانیم بفهمیم مجرد یا متاهل)

Address . ۲

- Access point ای هست که میخوایم بهش متصل بشیم (مثال: آدرس خونه موجودیت: محله فلان، خیابون فلان، کوچه، پلاک، طبقه، واحد)

برای ما access point (Address) فراهم میکنے اما Name نه، چون بگیم ممد ۱۰۰ تا ممد

(پیدا میشه)

- آدرس ممکن است location و باشہ یا نباشہ

$$\text{Address} = \text{IP Address} + \text{Port}$$

Identifier . ۳

- منحصر به فرد و unique موجودیت هارو مشخص میکنه. (کدلی میتونه به عنوان identifier در گستره جغرافیایی ایران عمل کنه ولی برای گستره بزرگ تر باید passport number داشت تا به صورت انحصاری مشخص شد).
- یک identifier معمولاً ۳ تا خاصیت داره:
 - هر هویت فقط و فقط ارجاع میدهد به یک موجودیت (at-most one entity)
 - هر موجودیت فقط و فقط حداقل توسط یک identifier میشه بهش ارجاع داده بشه (most one identifier)
 - هر identifier همیشه به یک موجودیت refer میکنه به عبارت دیگه reuse identifier ها نمیشن. (کدلی کسی که فوت کنه رو به کس دیگه ای نمین)

Naming system:

در سیستم های توسعی نامگذاری رو به عنوان یک میان افزار می بینیم. چون domain ها مستقل هستند ownership های مستقل دارند و هر کسی میتوانه قواعد نامگذاری خودش رو داشته باشے. نیاز به یک API مشترک هست که موجودیت های accessible بشن.

سامانه های نامگذاری ۳ دسته هستند:

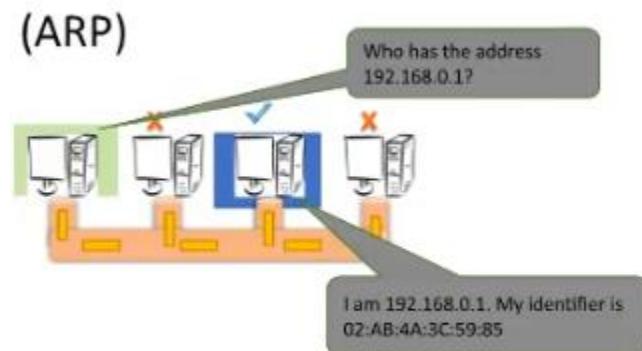
Flat naming . ۱

- flat یعنی یک random bit substring هست و unique هم نیست
- این اسمی اطلاعاتی در مورد locality یا موقعیت جغرافیایی موجودیت ها در اختیار نمیگذارند.
- ۴ تا روش برای resoul کردن موجودیت ها از طریق flat name ها میتوانه وجود داشته باشے:

Broadcasting ○

- ارسال name و address به تمامی شبکه همه پخشی میشه و هر کسی که اون آدرس نام رو داره یا اون آدرس بهش مربوط میشه باید دست بالا کنه و جواب بده.

مثال: برای تبدیل ip address به MAC address یه پروتکل به نام Address Resolution Protocol = ARP وجود داره. میاد ip address را همه پخشی میکنه اون دستگاهی که mac address اش معادل این هست جواب میده. عملاً این میشه همون identifier name که گفته شد. (ای که داشتیم نمیتوانست به من location بده)



■ چالش های broadcasting :

- Scalable resolution نیست و همه درگیر میشن
- سربار زیاد هست و شبکه flooded میشه و تاخیر شبکه بالا میره
- همه موجودیت ها ها باید گوش بدند مدام و این خیلی سربار دارد و مدام متابع باید صرف بشه.

○ Forwarding pointers

■ این روش بیشتر به درد موجودیت های موبایل و متحرک میخورد که access point آن ها مدام فرق میکنه و باید به صورت forward pointer جا پاهاشو نگه داریم.

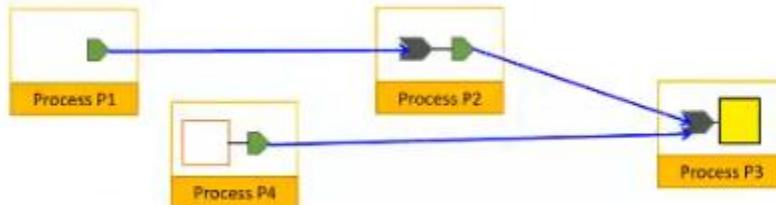
■ Name resolution mechanism

- باید یک chain of pointer رو برای دسترسی به یک موجودیت follow کرد.

■ و با یافتن location جدید موجودیت این ارجاع به موجودیت باید به روزرسانی بشه

■ چالش های این روش: chain طولانی باشه به علت تحرک زیاد موجودیت، باعث تاخیر در دسترسی و resolution میشه و ممکنه لینک ها break down بشه و نتوانیم دسترسی داشته باشیم.

مثال: پروتکل process P Stub-Scion Pair = SSP میخواهد process P بزنه به یک جایی دیگر، حالا اگر object یا ما متحرک باشند و به یک ماشین دیگر رفته باشند. اینجا باید خود یک server Stub بزنه به ماشین جدید. وقتی جای طرف معلوم بشه میشه از اون به بعد خودشون با اون مکان مقصد ارتباط بگیرند.



Home-based approaches

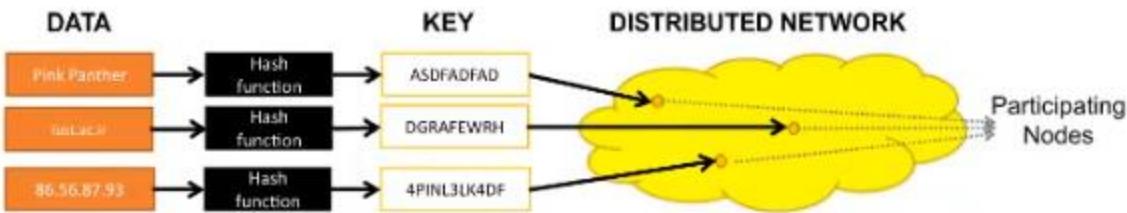
- معمولاً برای multiprocessor ها و سیستم هایی که در یک locality استفاده میشند.
- برای هر موجودیت یک home یا خونه در نظر میگیرند که fix هست (fix access point and address).
- یک current address هم برای هر موجودیت به صورت به روز در اختیار داشته باشیم.
- برای این کار به یک naming server نیاز داریم که آدرس home هر موجودیت در اون ثبت شود.
- اگر موجودیت حرکت کرد اون home ببیاد و current address رو داشته باشه و به هر کس که با این موجودیت کار داره اون آدرس رو بده و مدام هم اون آدرس رو به روز کنه (mobile IP مثل پروتکل



مشکل این روش: ممکن که کسانی که به موجودیت متحرک نیاز دارن نزدیک موجودیت باشند و با Home node فاصله زیادی داشته باشند و برای آدرس گرفتن مجبوران به این گره متصل بشن و سربار شبکه بیخود و بجهت زیاد بشه.

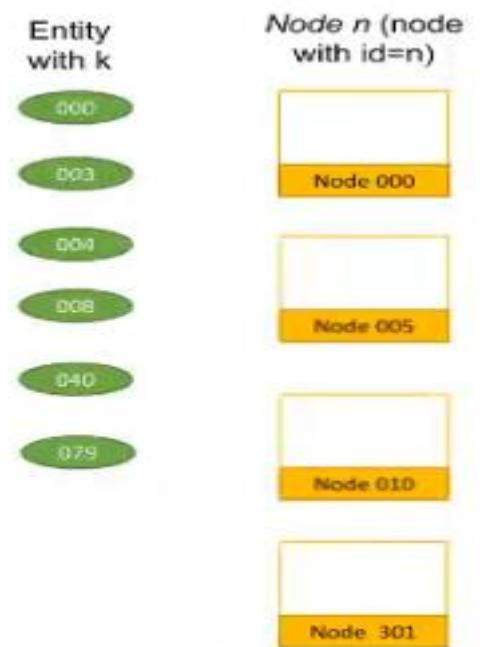
Distributed Hash Tables = DHTs ○

ما علاوه یک Name service (یک نام می‌گیریم و آدرس را مشخص می‌کنیم) داده‌هایی که دنبالش هستیم را hash می‌کنیم و تبدیل به یک سری کلید می‌کنیم و کلیدها رو در شبکه قرار میدیم و می‌گیم آقا value متاظر این کلیدها رو به من بده و یکسری node هستند که cooperate می‌کنن که value هارو به ما بدن. هر node ای که عضو این مجموعه همکار هست جفت key, value در آن ثبت و به روز شده وجود داره و این mapping بین key و value در تمامی این گره ها وجود دارد و ما علاوه یکدونه name service نداریم و به صورت توزیع شده در شبکه پخش هست.



■ مکانیسم های مختلفی مبتنی بر DHT ساخته شده یکی از معروف ترین ها هست.

- به هر node یک هویت می دهد که random انتخاب شده مثلاً id اش می شود میتوینم رو به network address node n عنوان این id در نظر بگیریم.
- در مرحله بعدی هر موجودیت رو map میکنیم به یک node و اگر تعداد زیادی موجودیت داشته باشیم بین n تا node که دارم اون ها رو توزیع می کنم.
- با چه الگوریتمی این توزیع رو انجام میدن؟ اگر id یعنی همون m-bit identifier در node مساوی K (هر موجودیت K دارد) بود موجودیت را بندارش داخل همون node اگر نبود بندارش داخل successor اش. (یعنی K که از شماره خونه بیشتر شد بره خونه بعدی)
- قبل از عمل:

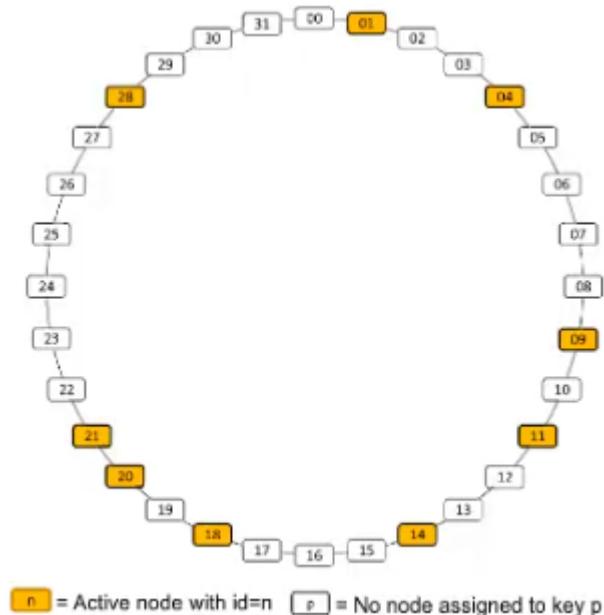


- بعد از عمل : $n = K$ دا خل node و لی $K = 5$ دا خل با id n رفت و لی $n = 0$ رفتد و به همین ترتیب.



توضیح بیشتر این بخش با شکل های زیر امکان پذیر است:

A NaiveKey Resolution Algorithm:



کل گره هایی که درگیر name resolution هستند در این loop قرار دارند و گره هایی که با رنگی مشخص شده اند فعال هستند و بی رنگ ها، هیچ موجودیت ای بهشون تخصیص داده نشده. برای به دست آوردن آدرس موجودیت هامون نیاز به دونستان succ و pred هست و چون بعضی node ها این وسط فعال نیستند و موجودیتی درشون نیست

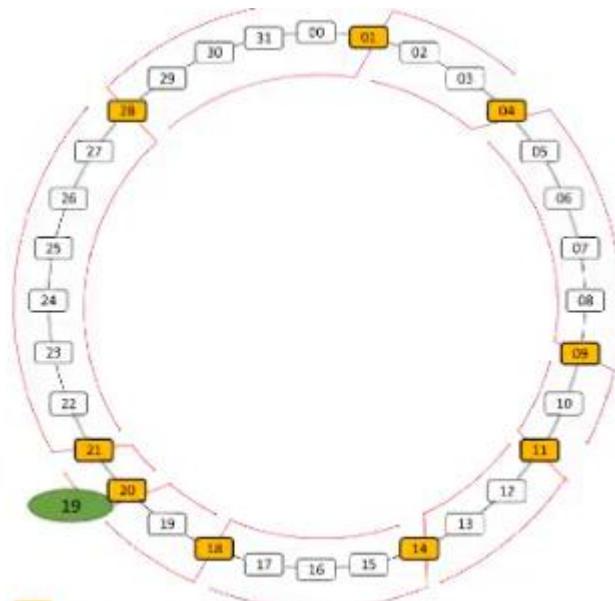
گره ۱۰ میشه ۴ چون ۲۰ و ۳۰ فعال نیستند.

مثلا یک نامی رو hash کردم و شده ۱۹. حالا ۱۹ رو میدم به گره شماره ۱۰ که برای من resol بکنه اما چون شرط:

$$\text{Pred}(p) < K \leq p \quad \text{Pred}(01) = 28 \quad p = 01 \quad K = 19$$

برقرار نیست node ۱۰ نمیتونه resol بکنه.

پس باید یا forward کنیم به pred اش یا به سمت succ اش. حالا دست به دست میره یعنی ۳ تا hop میره از سمت pred به عقب و میرسه به گره ۲۰.



این گره میتوانه access point مورد تقاضا را ارائه بده و موجودیت رو در خودش جا داده.

- این روش scalable n تا ماشین داشته باشم که دارن در این سیستم همکاری میکنن order من n هست که پیچیدگی زمانی بالایی داره. یا اگر یکی از node ها break down شه یا offline بشه مشکل سازه. Latency هم زیاد هست.

به منظور بهبود رویکرد دیگری معرفی شده است:

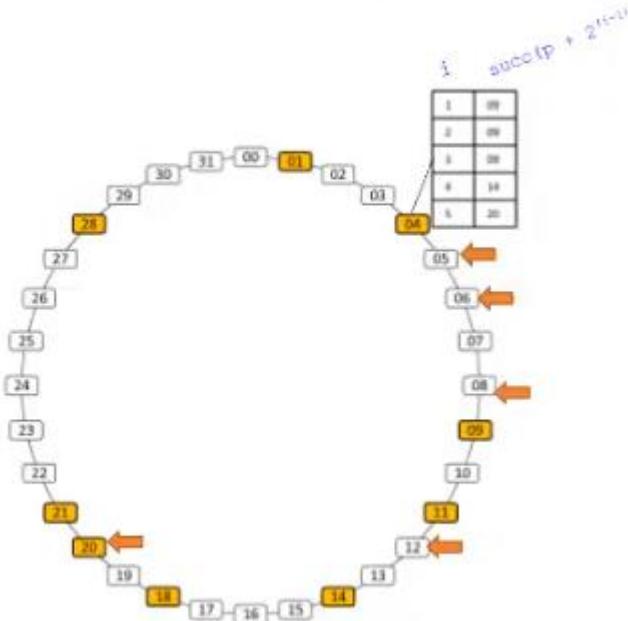
Key Resolution in Chord:

- تعداد node هایی که درگیر resol کردن میشن رو کاهش داده و پیچیدگی زمانی در log n زمانی order هست.
- هر node ای داره و هر چه تعداد سطرهای جدول هر گره بیشتر باشه به تعداد hop کمتری برای resol کردن یک موجودیت نیاز هست.

$$FT_p[i] = \text{succ}(p + 2^{i-1})$$

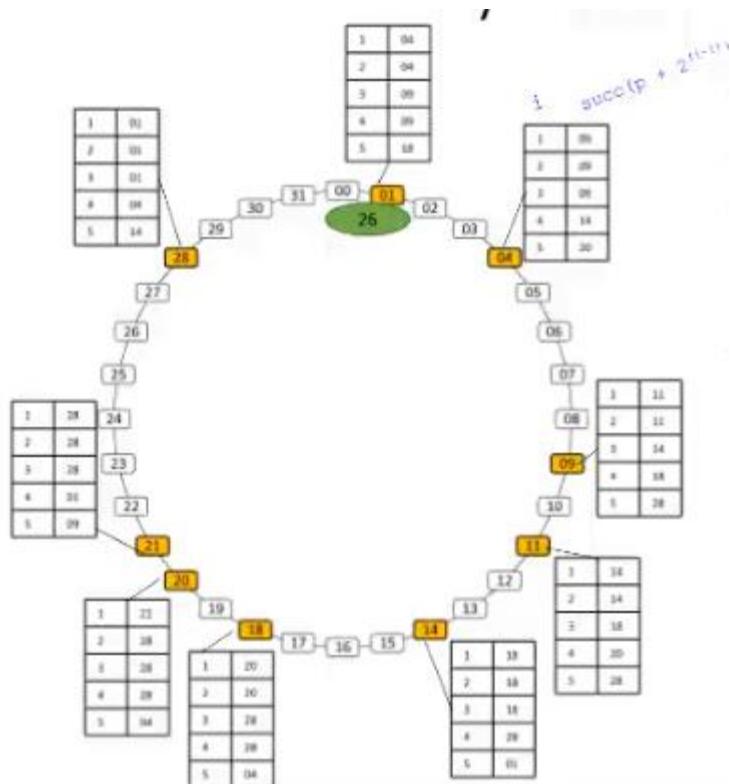
وقتی یک کلید جدید اومد:

ا در اینجا همون index در سطوح مختلف جدول هست و ستون بعدی از فرمول خط بالا حساب میشه.
مثلا در شکل برای $p = 4$ در سطر اول که $1 = \text{اهست}(\text{node } 5)$ بعده فعال بعد از ۵
میشه ۹ پس در جدول می نویسیم ۹.



برای $i = 3, 2, 1$ هم SUCC میشود گره ۹ و برای i بعدی می شود ۱۴ در جدول. یعنی اگر کلیدی با $K = 9$ اومد پاسش بده به گره ۹ و کلیدی بیشتر از ۹ رو بده به گره ۱۴ و در اینجا ۱۱ رو ما bypass کردیم. دوباره اگر $i = 5$ بود میشه گره ۲۰ و دوباره گره ۱۸ اینجا bypass میشه. با این مکانیسم از یه سری نودها رد میشیم و بی جهت به اونها پاس نمیدیم.

برای هر گره رنگی میتونم این جدول رو درست کنم.



در شکل بالا کلید $K = 26$ رو به گره ۱۰ میدیم. میاد این شرط رو حساب میکنه که بینه خودش میتونه `resol` کنه یا باید بدء به گره دیگری:

$$ETp[j] \leq K < ETp[j+1]$$

در اینجا $K = 26$ و بین ۴ و ۹ که نیست پس میره به سطر اخر جدول یعنی گره ۱۸.

در جدول گره ۱۸ کلید ۲۶ از دو سطر اول یعنی ۲۰ بیشتره اما از ۲۸ که سطر سوم هست کمتره پس میفرسته به گره ۲۰.

در گره ۲۰ سطر اول ۲۱ هست و سطر بعدی ۲۸ هست. پس میفرسته به ۲۱.

در گره ۲۱ سطر اول ۲۸ هست پس میده به گره ۲۸.

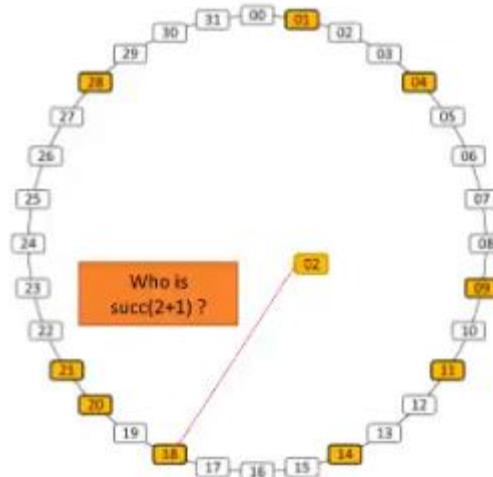
در گره ۲۸ به کلید ۲۶ `resol` و ادرس داده میشه. جاش پیدا شد.

انقدر پاس دادیم که به گره ای رسیدیم که کلید کوچکتر مساوی سطرهای اون گره باشه.

مشکل: خیلی خودمونو اسکل کردیم و پیچیدش کردیم. اگر جدولامونو درست تر مینوشتیم میشد با یک `hop` از ۱۰ بیاییم به ۲۸ و این همه پاسکاری اضافی رو نکنیم.

Chord - join and leave protocol:

بحث این هست که یک سری `node` ها اضافه بشن یا حذف بشن.



طبق تصویر ۲ ، میخواه به داستان اضافه بشه و یک گره رو به صورت random انتخاب میکنه مثل ۱۸ و از اون مپرسه succ من کیه؟ به ترتیب ۱۸ از ۱۴ و ۱۴ از ۱۱ و ۱۱ از ۹ و ۹ از ۴ مپرسه. ۴ میگه منم و ۲ میشنینه سر جاش. حالا ۱ هم باید خودشو update کنه و بدون succ اش دیگه ۴ نیست و ۲ هست. همینطور ۴ هم خوش مفهومه که pred اش ۰ میشه.

حالا اگر ۲ بره بیرون از بازی این عملیات ها بر عکس صورت میگیره که هر گره succ و pred خودشو بدونه.

- میتونیم از network proximity در Chord استفاده کنیم. یه چیزی شبیه Distance vector هست. چون این ring ای که در مثال های بالا دیدیم منطقی هست و overlay network هست و عملاً دستگاه های فیزیکی فاصله زیادی با هم دارند.

- میتونیم از راهکارهای topology-aware node assignment استفاده کنیم. یعنی به جای این که بگیرم یک network address identifier بگیرم و روابط pred و succ رو طبق اون ترتیب بدم و سعی کنم node های مرتبط به هم نزدیک باشند.

2. Structured naming

- اسامی به صورت compose درست بشن و برای اسامی ساختار قائل بشیم
مثلثاً برای این که به یک فایل دسترسی ایجاد کنیم در نظام ساختاریافته به صورت زیر عمل کنیم:

• **/home/userid/work/dist-systems/naming.txt**

یا برای مثال اگر بخوایم یک وبسایت رو address کنم به صورت زیر عمل کنم:

• **www.cs.iust.ac.ir**

- برای این روش فضای آدرس (name space) رو به صورت سازماندهی شده استفاده میکنم.
سازماندهی ما ساختار یافته هست.
- این ساختار یافته گراف باشه (directed graph)

Leaf node ○

هر leaf node میتوانه یک موجودیت باشه که آدرس و state و یا path موجودیت رو ثبت

کنه در خودش

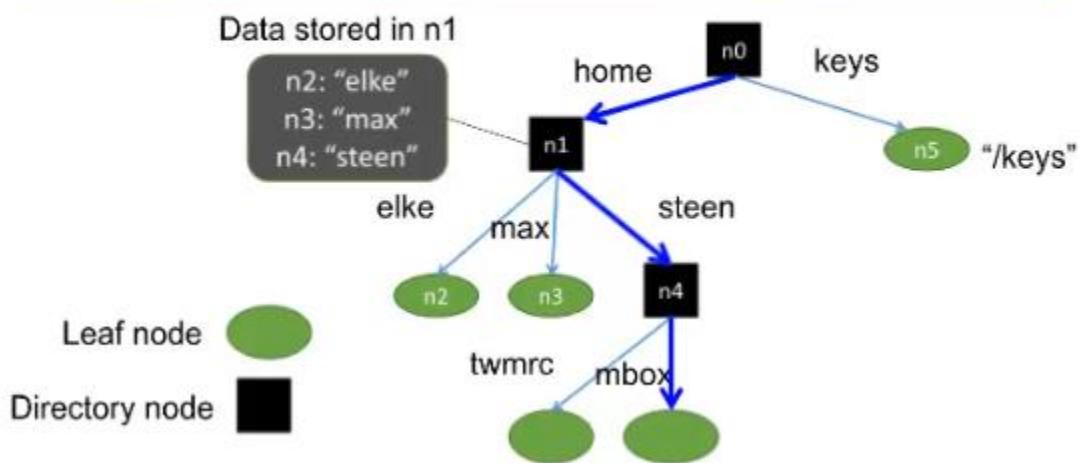
Directory node ○

هر directory node به یک directory node پا leaf node دیگر ارجاع می دهد

هر node میتوانه address یا state یا path را ذخیره کنه ●

مثال در شکل زیر: ●

Looking up for the entity with name “/home/steen/mbox”



به ۴ مکانیسم name resolution هم میگویند. ●

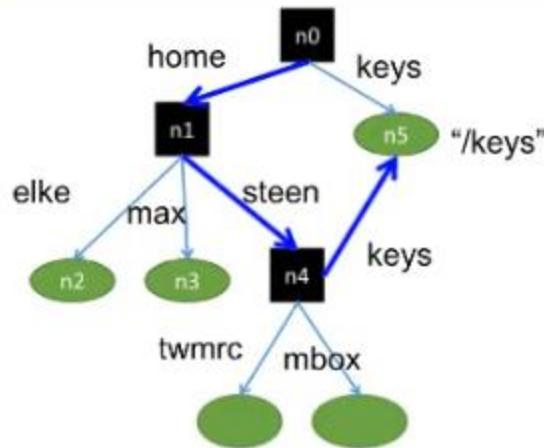
○ خود واژه closure یا بستار به چه معناست: برای مثال اعداد صحیح نسبت به عملیات جمع بسته هست. یعنی جمع دو عدد صحیح باز هم صحیح است.

○ در بحث name resolution گفته میشه که نمیشه این عملیات رو بدون یک initial directory node شروع کرد. یک مجموعه ای باید name resolution رو انجام بدن، از یک شروعی تا یک خاتمه ای.

○ فضای نام میتوانه مورد استفاده قرار بگیره به عنوان لینک بین ۲ موجودیت. ما نوع لینک بین node هامون داریم:

Hard link .a

"/home/steen/keys" is a hard link
to "/keys"

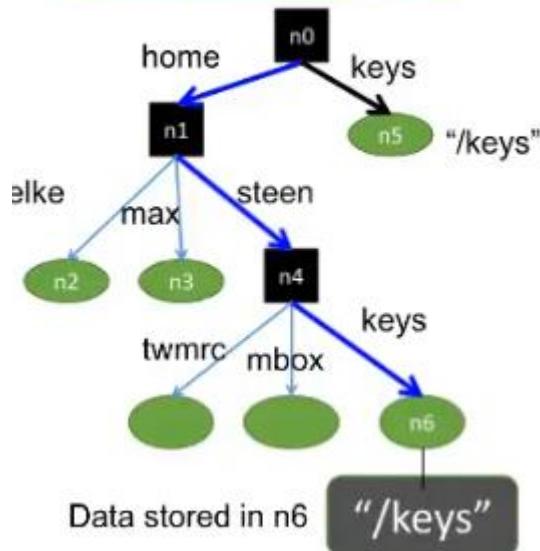


در این حالت بایستی از همون مسیری و لینک هایی که داده شده به موجودیت دست پیدا کنیم. محدودیت و شرایطش اینه که دور و loop نباید داشته باشیم.

Symbolic link .b

در این حالت نام هر گره original به عنوان data ذخیره میشه.

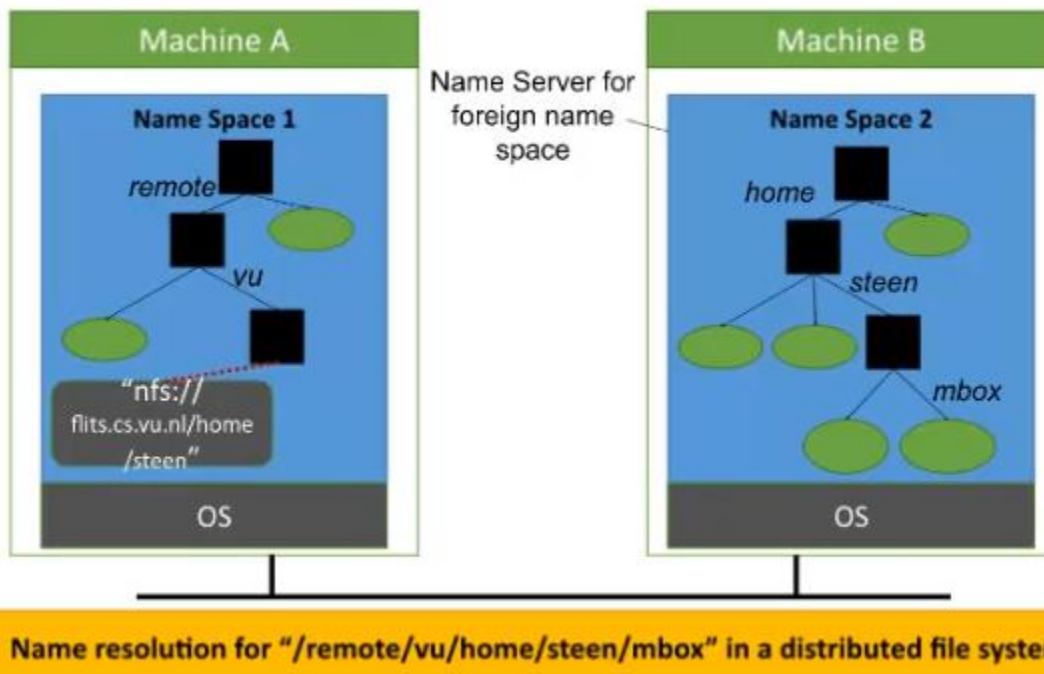
"/home/steen/keys" is a symbolic link to "/keys"



• نکته دیگر در بحث mounting namespace بحث هست

یعنی سوار کردن یا براق کردن. یعنی به چیزهایی در ش اتفاق بیوفته به واسطه براق کردن که فرمان پذیر شود.

در حوزه naming میتونم name space ها روی همیگر mount کنم و سوار کنم. NFS یا همون Network File System عملا همین کارو انجام میده.



در شکل بالا ما دو تا فضای نام جدگانه در ماشین A و B داریم. در ماشین A میخواستیم address کنیم اون آدرسی که نوشته شده رو ولی جاش یه فضای نام دیگه هست. ۱ Name space ۱ باید به صورت transparent منو وصل کنه به اون name space ۲. در واقع این دو تا به هم mount می شوند، یعنی فضای نامی که در ماشین B هست میشه در ماشین A.

Attribute-based naming . ۳

- ۱۱ جلسه -

:structured-based naming

Distributed name space: •

معمولًا فضای آدرس را توزیع شده در نظر میگیرند در سیستم‌های توزیع شده چون این سیستم‌ها large-scale هستند یعنی گستردگی از نظر اندازه و جغرافیا و administration name resolution دارند. بحث میتوانه ما به ازای هر کدام از این ابعاد توزیع شده در نظر گرفته بشه. این موضوع به ۳ حالت امکان پذیر است:

- 1. Distributed the nodes of the naming graph
- 2. Distributed the name space management
- 3. Distributed the name resolution mechanisms

Layers in Distributed name space •

میتوان در ۳ سطح این فضای نام رو لایه بندی کرد: (این ۳ سطح رو متناظر ۳ حالت تعریف شده در بالا معرفی کرده اند)

1. Global Layer

در این سطح یک سری directory node داریم که میتوانن توسط administration های مختلف مدیریت بشن.

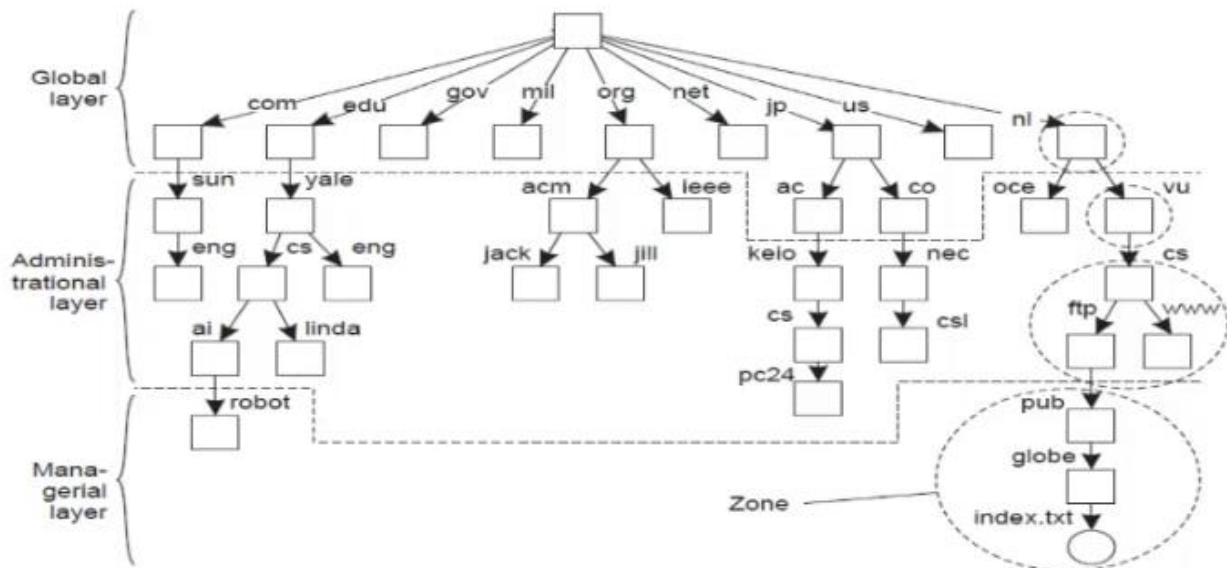
2. Administrational Layer

این لایه حالت سازمانی تر دارد و در اینجا هم یکسری directory node داریم که با هم group شدن و هر گروه تحت یک administration هست.

3. Managerial Layer

لایه خرد تر و ریزدانه تری هست که زیر یک administration هست و directory node های low-level resolution هستند که عملاً نزدیک به ای هست که بهش نیاز هست. مثال این ۳ لایه در اینترنت به صورت زیر است:

Distributed Name Spaces – An Example



در این مثال سمت راست nl میشه برای کشور هلند هست و در لایه بعدی VU برای یک دانشگاه است و در لایه بعدی خوش تقسیم شده به بخش هایی که در تصویر مشخصه. در اینجا فضای آدرس ما یک ساختار توزیع شده و لایه لایه دارد.

مقایسه ای از این ۳ لایه ارائه شده:

	Global	Administrational	Managerial
Geographical scale of the network	Worldwide	Organization	Department
Total number of nodes	Few	Many	Vast numbers
Number of replicas	Many	None or few	None
Update propagation	Lazy	Immediate	Immediate
Is client side caching applied?	Yes	Yes	Sometimes
Responsiveness to lookups	Seconds	Milliseconds	Immediate

از دید وسعت جغرافیایی بحث global در سطح جهانی هست و administration در حد یک سازمان هست و managerial در حد یک دپارتمان یک دانشگاه هست.

تعداد گره ها در لایه اول کم و به ترتیب زیاد و زیادتر میشه.

در بحث replica برای محدوده managerial کلا وجود نداره.

سرعت به روزرسانی در global کند و معمولاً انجام نمیشه.

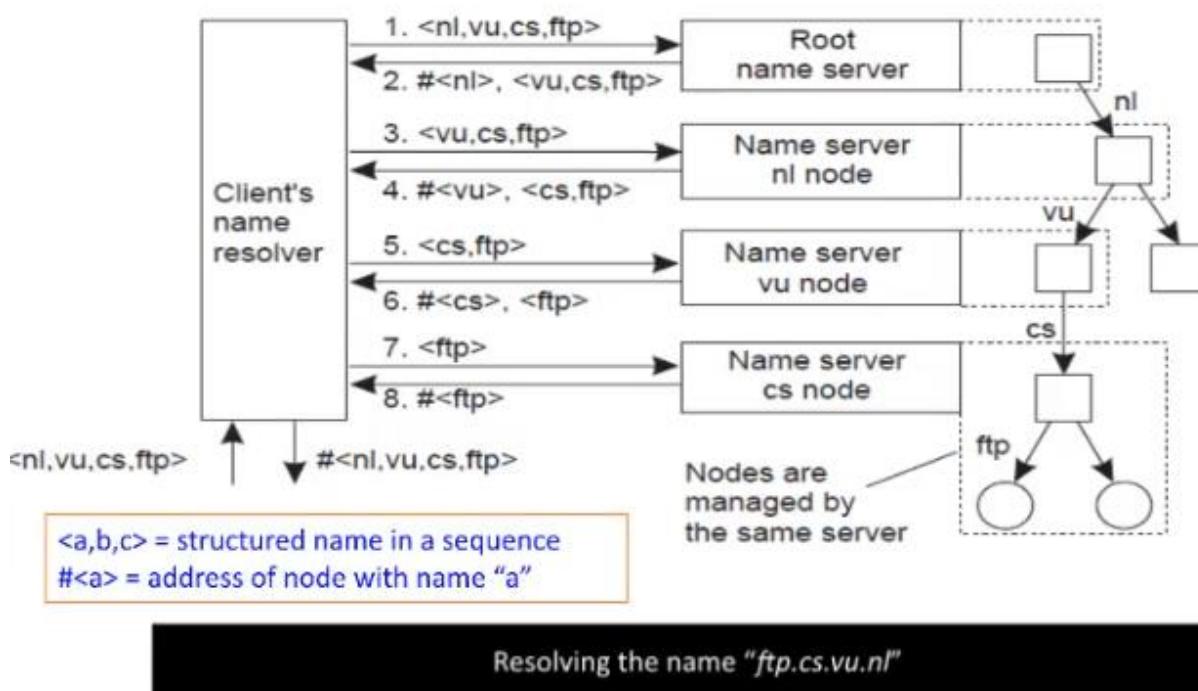
این که cache صورت بگیره در ۲ لایه بالا صورت میگیرد اما در لایه انتهای گاهی اوقات.

پاسخگویی این که name بدمی و آدرس تحويل بگیریم در global در حد ثانیه هست و لایه به لایه کم و کم تر میشه.

Distributed name resolution •

۲ روش برای این کار وجود دارد:

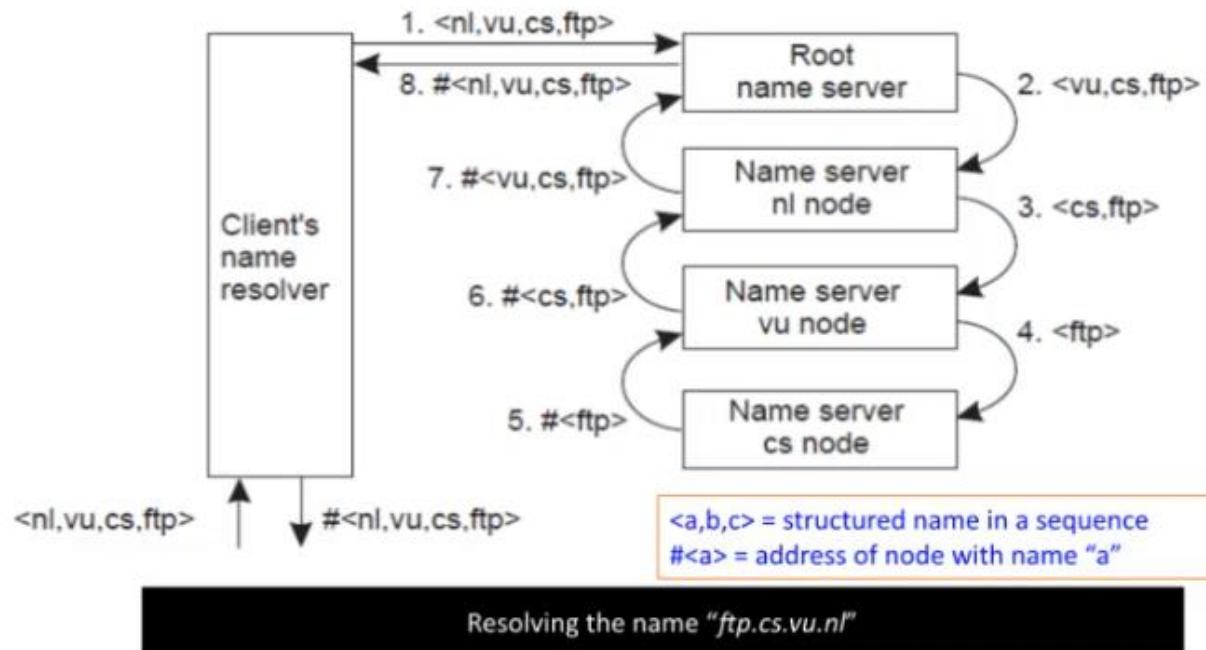
1. Iterative name resolution .



در سمت چپ تصویر یک name resolver داریم و به یک سری name server دیگر دسترسی داره. اول یک name server root پیدا میکنه و موجودیت که میخواهد resol کنتش رو بهش میده. اون root آدرس nl رو مشخص میکنه و بر میگردونه اطلاعات بقیشو دیگه این name server نداره و برای پیدا کردن باقی آدرس یک name server دیگر به نام nl رو پیدا میکنه و اون VU رو میکنه و به همین ترتیب آدرس موجودیت ای که می خوایم درمیاد. پس عملاً client داره iterate میکنه و با cooperate کردن با name server های دیگر به صورت recursive آدرس موجودیت رو به دست بیاره. این روش ساختار یافته هست.

• همه client ها باید نسبت به همه name server ها آگاهی داشته باشند

2. Recursive name resolution .



به جای اینکه **iterative** صورت بگیره من درخواست آدرس را به یدونه **root name server** میدم و اون هر چقدر شو تونست **resolution** میکنه و هر جایی رو نتونه پاس بدی به یک **name server** دیگه ای که اون **resolution** رو انجام بدی و قطی آدرس موجودیت مشخص شد برمیگرده عقب مدام تا به همون **client** بررسه و اون آدرس رو به **root name server**.

- ممکنه سربار زیادی روی **name server** ها باشد چون دخالت **client** کم هست
- خوب این روش این هست که **client** ها فقط یکدونه **root name server** میشناسند

نوع سوم سیستم های : **Naming**

Attribute-based Naming:

در این روش دسترسی به موجودیت از رهگذر یک سری ویژگی ها و خصایص هست. عملاً مثل **yellow page** داره عمل میکنه. دنبال هر چیزی مثل آدم و آدرس و شماره تلفن میگردیم سراغ این **yellow page** میرویم که یک **directory** میزیم. در این **service** هست.

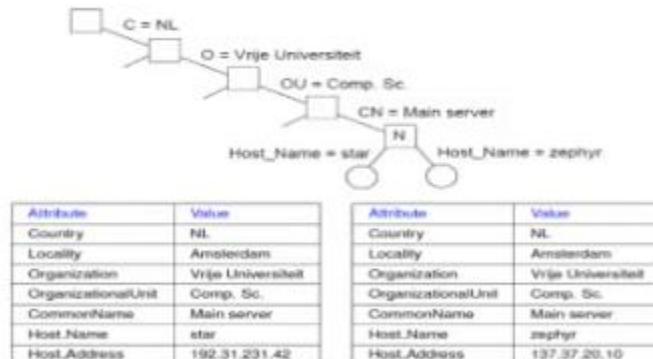
سربار این روش خیلی بیشتر از سربار روش های قبلی هست چون مدام باید **matching** صورت بگیره. برای جلوگیری از سربار زیاد یک **database management** ایجاد میکنن و یک **query** روش قرار میدن و ما به اون **query** میزنیم. در این DB ما تمام ویژگی و خصایص یک موجودیت رو در سطر مربوط به اون موجودیت قرار میدیم. پروتکل **Light-weight Directory Access Protocol = LDAP** در این بحث ارائه شده است.

- LDAP:

چیزی که در روش های قبلی به صورت ساختار یافته به name server ها داده میشد به صورت یک مجموعه می شود طبق شکل زیر: ۵ تا attribute اول رو همه موجودیت های DB باید داشته باشند که کشور و سازمان و ... است.

Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Comp. Sc.
CommonName	Main server
Host_Name	star
Host_Address	192.31.231.42

حالا اگر به این DB یک query زده بشه مشخص میشه:



اینجا به صورت DB میگردد و پیدا میکنه.

Computing's Paradigm:

یک چیز مجازی از مهندسی، علوم و ریاضیات هست.

:Taxonomy

طبقه بندی، make seprated groups based on features

در برنامه نویسی هم ما class تعریف میکنیم که مجموعه ای از attribute ها هست و مجموعه از operation ها هست. میخواهیم مسائل دنیای بیرون رو نگاشت کنم به فضای محاسباتی کامپیوتر. همه object ها رو هم نگاه میکنم و اون ها رو classify میکنم در فضای محاسباتی. این طبقه بندی و دسته بندی بر حسب تشابهات رفتاری، داده ای، پردازشی صورت می گیرد. این دسته بندی هارو draft میکنم و به یک زبانی طراحیش می کنیم و تبدیل به class میکنیم و کلاس کد مرده هست برای این که زندش کنیم ازش instance می سازیم.

فلانی در فلان علم خیلی عالم هست یعنی اون علم رو taxonomize کرده و اگر به چیز جدیدی بررسه میدونه مربوط به کدوم دسته و بخش هست.

Taxonomy Computing:

این دسته بندی هایی که ارانه میشه جامع و مانع نیست یعنی وحی منزل نیست که فقط همینا باشد.

- **Large-Scale Computing**

- Distributed Computing; Parallel Computing; Macro- And Micro- Computing; Activity-Based Computing; Data Intensive Computing; Resource-Constraint Computing; Grid Computing; Cloud Computing; Cluster Computing; On-Demand Computing; Ubiquitous/Pervasive Computing; Memristor Computing; Unconventional Computing; Evolutionary Computing; ULSC

- **Mechanism-Oriented Computing**

- Spatial Computation; Elastic Computing; Human-Centered Computing; Embedded Computing; Entertainment Computing; Time-sensitive/Temporal Computing; Soft Computing (Fuzzy Logic, Neural Computing, Evolutionary Computation, Machine Learning, and Probabilistic Reasoning + Belief Networks, + Chaos Theory + Learning Theory)

- **Technology-Oriented Computing**

- Peer-to-Peer Computing; Mobile Computing; Sensor-based Computing; Wireless Computing; Trusted Computing; Financial Computing; Genetic Computation

نکاتی در مورد تصویر بالا:

محاسبات توزیع شده رو به طور اعم به عنوان large scale computing میشناسند. چرا به طور اعم؟ چون محاسبات توزیعی ای که small scale باشه و داده و پردازش کم باشن و شبکه کوچک باشه که large scale computing نیست.

ULSC در واقع locality هست یعنی مثلاً ممکنه اون فقط کره زمین نباشه و مریخ رو هم شامل بشه.

دسته بندی دوم در تصویر دسته بندی مبتنی بر مکانیسم هست.

دسته بندی سوم مبتنی بر تکنولوژی می باشد. مثل mobile computing که قبلاً نبود و الان هست.

- **Intelligent Computing**
 - Cognitive Computing; Intelligent Computation; Ambient Computing; Unconventional Computing; Indeterminist Computing; Adaptive Computation; Autonomic Computation; Computation under Uncertainty; **Chaotic Computation**; Intentional Computing; Anticipative Computing; Evolutionary Computing
- **Computing Technologies**
 - Quantum Computing; Optical Computing; DNA (Genetic) Computing; Molecular Computing; Reversible Computing; Billiard Ball Computing; Neuronal Computing; Magnetic Computing; Gloopware Computing; Moldy Computing; Water Wave-based Computing; Graphene-Based Computing

در این تصویر طبقه بندی آخر یعنی Computing technologies بیشتر محاسبات applied هستند.

Comouting's Paradigm:

عملاً computing میکنه یعنی نمونه هست مهندسی و علوم رو اما نه مهندسی هست و نه غلوم. یعنی محاسبات نمونه ای از مهندسی و علوم هست ولی هر مهندسی و علومی محاسبات نیست.

: paradigm تعریف

پارادایم یا نظام فکری یک سیستم اعتقادی و belief system میکنه یعنی الزامات و متعلقات اون نظام فکری رو practice میکنه. مثلاً من مسلمانم آیا به متعلقات اسلام هم عمل میکنم؟ یعنی هم باید اعتقاد وجود داشته باشه هم پایداری نسبت به اون اعتقاد. در واقع دیدگاه ما به دنیا و مسائل در آن چیست و چه راهکارهایی برای حل آن مسائل داریم. این سیستم اعتقادی نبایستی مدام عوض بشه، اگر taxonomy درستی قبیلش انجام داده باشم این سیستم برای من مشخص هست.

مثال پارادایم ها:

- Engineering: electrical, mechanical, chemical, civil
- Science: Physical, Life, Social Sciences

مباحثه بین صاحبان نظر در مورد :computing

این ۳ دیدگاه برای فراهم شدن یک دیدگاه جامع ارائه شده است:

- [Alan Perlis, Allen Newell and Herb Simon, 1962]
 - Computing being unique among all sciences and engineering in its study of **information processes**
- [Edsger Dijkstra and Donald Knuth, 1967]
 - **Programming**, seen as the art of designing information processes
- [Bruce Arden – NSF, 1970]
 - Computing as **automation of information processes** in engineering, science, and business

- در دیدگاه اول گفته شده محاسبات یک ویژگی مشخصی نسبت به علوم و مهندسی دارد و آن مطالعه فرآیندهای اطلاعاتی است. یعنی فرآیندهای اطلاعاتی رو فقط در computing داریم.
- در دیدگاه دوم می گویند Program کردن کامپیوتر هنر فرآوری و پردازش اطلاعات است. در اینجا ترجمه صحیح programing برنامه سازی است و نه برنامه نویسی
- در دیدگاه سوم گفته می شود اگر بتوان فرآیندهای اطلاعاتی در حوزه های مهندسی، علوم و تجارت را automate کرد میشود محاسبات یا computing

از سال ۱۹۹۷ تا ۲۰۱۳ فرض بر این بود که **Information Technology** یا به اختصار **IT** همون **Computing** Discipline هستش ولی سال ۲۰۱۳ یه مقاله‌ای توی ACM به چاپ رسید و با شواهد نشون داده که اینطوری نیست. در واقع **IT** نهایتاً یه سری زیرساخت‌های فناورانه با دغدغه‌ی مالی هستش. بیشتر جنبه بهره برداری دارد.

زیر پارادایم‌های نهفته در محاسبات (sub-paradigms embedded in computing) ریاضیات:

	Maths
Initiation	Characterize objects of study (definition)
Conceptualization	Hypothesize possible relationships among objects (theorem)
Realization	Deduce which relationships are true (proof)
Evaluation	Interpret results
Action	Act on results (apply)

در فاز اول میان یه تعریفی میدن از دنیای مورد مطالعه... در فاز دوم میان احتمال ارتباط بین این اشیا رو نظریه سازی میکنن. در فاز سوم میان یه proof-checking میکنن یعنی اثبات صحت و درستی اون تئوری که ببین آیا درست بوده یا نه... در فاز ارزیابی میان نتایج رو تفسیر میکنن. در نهایت هم میان از این نتایج در نظریه‌های بعدی استفاده میکنم. از این نتایج در حرکت بعدی خودم یا دیگران استفاده کنم.

علوم:

	Science
Initiation	Observe a possible recurrence or pattern of phenomena (hypothesis)
Conceptualization	Construct a model that explains the observation and enables predictions (model)
Realization	Perform experiments and collect data (validate)
Evaluation	Interpret results
Action	Act on results (predict)

در علوم فازها کمی متفاوت هست. ابتدا میان احتمال رخداد یک الگو رو مشاهده میکنند (نظریه). سپس یه مدل که توضیح دهنده مشاهده و بیان کننده پیش‌بینی‌ها هستش رو میسازن (مدل). در قدم بعد شروع به آزمایش و جمع آوری داده میکنند (ارزیابی - بر عکس ریاضی که می‌شست proof میکرد). سپس نتایج رو ارزیابی کرده و در انتها نیز روی نتایج عملی انجام میده (predict).

مهندسی:

	Engineering
Initiation	Create statements about desired system actions and responses (requirements)
Conceptualization	Create formal statements of system functions and interactions (specifications)
Realization	Design and implement prototypes (design)
Evaluation	Test the prototypes
Action	Act on results (build)

در علم مهندسی اولین کاری که انجام میشه میگن «خب ما میخوایم یه چیزی بسازیم حالا هر چی... نیازمندی‌هایی که متصور هستیم چیه؟». در فاز بعد تلاش میکنه یه وضعیت مشخص از عملکردهای سیستم و تعاملات بین‌شون رو استخراج

میکنے. سپس نمونه‌هایی از اون محصول مورد نظر طراحی و پیاده‌سازی میکنے. سپس این نمونه‌ها رو تست کرده و بر حسب بازخوردنی که از تست‌ها گرفتم میرم سراغ ساختش.

دیگه ما مفهوم information رو فقط محدود به computing machines نمیدونیم و وارد فاز process شدیم و حتی میتوانیم پردازشی رو DNA ها داشته باشیم.

دیگه فقط محدود به Computing

الگوریتم‌ها، ساختمان داده، نظریه اعداد. زبان‌های برنامه‌نویسی، سیستم‌های عامل، شبکه‌ها، پایگاه‌های داده، گرافیک‌ها، هوش مصنوعی، مهندسی نرم‌افزار

نیست (تا قبل ۱۹۸۹ بهشون میگفتند Computing). ولی الان اگر کسی موارد زیر رو ندونه نمیشه بگیم میدونه:

اینترنت، علوم وب، پردازش موبایل، طراحی AI، مصورسازی اطلاعات، شبکه‌های اجتماعی، استریم کردن ویدئو چارچوب‌های جدیدتر عبارتند از:

الگوریتم‌های هیوریستیک، داده توزیع شده، شبکه‌های توزیع شده، شبکه‌های اجتماعی و ...

این موارد پیچیده هستند و با کار ریاضی دان‌ها قابل مطالعه و بررسی نیستند و بیشتر کار scientist‌ها هست که این‌ها رو بررسی کنند.

نکته: دانشمندان بیشتر دنبال اکتشاف‌نما ساخت و طراحی.

اکتشاف و طراحی به صورت نزدیک با هم ارتباط دارند:

- رفتار تعداد زیادی از سیستم‌های بزرگ (مثل وب) از طریق مشاهده کشف شده. شبیه‌سازی به منظور تقلید کشف کردن فرآیندهای اطلاعاتی طراحی شده.

فریمورک‌های جدید همچنین به عنوان natural information processes شناخته میشه که شامل: علوم شناختی در موجودات زنده، فرآیندهای فکری، تعاملات اجتماعی، اقتصادها و ...

تمرکز اصلی computing paradigm در information processes (میتوانه فرآیندهای طبیعی یا ساخته شده باشه که اطلاعات رو منتقل میکنه - میتوانن گستره یا پیوسته باشن) خلاصه می‌شود.

• در واقع computing فرآیندهای اطلاعاتی رو «عبارت‌هایی (Expressions) که کاری انجام میدن.» بیان میکنن. یه عبارت، توصیف مرحله یک فرآینده که به صورت تجمعی دستور العمل هاست.

- «عبارات میتوزن مصنوع (artifact) باشن». مثل برنامه‌هایی که توسط مردم طراحی یا خلق شدن. یا توصیفی از رخدادهای طبیعی DNA یا تفسیر DNA در بیولوژی. مثل داستان سفینه‌های ناسا که تو دمای بالا سفینه آتش نگیره که خروجی یه تیکش شد تغلون قابل‌مها.
- «عبارات فقط representational نیست، در واقع generative هستن. اونا موقع تفسیر (اجرا) توسط ماشین‌های مناسب، اقداماتی رو ایجاد میکنن.
- این عبارات مستقیماً محدود به قوانین طبیعی فیزیکی نیست (یعنی بر اساس تصورات و تخیلات میتونیم این عبارات رو تعیین کنیم).
- در شبیه‌سازی کارهای علمی ما عوارض یا side effect های فیزیکی نداریم یعنی اگر یه ماده به مرز انفجار برسه اتفاقی نمی‌افتد جز اینکه من یه هشدار بدم که بورووووم ترکید.
- بعضی از این روش‌ها به ریاضیات محض برای اثبات اینکه اقدامات ایجاد شده توسط عبارات با مشخصات مطابقت دارند، تکیه می‌کنند.
- بسیاری دیگه هم برای تایید فرضیه‌های مربوط به رفتار اعمال و کشف حدود عملکرد قابل اطمینان آن‌ها، به آزمایشات (experiments) اعتماد می‌کنند.

از دیدگاه Computing:

	Computing
Initiation	Determine if the system to be built (or observed) can be represented by information processes , either finite (terminating) or infinite (continuing interactive).
Conceptualization	Design (or discover) a computational model (for example, an algorithm or a set of computational agents) that generates the system's behaviors.
Realization	Implement designed processes in a medium capable of executing its instructions . Design simulations and models of discovered processes. Observe behaviors of information processes.
Evaluation	Test the implementation for logical correctness, consistency with hypotheses, performance constraints, and meeting original goals. Evolve the realization as needed.
Action	Put the results to action in the world. Monitor for continued evaluation.

نقطه شروع کار اینکه آیا چیزی که دارم می‌بینم یا چیزی که می‌خوام بسازم به صورت یه سری فرآیند اطلاعاتی (finite) یا قابل بیان کردن هست یا نه (finite یعنی نقطه پایان داشته باشه و infinite یعنی نقطه خاتمه‌ای نداشته باشه). اگر بود در ادامه می‌ام یه مدل محاسباتی طراحی می‌کنم که بتونه رفتار سیستمی رو تولید بکنه. در قدم بعد می‌ام یه سری

طراحی میکنم، که در نهایت به یه برنامه تبدیل میشه. میام شبیهسازی و مدل‌های اکتشافی فرآیندها رو طراحی میکنم که نشون بدیم این **instruction** ها روی هوا نباشن. در نهایت هم رفتار فرآیندهای اطلاعاتی رو هم مشاهده می‌کنم. در فاز ارزیابی نیز پیاده‌سازی رو برای درستی منطق، سازگاری با فرضیه‌ها، محدودیت‌های کارابی و رسیدن به اهداف اصلی تست میکنم. در فاز **action** هم میایم کار رو توی دنیای واقعی قرار میدیم و به صورت دائمی به منظور ارزیابی مانیتور خواهیم کرد.

- مهندسا، دانشمندا و ریاضی‌دانان جایگاه **outside observer** دارن یعنی از بیرون نگاه میکنن و توی سیستم نیستن و به همین خاطر تومم کارهاشون **representational** هستش (یعنی **generative** یا مولد نیست). بنابراین نمونه اولیه‌های سنتی، مدل‌های علمی و مدل‌های ریاضی قابلیت اجرایی ندارن.
- زمانی که با سیستم‌های محاسباتی ترکیب میشن، سازنده‌های خودکار، شبیه‌ساز مدل‌ها کتابخانه‌های نرم‌افزاری ریاضی در اختیارشون قرار خواهد گرفت.
- عبارات محاسباتی محدودیتی از این نظر که حتما خارج سیستمی باشند که نشون میدن ندارد.
- عبارات محاسباتی از ویژگی به نام **self-reference** برخوردارند (همون ارجاع بازگشتی به خود شی در **natural information** (برنامه‌نویسی شی‌گرا)). در واقع **self-reference** در فرآیندهای اطلاعاتی طبیعی (processes) رایج است. برای مثال یک سلول اطلاعات مربوط به خودش رو درونش داره.

۱۲ انتهای جلسه

جلسه ۱۳

Synchronization

جلسه قبل در خصوص نامگذاری و پارادایم‌های محاسباتی بحث کردیم و این جلسه سراغ مبحث زمان در سیستم‌های توزیع شده میریم. راه‌هایی که ارائه شده و قرار بهشون بپردازیم:

- یک ساعت واحد جهانی (UTC)
- زمان در حوزه‌ی کامپیوتر و سیستم‌های توزیع شده
- اگر بخوایم ساعتها رو تو سیستم‌ها با هم تنظیم کنیم چه الگوریتم‌هایی وجود داره (Cristian, Berkeley, NTP)

زمان: **Time**

ساعت: Clock

سؤال: تفاوت Time و Clock چیست؟

زمان گذار از یک حالت به حالت دیگه رو داره به من نشون میده، اما ساعت یک ابزار برای نشون دادن گذار زمان هستش.
ساعت رو میشه عقب و جلو کرد ولی زمان رو نمیشه عقب برد.

چیزایی که تا الان بحث شده:

- اینکه چطور موجودیت‌های سیستم‌های توزیع شده با همیگه ارتباط برقرار میکنن.
- چطور موجودیت‌های نامگذاری و شناسایی میشن.
- **اما** علاوه بر این موارد، موجودیت‌ها خیلی مایلن تا به صورت هماهنگ به حل یک مسئله بپردازن.
- برای مثل در سیستم‌های توزیع شده پروسس‌ها باید با همیگه هماهنگ یا Sync باشن و بتونن همکاری داشته باشند، به طوری که دو پروسس به صورت همزمان روی یک فایل شروع به نوشتن نکنن.

نیاز به هماهنگ‌سازی - مثال ۱

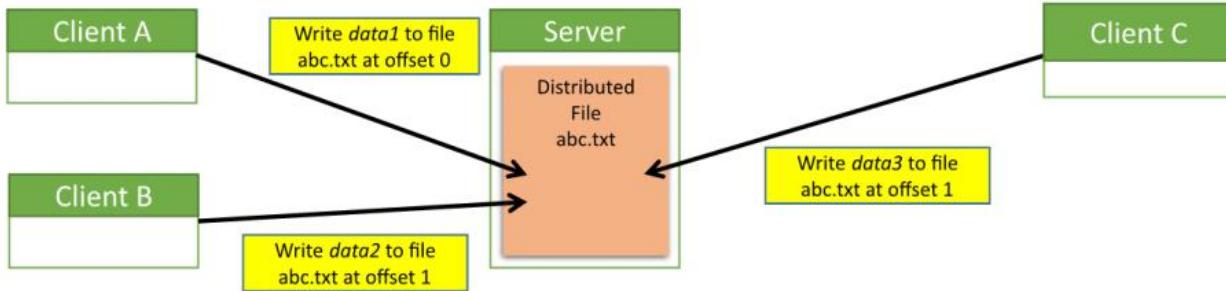
مثلثاً سناریویی رو در نظر بگیریم که یک ماشین از یه میدا تو شهر میخواهد به یه مقصدی بره و در سر راه دوربین‌های نظارتی وجود داره که بخوان مکان و زمانی که درش بوده رو ثبت کنن. اگر ساعت دوربین‌ها هماهنگ باشن که خوب خیلی خوبه... مشخصه اون وسیله نقلیه در چه زمانی کجا بوده و چقدر طول کشیده تا از میدا به مقصد بره. ولی مصیبت اونجاست که ساعات این دوربین‌ها با همیگه هماهنگ نباشند، مثلًا طرف ۳ بعد از ظهر راه افتاد و ۱:۳۰ رسیده مقصد و یعنی انگار در زمان به عقب برگشته.



نیاز به هماهنگسازی - مثال ۲

نوشتن بر روی یک فایل در فایل سیستم‌های توزیع شده

در این سناریو مشابه شکل زیر، دو سیستم B و C قصد دارند تا روی سرور فایلی به نام abc.txt را تغییر بدن و روش بنویسن. حال اگر ساعت هماهنگ نباشد بعد از نوشتن، داده روی فایل خراب خواهد شد.



A Broad Taxonomy of Synchronization

دلالی هماهنگسازی و همکاری	مواردی که برای هماهنگسازی و همکاری مورد نظر هستند	مواردی که برای هماهنگسازی و همکاری مورد نظر نیستند
مثال‌ها	- برای مثال مسیریابی یک وسیله نقلیه توسط دوربین‌ها - تراکنش‌های مالی در سیستم‌های تجارت الکترونیک توزیع شده	- خواندن و نوشتن بر روی یک فایل سیستم توزیع شده
نیازمندی موجودیت‌ها (که ممکنه منع هل بدء به سمت هماهنگسازی)	مواردی که باید فهم مشترکی از زمان بین کامپیوترهای مختلف داشته باشند	مواردی که باید هماهنگ باشند و روی اینکه چه زمانی و چگونه به منابع دسترسی داشته باشند، به توافق برسنند
مباحثی که خواهیم خواند	همانگسازی زمانی	Mutual Exclusion

همانگسازی زمان

- هماهنگسازی ساعت فیزیکی (یا به صورت ساده هماهنگسازی ساعت):
- در اینجا، زمان حقیقی روی کامپیوترها هماهنگ می‌شون.
- هماهنگسازی ساعت منطقی

- کامپیوترها بر اساس ترتیب نسبی رخدادها هماهنگ میشن.

Mutual Exclusion

- جطوری بین پردازهای مختلف که به منع یکسان دسترسی دارن، هماهنگی ایجاد کنیم.

الگوریتم‌های انتخابات

- در اینجا، گروهی از موجودیت‌ها یک موجودیت رو به عنوان هماهنگ‌کننده برای حل مسئله انتخاب میکنن.

هماهنگ‌سازی ساعت

سازوکاری به منظور هماهنگ‌سازی زمان همه‌ی کامپیوترها در سیستم توزیع شده است.

ساعت هماهنگ جهانی یا (UTC))

- همه‌ی کامپیوترها به صورت عمومی با یک زمان استاندارد به نام UTC هماهنگ میشن.

- ابزار UTC یک زمان استاندارده که تمام کامپیوترها از اون برای تنظیم ساعت و زمان خودشون استفاده میکنن.

مقدار UTC از طریق ماهواره‌ها broadcast میشه.

- چون فاصله فیزیکی بین ماهواره و زمین وجود داره، پس قطعاً یک فاصله زمانی حدوداً ۰.۵ میلی ثانیه وجود خواهد داشت.

سرورهای کامپیوتری و سرویس‌های آنلاین که سیستم UTC receivers دارن میتوانن با استفاده از broadcast ماهواره‌ای هماهنگ بشن.

- تعداد زیادی از پروتکل‌های مشهور هماهنگ‌سازی در سیستم‌های توزیع شده میتوانن از UTC به عنوان یک زمان مرجع به منظور هماهنگ کردن ساعت کامپیوترها استفاده کنن.

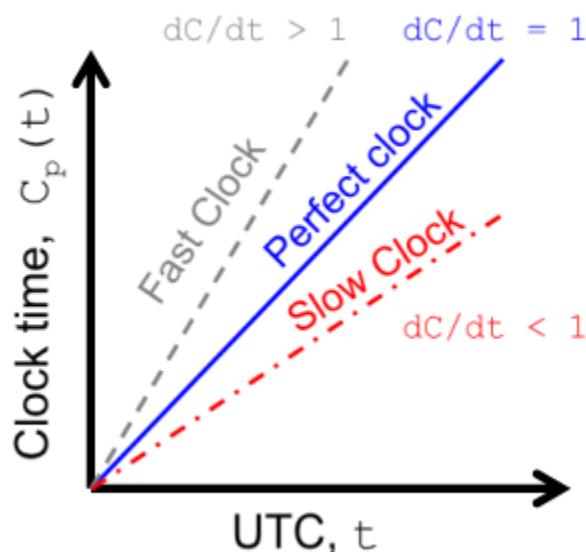
نگهداری زمان بر روی یک کامپیوتر

چطور یه کامپیوتر میتونه زمان خوش رو نگهداری کنه؟

- هر کامپیوتر یک ساعت سخت‌افزاری داره.

- تعداد H وقهه رو در واحد زمان (ثانیه) ایجاد میکنه.
- یک handler وقهه هستش که ۱ واحد به ساعت نرمافزاری (C) اضافه میکنه.
- مشکلات ساعت ها بر روی یک کامپیوتر
- در عمل، تایمر سختافزاری دقت کمی داره.
- به دلیل نقص مواد سازنده سختافزار و دماهای مختلف دقیقا H بار در ثانیه وقهه ارسال نمیکنه.
- کامپیوتر زمان رو آهستهتر یا تندتر از زمان واقعی میشمره (clock skew).
- مبحث Clock Skew حاشیه بین:
- ساعت کامپیوتر و زمان واقعی (مثل UTC) هستش.

چند نوع Clock Skew داریم:



حب حالا فرض کنیم متوجه شدیم ساعتمون سریعتر بوده یا کندتر، چیکار میتونیم باهاش بکنیم؟ مثلا تو حوزه الکترونیک درسته یکی میگه این دستگاه ۲۲۰ ولته ولی از ۲۰۰ تا ۲۵۰ ولت رو پذیرا هستش ولی کمتر و بیشتر از این مقدار رو دیگه مشخص نیست چی میشه. داخل حوزه زمان هم یه همچین چیزی به نام Maximum Drift یا Rate p داریم که به صورت زیر تعریف میشه و میگه من که هیچ وقت نمیتونم ادعا کنم perfect clock دارم ولی با این p میتونم به اختلاف زمان با زمان UTC دست پیدا کنم:

$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$$

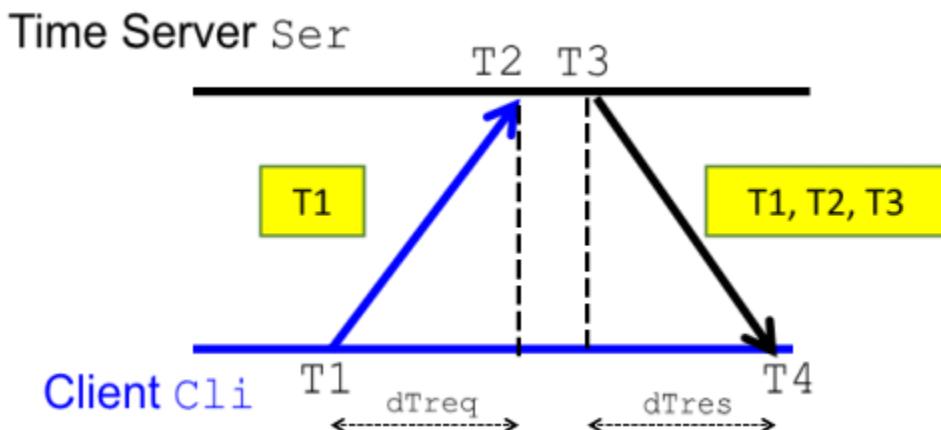
دو ساعت چقدر میتوانی با هم اختلاف زمانی داشته باشن؟

اگر دو ساعت هر دو در جهت های مختلف زمانی نسبت به UTC قرار داشته باشن (یعنی یکی سریعتر بره و یکی کندتر) و در زمان ΔT بعد از هماهنگ سازی شون، میتوونند اندازه $p\Delta T^2$ ثانیه اختلاف زمانی داشته باشن.

تضمین Maximum drift بین کامپیوترها در سیستم‌های توزیع شده:
 اگر maximum drift ممکن در یک سیستم توزیع شده δ ثانیه باشد، بنابراین ساعت هر کامپیوتر باید حداقل هر $\delta/2p$ ثانیه دوباره هماهنگ شوند.

Cristian الگوریتم

در این حالت یک Time Server داریم که روی ساعت UTC قرار دارد و خودش فرستنده و دریافت‌کننده UTC دارد. به همین دلیل دیگه نیازی نیست که گره‌های موجود تو به شبکه‌ی LAN هم‌شون فرستنده و گیرنده UTC برای هماهنگ‌سازی زمان به واسطه ماهواره ما داشته باشند. طبق شکل زیر کلاینت در زمان T_1 درخواست را به Time Server ارسال کرده که با یک تأخیر زمانی $dTreq$ این درخواست به مقصد میرسند که می‌شود T_2 . حال در زمان T_3 این Time Server می‌داند پاسخ را که حاوی زمان‌های (T_1, T_2, T_3) به سمت کلاینت برمی‌گرداند که با تأخیر زمانی $dTres$ و در زمان T_4 به کلاینت میرسند.

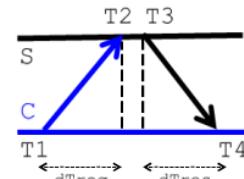


با فرض اینکه تأخیر زمانی از کلاینت به سمت سرور و بالعکس یکسان باشد:

$$T_2 - T_1 \approx T_4 - T_3$$

حال با توجه به شکل زیر و بدست آوردن مقدار Θ می‌توان این عقب یا جلو بودن ساعت کلاینت‌ها را درست کرد:

$$\begin{aligned} \Theta &= T_3 + dTres - T_4 \\ &= T_3 + ((T_2-T_1)+(T_4-T_3))/2 - T_4 \\ &= ((T_2-T_1)+(T_3-T_4))/2 \end{aligned}$$



در صورتی که $\Theta < 0$ یا $\Theta > 0$ باشه زمان کلاینت را به اندازه Θ ثانیه عقب یا جلو می‌کنیم. این روش یه مشکلی دارد و باید حواسمند بشه باشه. مثلا اگر ۵ دقیقه زمان عقب باشه نمایم یه ۵ دقیقه جلو بکشیم و به جاش به صورت تدریجی یا Gradual این هماهنگ‌سازی زمانی را انجام میدیم.

ولی اگر جلو باشیم این عقب کشیدن داستان دارد. این زمان یعنی گذر از یه حالت دیگه یعنی تغییر پس با این اوصاف عقب کشیدن ساعت اگر ما یه بروزرسانی یا رخدادی در سیستم داشته باشیم چار مشکل خواهد شد و اون سناریو ماشین که ساعت ۳ راه افتاده و ۱ رسیده پیش میاد، یعنی تقدم و تأخیر بین رخدادها نمیتوان متصور شد.

پس به صورت خلاصه **عیب** این الگوریتم از این قراره:

اگر من بخوام ساعت رو جلو بکشم که منعی نداره، ولی اگر عقب بکشم و اون وسط رخدادی انجام شده باشه چار مشکل میشیم چون بحث تقدم و تأخیر بین رخدادها از بین خواهد رفت که مسئله‌ی خیلی مهمیه.

اگر بخوام این الگوریتم رو درست کنم میتونم برم اون اتفاقاتی که افتاده رو undo کنم ولی خب خیلی از کارا از جنس transaction نیست که بشه undo کرد. مثل کسایی که میرن عروسی و کلی گل و شیرینی میگیرن و حلقه میخرن ولی بعدا مشخص میشه که بهم نمیخورن، حالا داماد میاد میگه حلقه و گل و شیرینی رو پس بدید. حلقه رو میتونه پس بده ولی شیرینی و گل دیگه مصرف شدن رفته و نمیشه پسش داد.

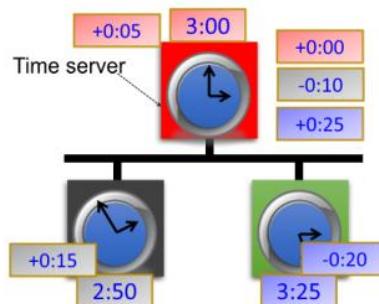
این همه بروزرسانی شده و برچسب زمانی برای بقیه فرستاده شده و این عقب برگرداندن و undo کردن قطعا سیستم رو چار مشکل میکنه.

نکته: این الگوریتم به درد اینترنت نمیخوره و فقط در شبکه محلی یا LAN محدود با کاربرد مشخص به درد نمیخوره.

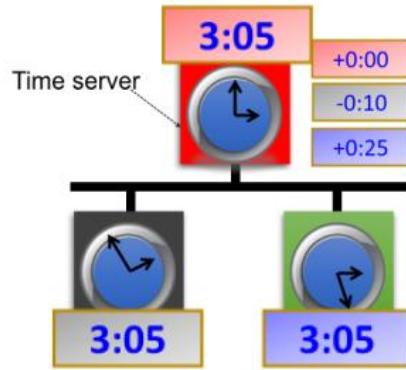
الگوریتم Berkeley

این الگوریتم یک رویکرد توزیع شده برای هماهنگ سازی زمانی هستش. روش قبلی pull-based بود یعنی اینکه ملت به push-based time server درخواست میدادن که زمان رو بگه و خودشون رو باهش هماهنگ کنن ولی این الگوریتم هستش.

- ۱) به صورت دوره‌ای (مثلا ۴ دقیقه) زمان خودش رو برای همه سیستم‌های شبکه broadcast میکنه.
- ۲) کامپیوترها اختلاف زمانی رو محاسبه میکنن و جواب برمنی‌گردن.
- ۳) سرور میانگین اختلاف زمانی رو برای هر کامپیوتر محاسبه میکنه و مجدد برای همه Broadcast میکنه.



۴) تو شکل بالا اون اختلاف زمانی از سمت سرور حساب شده و برای تمامی گره‌ها شبکه ارسال میشه و اونا هم با توجه به این اختلاف زمانی ساعتشون رو عقب یا جلو میکنن (با استفاده از هماهنگ سازی به مرور زمان).



نکته: در این روش دیگه نیازی به گیرنده‌ی UTC نیستش.

نکته: میانگین‌گیری با تحمل پذیری خطا، به صورتی که ساعت‌های outlier در نظر گرفته نمی‌شوند، به راحتی می‌توانن تو Berkeley انجام بشن.

نکته: اگر time server دچار شکست یا fail بشه یکی دیگه می‌توانه به عنوان time server توسط باقی گره‌ها انتخاب بشه.

نکته: این روش هم به درد اینترنت نمی‌خوره و تو LAN با قابلیت محدود بهتر کار می‌کنه.

پروتکل زمان شبکه (NTP)

- در واقع NTP میاد معماری برای یک سرویس زمانی و یک پروتکل برای توزیع اطلاعات زمانی روی اینترنت استفاده می‌کنه.

- در NTP، سرورها در یک سلسله مراتب منطقی به نام Synchronization subnet به هم متصلن.
- سطوح synchronization subnet رو به عنوان stratum ایجاد می‌کنه:

- سرورهای ۱ Stratum باید اطلاعات زمانی دقیقی داشته باشند (متصل به یک گیرنده‌ی UTC)
- سرورهای ۲ Stratum، سرورهای ثانویه هستن که مستقیماً با سرورهای primary ارتباط داره.
- سرورهای ۳ Stratum با سرورهای ۲ هماهنگ می‌شوند.
- ...

عملیات‌های پروتکل NTP

زمانی که سرور زمانی A با سرور زمانی B برای هماهنگ‌سازی زمانی ارتباط برقرار می‌کنه دو حالت رخ میده:

(۱) اگر stratum A <= stratum B اونوقت A با B از لحاظ زمانی هماهنگ نیست.

(۲) اگر stratum A > stratum B، سپس:

- سرور زمانی A با سرور زمانی B هماهنگ هستش.
- یک الگوریتم مشابه الگوریتم Cristian برای هماهنگسازی زمانی به کار گرفته میشه.
- سرور زمانی A میاد stratum A خودش رو بروز میکنه: $1 + \text{stratum A} = \text{stratum B}$

در این حالت من همیشه ساعت رو به سمت جلو میکشم ولی اگر عقب باشه چی؟
کاری که میکنم اینه که نبض خودم رو به مرور کم میکنم تا بهم برسیم.

نکته: هماهنگسازی زمانی دقیقه با زمان UTC

نکته: مقیاسپذیری، سرورهای NTP به صورت سلسله مراتبی برای هماهنگسازی زمانی طبقهبندی شدن، و به مقیاسپذیری به تعداد بالای کلاینت‌ها و سرورها به راحتی پاسخگو هستش.

نکته: این سیستم دارای اطمینانپذیری و تحملپذیری بالای خطاست. به دلایل زیر:

- There are redundant time servers, and redundant paths between the time servers
- The architecture provides reliable service that can tolerate lengthy losses of connectivity
- A synchronization subnet can reconfigure as servers become unreachable. For example, if Stratum 1 server fails, then it can become a Stratum 2 secondary server

نکته: این پروتکل از یک سیستم احراز هویت مبتنی بر پیام هم استفاده میکنه (security)

انتهای جلسه ۱۳

جلسه ۱۴

آقای لمپورت سال ۱۹۷۸ نشون داد که هماهنگسازی ساعت در همه سناریوهای ضروری نیستن. یعنی شاید خیلی وقتاً ما متوجه بشیم که رخدادها به یک ترتیبی در سیستم‌های نوزیع شده اتفاق افتدن برای ما کافیت کنه و نیازی به هماهنگسازی زمان نداشته باشیم.

- اگر دو پردازه با همیگه تعامل نداشته باشن، لزومی نداره که ساعت‌هاشون هماهنگ باش.
- ساعت‌های منطقی به منظور تعریف ترتیبی از رخدادها بدون اندازهگیری زمان فیزیکی که اون رخداد اتفاق افتاده، استفاده میکنه.

با این ایده، آقای لمپورت میاد یک چیزی به نام ساعت منطقی لمپورت (یا به صورت ساده ساعت لمپورت) ارائه میده.

ساعت منطقی لمپورت

لمپورت از حفظ ساعت‌های منطقی در پردازه‌ها برای پیگیری ترتیب وقایع ارائه کرد. برای هماهنگ کردن ساعت‌های منطقی، لمپورت رابطه‌ای به نام «happened-before» (رخ دادن قبل از) تعریف کرد.

- عبارت $a \rightarrow b$ (خوانده شود که a قبل از b رخداد) به این معنیه که همه‌ی موجودیت‌ها در یک سیستم توزیع شده توافق کردن که رخداد a قبل از رخداد b اتفاق افتاده.

عبارت Happened-Before

رابطه‌ی happened-before می‌تواند به صورت مستقیم در دو موقعیت مشاهده شود:

۱. اگر a و b رخدادهایی در یک پردازه یکسان باشند، و a قبل از b اتفاق افتاده باشد، در نتیجه $a \rightarrow b$ درست است. (True)
۲. اگر a یک رخداد پیام (m) ارسال شده توسط یک پردازه باشد، و b رخداد m (یعنی پیام یکسان) توسط پردازه‌ی دیگری دریافت شده باشد، بنابراین $b \rightarrow a$ درست است. (True)

نکته: رابطه‌ی happened-before می‌تواند transitive باشد، یعنی:

اگر $a \rightarrow b$ و $b \rightarrow c$ اتفاق افتاده باشد، آنگاه $c \rightarrow a$ نیز درست است.

مقادیر زمانی در ساعت منطقی

برای هر رخداد a که اتفاق می‌وقتی یک logical time value به نام C_a برای هر رخداد که همه‌ی پردازه‌ها روش توافق دارند در نظر می‌گیریم (C بازم به پردازه ارتباط دارد و نه رخداد، ولی وقتی رخداد اتفاق می‌وقتی بروز می‌شود).

رابطه‌ی زیر نیز برای Time value هر رخداد صادقه:

If $a \rightarrow b$, then $C(a) < C(b)$

خواص ساعت منطقی

از رابطه‌ی happened-before می‌توانیم استنباط کنیم که:

- اگر دو رخداد a و b داخل پردازه یکسان اتفاق افتاده باشند و $a \rightarrow b$, آنگاه $C_a < C_b$ در نتیجه: $C(a) < C(b)$

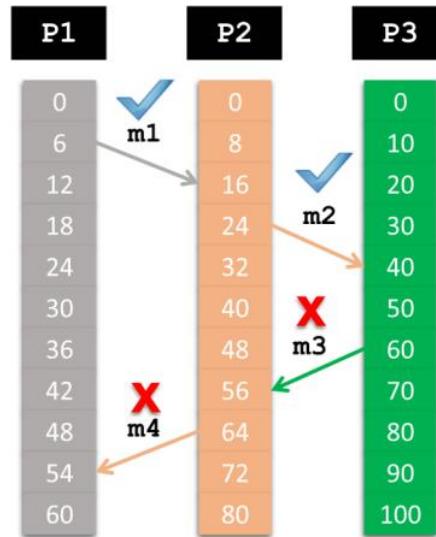
- اگر a رخداد پیام ارسالی m از یک پردازه (مثلا $1P$), و b رخداد m دریافتی (یعنی پیام یکسان) در پردازه‌ی دیگری باشد (مثلا $2P$), سپس:
- مقدار زمانی $C1_a$ و $C2_b$ با توافق دو پردازه به نحوی انجام می‌شود که $C1_a < C2_b$ باشد.
- در این ساعت منطقی همیشه اون C افزایش پیدا می‌کند (جلو میره) و هیچ وقت کاهشی عمل نمی‌کند (عقب نمیره).
- برخلاف راهکارهای Berkley و Cristian

مثالی از هماهنگ‌سازی ساعت‌های منطقی

فرض کنیم که سه پردازه P , $2P$ و $3P$ با نرخ (rate)‌های مختلف بر روی سه ماشین مختلف در حال اجرا هستند.

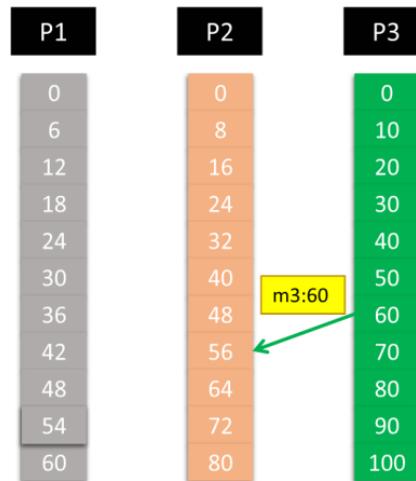
$P1$	$P2$	$P3$
0	0	0
6	8	10
12	16	20
18	24	30
24	32	40
30	40	50
36	48	60
42	56	70
48	64	80
54	72	90
60	80	100

در شکل بالا مشاهده می‌شود که $1P$ در زمان ۶ ثانیه $1m$ را به $2P$ ارسال کرده و در ۱۶ ثانیه بهش رسیده. ولی زمانی که $1m$ به $2P$ رسیده بود توی $1P$ زمان ۱۲ ثانیه است (یعنی ساعت‌هایشون هماهنگ نیست)، با این اوصاف در این مدل ساعت منطقی چون زمان $1P$ زودتر از زمان $2P$ بوده به ریز تایم کاری نداریم و می‌گیم $1m$ در پردازه اولی- *happened-before* پردازه دومی. همینطور برای $2m$ که از $2P$ به $3P$ ارسال می‌شود. تا حالا سناریو این بوده که از زمان عقب‌تر به زمان جلوتر میرفتیم در ادامه عکس زیر سناریو از ساعت جلو به ساعت عقب را در $3P$ به $2P$ و $2P$ به $1P$ ارسال می‌کنیم. طبق شکل نشون میده که در این سناریوی $3m$ و $4m$ در عبارت *happened-before* صدق نمی‌کنند و *false* هستند.



حال آقای لمپورت چطوری میخواود این سناریو رو حل کنه؟

طبق شکل زیر لمپورت میگه اگر ۳P بخواهد پیام ۳m رو در زمان ۶۰ به ۲P ارسال کنه و زمان ۲P از ۳P عقب باشه، باید ۲P زمانش رو به زمان ۳P برسونه. اصلا برای مهم نیست که چقدر عقبه، فقط یک ترتیب منطقی یا logical order میخوایم که رعایت بشه.



هر پیامی که از سمت یه سیستم ارسال میشه یک timestamp رو با خودش حمل میکنه، زمان سیستم میشه:
 $\text{currentTime} = \text{timestamp} + 1$



تا الان فرض میشد که ساعت منطقی ما یک ساعت فیزیکی داره که میایم با کمک اون این پردازه‌ها رو هماهنگ (Sync) میکردیم ولی سناریوهایی هم وجود داره که ساعت منطقی بدون ساعت فیزیکی داشته باشیم. حالا سوال اینه که چطوری زمان رو به یه رخدادی بچسبوئیم که هیچ ساعت جهانی وجود نداره؟

حالت اول

در این حال هر ماشین میتونه یک شمارنده محلی یا local counter داشته باشه:

۱) برای هر دو رخداد موفقی که در P_i رخ بده، C_i یک واحد افزایش پیدا میکنه.

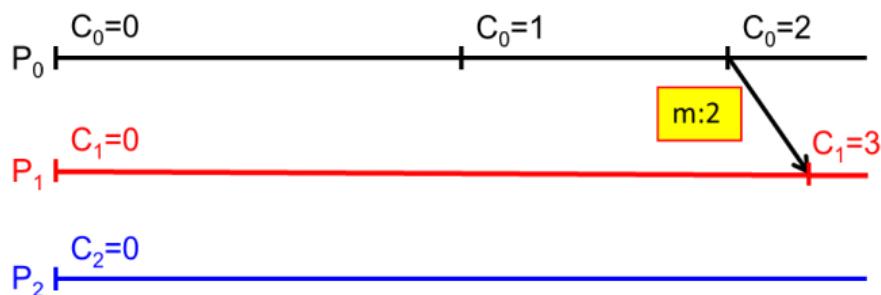
۲) هر بار که یه پیام m توسط P_i ارسال بشه، یک برچسب زمانی به اون m اطلاق میشه یعنی:

$$ts(m) = C_i$$

۳) هر زمان که یه پیام m توسط یک پردازه مثل P_j دریافت می‌شود، P_j شمارنده محلی خودشون یعنی C_j رو به صورت

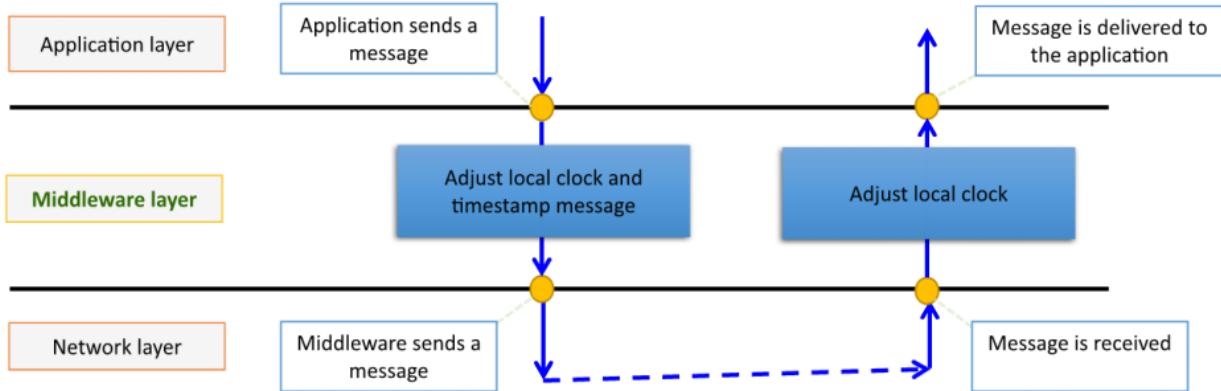
زیر تنظیم میکنه:

$$\max(C_j, ts(m)) + 1$$



حالت دوم

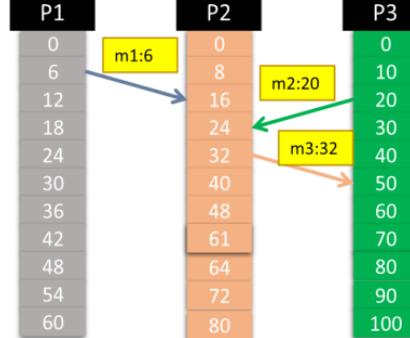
در این حالت به جای اینکه پردازه‌ها رو درگیر هم کنیم، ولی اگر تعداد ماشین و پردازه‌ها زیاد باشه میایم یه ساعت منطقی رو به وسیله‌ی میان‌افزاری به عنوان یه سرویس زمانی پیاده‌سازی میکنیم.



تو شکل بالا اگر لایه اپلیکیشن بخواهد پیامی را ارسال کند، به جای اینکه اونو مستقیم از طریق لایه شبکه به مقصد بده، میاد اول میده به لایه میان‌افزار و اون زمان محلی و برچسب زمانی را تنظیم میکنیم. بعدش میان‌افزار میاد به شبکه پاسخ میده و شبکه مبداء هم برای شبکه‌ی مقصد ارسال میکنه اونم دوباره زمان محلی را میاد تنظیم میکنه و به لایه اپلیکیشن مقصد میفرسته (چون رابطه‌ی a happened-before b برقرار باشد).

حدودیت‌های ساعت لمپورت

- خب اولاً تضمین میکنه که اگر $a \rightarrow b$ باشد در اون صورت $C_a < C_b$ هستش. اما بر عکس نه.
- اگرچه، هیچ اطلاعاتی درباره‌ی هر دو رخداد دلخواه (همروند یا مستقل) a و b، با تنها مقایسه‌ی مقادیر زمانیشون در اختیار قرار نمیده. (**بدی ساعت لمپورت**)
- یعنی اگر برای هر دو رخداد دلخواه a و b که $C_a < C_b$ به این معنی نمی‌باشد که $a \rightarrow b$.



در شکل بالا ۲P نمی‌تونه نشون بده که $m1$ happened before m و $m2$ happened before $m1$. ولی ۲P نمی‌تونه نشون بده که $m1$ happened before m .

نکته: ساعت لمپورت نمی‌تونه ترتیب رخدادها رو تنها با استفاده از مشاهده‌ی مقادیر زمانی دو رخداد دلخواه تضمین کنه.

Vector Clock

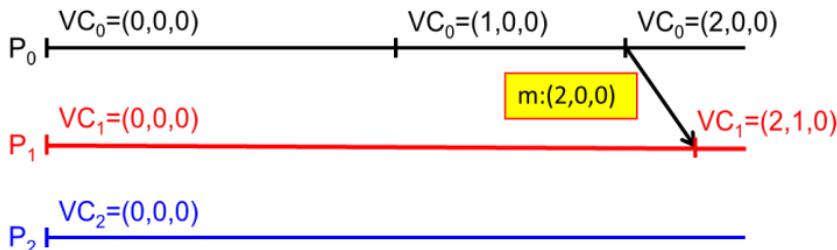
ساعت Vector به منظور غلبه بر محدودیت ساعت لمپورت ارائه شده.

- ویژگی استنباط اینکه a قبل از b اتفاق افتاده با عنوان causality property (علت و معلولی) شناخته میشے.
- یک بردار زمانی یا vector clock برای یک سیستم با N پردازه برابر میشه با یک آرایه عدد صحیح N تایی.
- هر پردازه P_i بردار زمانی VCi خودش رو ذخیره میکنه:
- مقادیر زمانی لمپورت برای هر رخداد در VCi ذخیره میشوند.
- عبارت VCi_a به رخداد a اختصاص داده میشے.

حالا اگر $VCi_a < VCi_b$ اونوقت میتونم ادعا کنم که $a \rightarrow b$ (یا به صورت دقیق‌تر، که رخداد a علت رخداد b است).

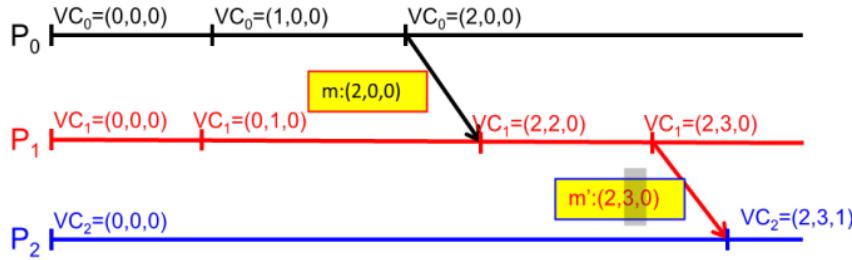
الگوریتم بروزرسانی Vector Clock

- هر زمان که رخداد جدید در پردازه P_i اتفاق بیوافته، مقدار VCi_i رو یک واحد اضافه میکنه.
- زمانی که پردازه P_i یک پیام m به P_j ارسال میکنه:
- مقدار VCi_i یک واحد افزایش پیدا میکنه.
- مقدار برچسب زمانی پیام m رو به صورت ts_m برای بردار i VC_i تنظیم میکنه.
- زمانی که پیام m توسط پردازه P_j دریافت بشه:
- $VCj[k] = \max (VCj[k], ts(m)[k]) ; \text{ (for all } k)$
- مقدار VCj یک واحد افزایش پیدا میکنه.



استنباط رخدادها با استفاده از بردار زمانی

- فرض کنیم که پردازه P_i پیام m رو به پردازه P_j در برچسب زمانی ts_m ارسال کرده، سپس:
- پردازه P_j تعداد رخدادهای در سمت فرستنده یعنی P_i که قبل از پیام m رخداد رو میدونه.
 - تعداد رخدادهای پردازه i آی رو نشون میده ($ts(m)[i] - 1$)

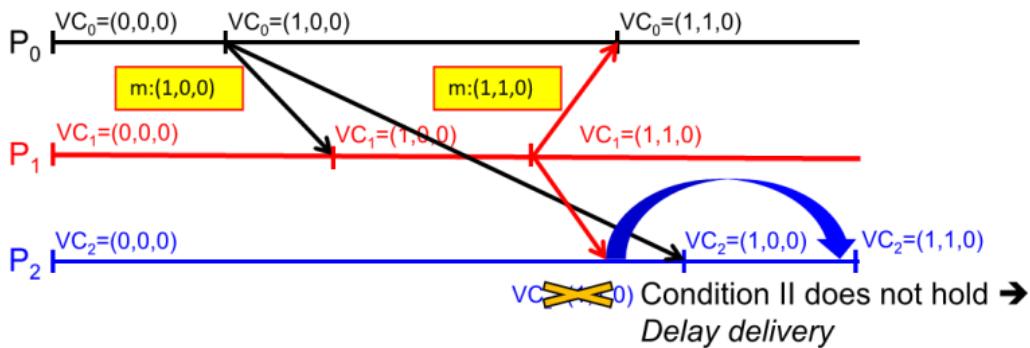


اجرای ارتباط علت و معلولی

فرض کنیم که پیام‌ها داخل یک گروهی از پردازهای P_0, P_1, P_2 به صورت multicast پخش شدن. به منظور اجرای ترتیب علت و معلولی در multicast، پیام m که از P_i به P_j ارسال شده، تا وقتی که دو شرط زیر برآورده نشده باشد منتظر تحویل بموانه:

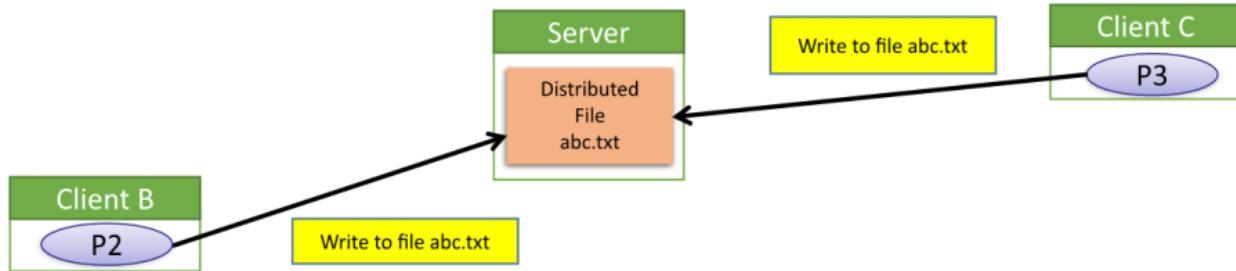
1. $ts(m)[i] = VC_j[i] + 1$ (condition I)
2. $ts(m)[k] \leq VC_j[k]$ for all $k \neq i$ (condition II)

این با تأخیر رسیدن پیام ممکن است در شرایط شبکه رخ بده که نمونه اونو در شکل زیر می‌بینیم.



Mutual Exclusion

پردازهای توزیع شده به یه هماهنگ‌کننده برای دسترسی به منابع به اشتراک گذاشته شده نیاز دارن. برای مثال: مطابق شکل زیر نوشتن یک فایل روی یک فایل سیستم توزیع شده.

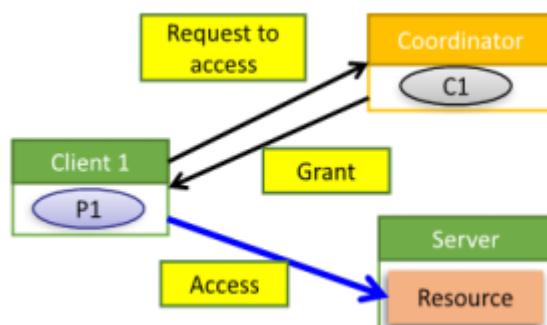


- در سیستم‌های تک پردازنده‌ای، Mutual Exclusion به منظور دسترسی به منابع اشتراکی از طریق متغیرهای اشتراک‌گذاری شده خود سیستم عامل انجام می‌شود.
- البته چنین پشتیبانی به منظور فعال کردن mutual exclusion در سیستم‌های توزیع شده ناکافیه. پس در سیستم‌های توزیع شده این مسئله به واسطه‌ی message passing اتفاق می‌وفته.

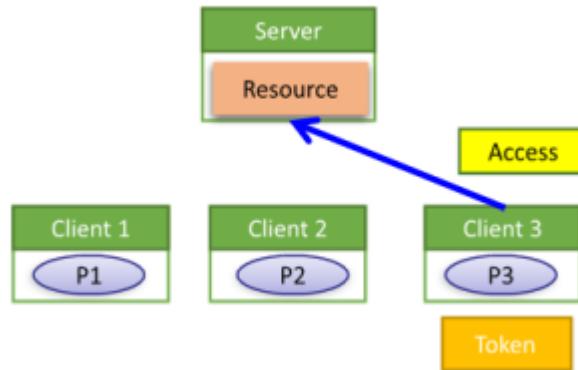
أنواع mutual exclusion توزيع شده

الگوریتم‌های mutual exclusion در دو دسته طبقه‌بندی می‌شوند:

1. راهکارهای مبتنی بر permission
 - a. پردازه‌ای که قصد دسترسی به یک منبع مشترک رو دارد، از یک یا چند هماهنگ‌کننده درخواست اجازه می‌کند.



2. راهکارهای مبتنی بر token
 - a. هر منبع اشتراکی یک توکن دارد.
 - b. توکن بین تمام پردازه‌ها در حال گردش هستش.
 - c. یک پردازه در صورتی که توکن رو داشته باشه می‌توانه به یک منبع دسترسی داشته باشه.



انهای جلسه ۱۴

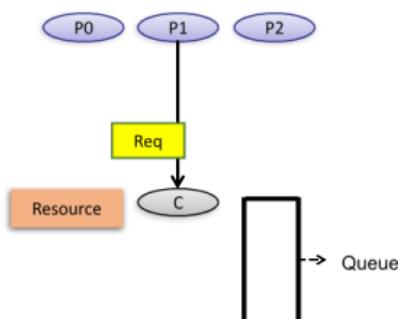
جلسه ۱۵

دو نوع الگوریتم mutual-exclusion مبتنی بر permissioned داریم:

۱. الگوریتم‌های متمرکز
۲. الگوریتم‌های غیرمتمرکز

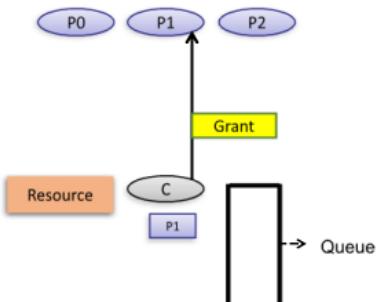
مثالی از الگوریتم متمرکز

- در این الگوریتم خاص، یک پردازه به عنوان هماهنگ‌کننده (C) یک منبع مشترک انتخاب می‌شود.
- هماهنگ‌کننده یک صفت برای دسترسی به درخواست‌ها را هم نگهداری می‌کند.
- هر زمان که یک پردازه بخواهد یک منبع دسترسی پیدا کند، یک پیام درخواست دسترسی به منبع را برای هماهنگ‌کننده یا coordinator ارسال می‌کند.

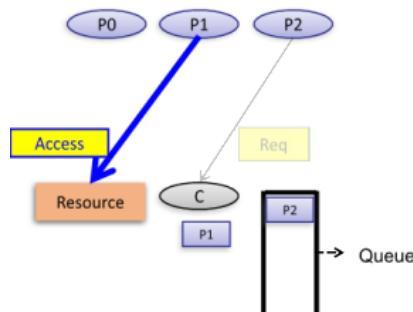


- زمانی که یک هماهنگ‌کننده درخواستی را دریافت می‌کند:

- اگر پردازه‌ی دیگری در حال حاضر در حال استفاده از منبع نبود، اجازه‌ی دسترسی به پردازه را با ارسال یک پیام "grant" بهش ارسال می‌کنه.



- اگر پردازه‌ی دیگری در حال استفاده از اون منبع باشه، در اون صورت هماهنگ‌کننده درخواست رو داخل صف قرار میده و پیامی در جواب درخواست ارسال نمی‌کنه.



- حال اگر پردازه‌ای که منبع رو در اختیار داشت کارش با اون منبع تموم بشه، بعدش هماهنگ‌کننده درخواست رو از توی صف در میاره و یک پیام "grant" برای اون پردازه که درخواست داده بود ارسال می‌کنه.

مزایای روش‌های مرکزی:

- انعطاف‌پذیری (flexibility): درخواست‌های blocking در برابر non-blocking هماهنگ‌کننده می‌توانه درخواست پردازه رو تا زمانی که منبع آزاد بشه بلاک کنه (Blocking).
- یا هماهنگ‌کننده می‌توانه یک پیام "permission-denied" (بدون بلاک کردن درخواست‌کننده) به پردازه ارسال کنه (non-blocking).
- سادگی (simplicity): الگوریتم mutual exclusion را تضمین می‌کنه، و به راحتی می‌شود پیاده‌سازیش کرد.

معایب روش‌های مرکزی

- تحمل‌پذیری پایین خطای (Fault-Tolerance)

- الگوریتم‌های متمرکز در برابر آسیب‌پذیری single point of failure (در سطح هماهنگ‌کننده) شکننده هستند.
- در این حالت پردازه‌ها نمیتوانن تفاوت بین یک هماهنگ‌کننده دچار مشکل و بلاک کردن درخواست به جهت مشغول بودن منبع توسط پردازه‌ای دیگر را تشخیص بدن.
- مشکل performance bottleneck میتوانه داشته باشد
- در یک سیستم با مقیاس بزرگ، یک هماهنگ‌کننده میتوانه در درخواست‌های زیاد غرق بشد.

مثالی از یک الگوریتم غیرمتمرکز

به منظور جلوگیری از مشکلات الگوریتم متمرکز، الگوریتم‌های mutual exclusion غیرمتمرکز ارائه شدن. فرضیات این الگوریتم عبارتند از:

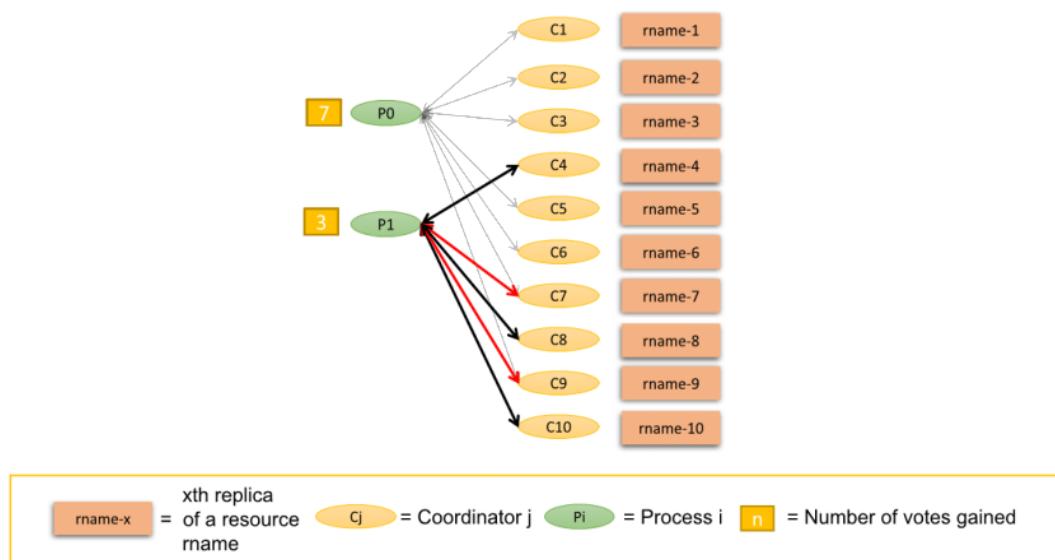
- پردازه‌های توزیع شده در یک سیستم مبتنی بر Distributed Hash Table یا DHT هستند.
- هر منبع n مرتبه کپی شده (replicated)
- کپی آم منبع rname با نام rname- i شناخته میشود.
- هر کپی یه هماهنگ‌کننده برای کنترل دسترسی خودش دارد.
- هماهنگ‌کنندشونم از طریق همین سیستم hashing مشخص میشه.

رویکرد:

- زمانی که یک پردازه میخواهد به یک منبع دسترسی داشته باشد، باید آرای اکثربت از $n > \lceil \frac{m}{2} \rceil$ هماهنگ‌کننده‌ها رو بددست بیاره.

یک مثال:

- If $n=10$ and $m=7$, then a process needs at-least 7 votes to access the resource



مزایای الگوریتم‌های غیر مرکز

- تحمل پذیری بالای خطای در این الگوریتم‌های غیر مرکز فرض می‌شود که هماهنگ‌کننده‌ها به سرعت می‌توانند از یک شکست یا **failure** بپیواد.
- فقط بدیش اینه که وقتی ریکاور شد همه‌ی اطلاعاتی که قبلاً داشت بپره (پس بهتره قبلاً اینا رو لاغ بکنه).
- بنابراین هماهنگ‌کننده ممکن است به صورت نادرست درخواست اجازه به یک پردازه بده.
- رویکرد **mutual exclusion** به صورت قطعی تضمین شده نیست.
- ولی، الگوریتم هنوز به صورت احتمالی **mutual exclusion** را تضمین می‌کند.

تضمين‌های احتمالی در الگوریتم‌های غیر مرکز

حداقل چه تعداد از هماهنگ‌کننده‌ها باید دچار شکست بشون تا **mutual exclusion** را نقض کنند؟

- حداقل $1 + n - m$ هماهنگ‌کننده باید **fail** بشون.

Let the probability of violating mutual exclusion be P_v

- Derivation of P_v

- Let T be the lifetime of the coordinator
- Let $p = \Delta t/T$ be the probability that a coordinator crashes during time-interval Δt
- Let $P[k]$ be the probability that k out of m coordinators crash during the same interval

$$P[k] = \binom{m}{k} p^k (1-p)^{m-k}$$

$$P_v = \sum_{k=2m-n}^n P[k]$$

In practice, this probability is typically very small

- For $T=3$ hours, $\Delta t=10$ s, $n=32$, and $m=0.75n$: $P_v = 10^{-40}$

پروتکل quorum-based

این الگوریتم پیاده‌سازی از یک پروتکل عمومی‌تر که با نام quorum-based شناخته می‌شود است.

ایده‌ی اصلی الگوریتم:

کلاینت‌ها نیازمند منابع باید مجوز بگیرن و این مجوز رو از چند سرور می‌گیرن. اگر درخواست خواندن یا نوشتمن باشه باید از هر دو کوپن یا quorum باید از این سرورها بگیریم که این کارا رو روی این replica data item انجام بدیم.

مثال عملیاتی:

یک فایل سیستم توزیع شده رو در نظر می‌گیریم و فرض می‌کنیم که روی N سرور تکثیر شده.

قانون نوشتمن:

- یک کلاینت باید ابتدا با $N/2 + 1$ سرور (اکثریت) قبل از بروزرسانی فایل ارتباط بگیره.
- به محض اینکه آرا کسب شد، فایل بروز می‌شود و شماره ورژن افزایش پیدا می‌کند.
- این مسئله در همه بخش‌های replica دنبال می‌شود.

قانون خواندن:

- یک کلاینت باید ابتدا با $N/2 + 1$ سرور ارتباط برقرار کند، و از شون شماره نسخه‌ی فایل درخواستی رو طلب می‌کند.

- اگر شماره ورژن همه یکی بود، این باید آخرین ورژن اون فایل بوده باشه.
- مثال: اگر $N = 5$ بوده باشه از ۳ سرور شماره ورژن مساوی ۸ دریافت کنه، این غیرممکنه که ورژن دو سرور دیگه ۹ بوده باشه.

یه مدل دیگهای هم داریم به نام Gifford:

قانون خواندن: یه کلاینت نیاز داره به جمع آوری یک read quorum داره، که مجموعه‌ی دلخواهی از هر N_r سرور یا بیشتر هستش.

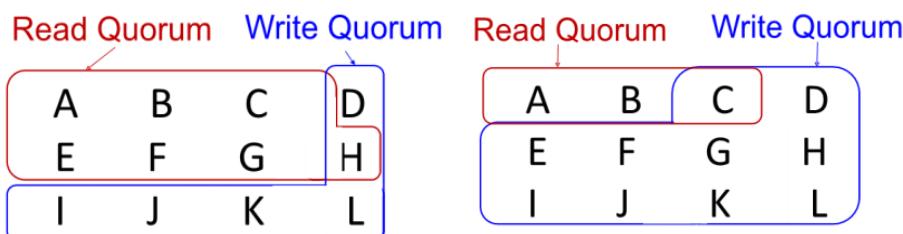
قانون نوشتن: به منظور تغییر یک فایل، یک write quorum از حداقل N_w سرور نیاز هستش.

- مقادیر N_r و N_w دو محدودیت دارن:
- محدودیت ۱ یا $1C$ میگه که: $N_r + N_w > N$
- محدودیت ۲ یا $2C$ میگه که: $2/N_w > N$

ادعا میشه که:

- محدودیت $1C$ از تداخلات read-write یا RW جلوگیری میکنه.
- محدودیت $2C$ از تداخلات write-write یا WW جلوگیری میکنه.

مثال ۱:



$$N_R = 7 \text{ and } N_W = 6$$

$$N_R = 3 \text{ and } N_W = 10$$

$C1: N_R + N_W = 13 > N = 12 \rightarrow$ No RW conflicts $C1: N_R + N_W = 13 > N = 12 \rightarrow$ No RW conflicts

$C2: N_W \nmid 12/2 = 6 \rightarrow$ WW conflicts may arise

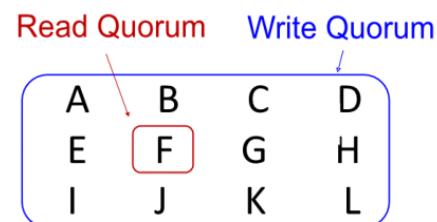
$C2: N_W > 12/2 = 6 \rightarrow$ No WW conflicts

در مثال دوم:

چرا محدودیت $C = 2$ یعنی تداخلات WW نقض میشه؟

- اگر یک کلاینت $\{A, B, C, E, F, G\}$ به عنوان write set انتخاب کنه.
- و کلاینت دیگهای هم $\{D, H, I, J, K, L\}$ رو به عنوان write set خودش انتخاب کنه.
- این دو تا بروزرسانی بدون اینکه تداخل واقعی شناسایی بشه میتوان انجام بشن. بنابراین منجر به یک دید ناپایدار بشه.

مثال ۳:



$$N_R = 1 \text{ and } N_W = 12$$

$$C1: N_R + N_W = 13 > N = 12$$

→ No RW conflicts

$$C2: N_W > 12/2 = 6$$

→ No WW conflicts

یک کلاینت میتونه با پیدا کردن یک replica file از هر کپی دست پیدا کنه.

- Good read performance

یک کلاینت نیاز بdest آوردن کوپن نوشتن یا write quorum روی همه کپیها داره.

- Slow write performance

این مدل بیان کننده یک شماتیکی که به ROWA (Read-Once, Write-All) ارجاع داره رو نشون میده

انتهای جلسه ۱۵

جلسه ۱۶

الگوریتم‌های توکن حلقه

به واسطه‌ی یک الگوریتم توکن حلقه:

- هر منبع به یک توکن مرتبط است.
- هر توکن بین پردازه‌ها در حال گردش است.
- پردازه‌ای که توکن را در دست دارد میتوانه به منبع دسترسی داشته باشد.

چطوری توکن بین پردازه‌های مختلف جابه‌جا میشے؟

- تمامی پردازه‌ها یک حلقه‌ی منطقی که هر پردازه پردازه‌ی کناری خودش را میشناسه رو تشکیل دادن.
- اگر کار یک پردازه با یک منبع تلوم شده باشد، نخواهد به منبع دسترسی داشته باشد:
- توکن را به پردازه‌ی بعدی روی حلقه پاس میده.

مزایا

این روش عملاً داره به صورت قطعی بحث mutual exclusion را برطرف میکنه.

- چون فقط یک توکن وجود داره و بدون اون توکن نمیشه به منبع دسترسی داشت.

روش‌های مبتنی بر توکن چار قحطی دسترسی به منبع نمیشن چون به همه میرسه.

- هر پردازه توکن را دریافت خواهد کرد.

معایب

روش token ring سربار پیامی بالایی داره.

- زمانی که هیچ پردازه‌ای به منبع نیازی نداشته باشد، توکن با سرعت بالایی در حل چرخیدن.

چراگر توکنی گم بشه، باید مجدد باز تولید بشه.

- شناسایی گم شدن توکن سخته (مخصوصاً اگر مقدار زمان بین حضورهای موفق توکن نامحدود باشد).

پردازه‌های مرده باید از داخل لینک به کلی حذف بشن.

- سیستم تحويل توکن مبتنی بر ACK میتوانه به حذف کامل پردازه‌های مرده کمک کنه.

مقایسه میان الگوریتم‌های مختلف Mutual Exclusion

در این بخش به مقایسه‌ی الگوریتم‌های ارائه شده در بخش‌های مختلف پرداخته خواهد شد. فرض میکنیم که:

$n =$ تعداد پردازه‌ها در یک سیستم توزیع شده

برای الگوریتم‌های غیرمتمرکز:

m = حداقل تعداد هماهنگ‌کننده‌هایی که باید برای دسترسی به یک منبع توافق کنن
 k = میانگین تعداد درخواست‌های ایجاد شده توسط یک پردازه به یک هماهنگ‌کننده جهت درخواست یک رای

مشکلات	تعداد پیام‌های مورد نیاز برای یک پردازه به منظور دسترسی و آزاد کردن منبع اشتراکی	تلخیر قل از اینکه پردازه بتونه به منبع دسترسی پیدا کنه (در هر بار پیام)	الگوریتم
Single point of failure	۳	۲	مت مرکز
تعداد زیادی از پیام‌ها	$2mk + m$; $k = 1, 2, 3$	$2mk$	غیر مت مرکز
- توکن‌ها ممکنه گم بشنه - رینگ من ممکنه پاریشن‌بندی بشنه به چند بخش (به دلیل به فنا رفتن چند پردازه)	n تا ۱	۰ تا $n-1$	حلقه توکن

انتخابات در سیستم‌های توزیع‌شده

بیشتر الگوریتم‌های توزیع‌شده نیازمند یک پردازه هستن تا نقش هماهنگ‌کننده رو بازی کنه.

- معمولاً، مهم نیست که کدام پردازه به عنوان هماهنگ‌کننده انتخاب شده.

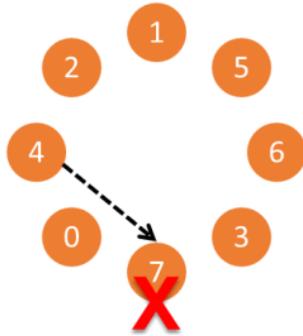
فرآیند انتخابات در یک NutShell

- فرض میکنیم که هر پردازه‌ی P_i میتونه آغازگر الگوریتم انتخابات به منظور انتخاب هماهنگ‌کننده‌ی جدید باشه.
- فرضمون اینه که در پایان انتخابات، هماهنگ‌کننده‌ی انتخاب شده باید منحصر به فرد باشه.
- هر پردازه شناسه‌ی منحصر به فرد (process ID) باقی پردازه‌ها رو میشناسه، ولی نمیدونه کدام پردازه دچار crash شده.
- عموماً، هماهنگ‌کننده‌ی ما باید کسی باشه که بالاترین process ID رو داشته باشه (حالا هر چیز دیگه‌ای رو میشه جایگزین این مورد کرد).
- مثلاً هر کدومی که کمترین بار محاسباتی رو داره.

الگوریتم‌های مختلف انتخابات

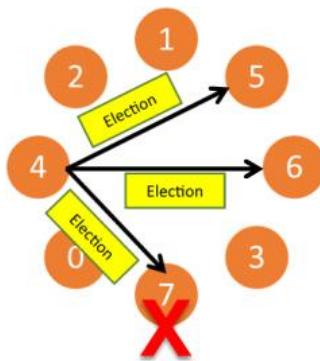
1. Bully Algorithm:

- یک پردازه (مثلاً P_1) زمانی که متوجه بشه که هماهنگ‌کننده‌ی موجود دیگه جواب نمیده، الگوریتم انتخابات رو شروع میکنه.

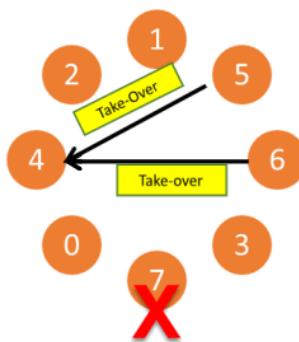


- پردازه‌ی P_i انتخابات رو به نحو زیر دنبال میکنه:

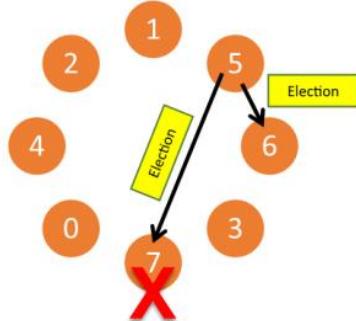
- پردازه‌ی P_i یک پیام با عنوان "Election" به همه‌ی پردازه‌هایی که process ID بیشتر از خودش دارن میفرسته.



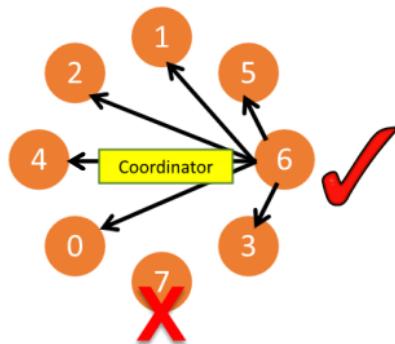
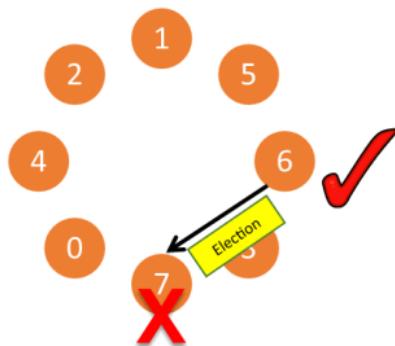
- زمانی که پردازه‌ی P_j با $j > i$ پیام رو دریافت کرد، پیامی با عنوان "take-over" رو در پاسخ برミگردونه. P_i متوجه میشه که کاندیدای مناسبی برای انتخاب شدن نیست و میره کنار.



- پردازه‌ی P_j مجدداً انتخابات رو شروع میکنه و دو قدم بالا ادامه پیدا میکن (هم ۵ و هم ۶ ولی تو این تصویر صرفاً ۵ رو برای نمونه نشون داده)



- حال طبق شکل بالا هم ۵ و هم ۶ پیغام رو ارسال میکنن. ۵ به ۶ و ۷ میفرسته و ۶ هم فقط به هم. حال ۶ به ۵ پیام take-over ارسال میکنه ولی هیچکی به ۶ جوابی برنمیگردونه چون ۷ به فنا رفته. به همین دلیل ۶ میشه coordinator و برندهی انتخابات شده. در انتها هم ۶ به همه پیامی مبنی بر اینکه من از این به بعد coordinator هستم برای همه پردازه‌ها ارسال میکنه.



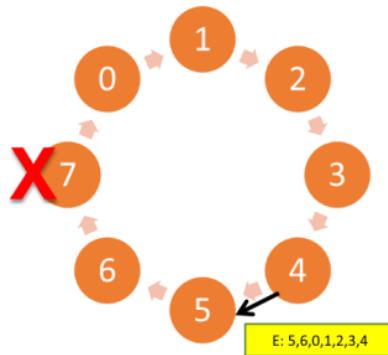
Ring Algorithm .۲

- این الگوریتم عموما در یک توپولوژی حلقه به کار میرن.
- زمانی که یک پردازه‌ی Pi شناسایی میکنه که هماهنگ‌کننده crash کرده، الگوریتم انتخابات رو شروع میکنه.

- پردازه‌ی P_i یک پیام با عنوان "election" درست می‌کنε (E) و برای پردازه‌ی کناری خودش ارسال می‌کنε. همچنین ID خودش رو هم داخل این پیام election ارسال می‌کنε.

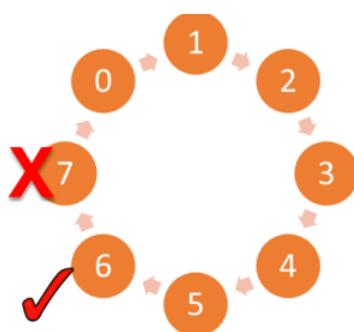
- زمانی که پردازه‌ی P_j پیام رو دریافت می‌کنε، شناسه (ID) خودش رو بهش اضافه می‌کنε و پیام رو فوروارد می‌کنε.

- اگر گره بعدی crash کرده بود، P_j گره زنده‌ی بعدی رو پیدا می‌کنε و همینطوری دست به دست بین همه می‌چرخε.



- زمانی که پیام مجدد به گره شروع کننده انتخابات یعنی در این مثال ۵ برمی‌گردد: - پردازه‌ی P_i , پردازه با بیشترین process ID رو به عنوان coordinator انتخاب مکینه.

- پردازه‌ی P_i پیام رو به یک پیام از نوع "Coordination" تغییر میده (c) و برای همه تو شبکه ارسالش می‌کنε که بگه coordinator کیه.



مقایسه‌ی الگوریتم‌های انتخابات

فرض کنیم که n برابر با تعداد پردازه‌ها در یک سیستم توزیع شده

مشکلات	تعداد پیام برای انتخاب	الگوریتم
--------	------------------------	----------

	coordinator	
سربار بالای پیام	$O(n^2)$	الگوریتم Bully
یک توپولوژی overlay حلقه موردنیازه	$2n$	الگوریتم Ring

در ادامه این کلاس مقاله‌ی WOSP یا The Web of System Performance بررسی شد.

$$\text{کارایی} = \frac{\text{efficiency}}{\text{performance}}$$

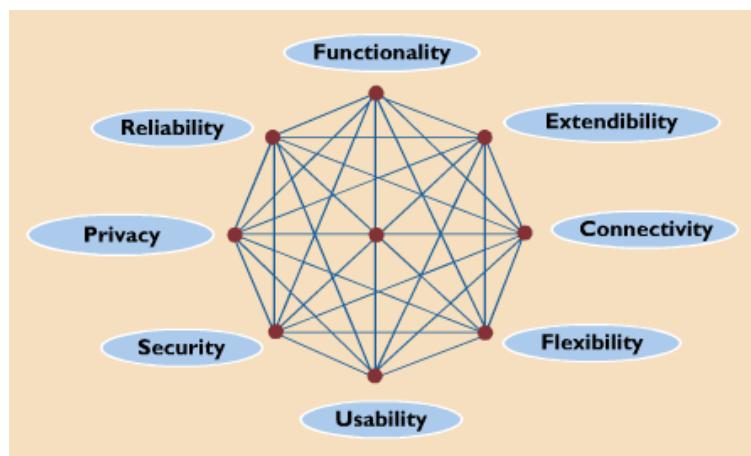
$$\text{کارآمدی} = \frac{\text{performance}}{\text{efficiency}}$$

یکی از ابعاد efficiency میشه

مثال از performance

گوشیم اپل با کلی پردازنده و رم و بند و بساطه به درد یه فرد خیلی مسن نمیخوره یعنی کارآمدی برای اون فرد نداره. کل کاری که اون بخود با اون گوشی انجام بده چند تا تماس و شبکه اجتماعیه. برای کسی که بخود از همه ویژگی هاش استفاده کنه میشه performable یا کارآمد.

این مقاله گفته که میتونه ۸ بعد برای performance در دنیای محاسبات یا information system تعریف کنه:



۱) یکی از وجود performance مقوله‌ی functionality هستش. یعنی اینکه آیا به کارم میاد، مثل همون داستان آیفون که بالا گفته شد. مثلا اگر برای اون فرد مسن بخوایم این وجه رو در نظر بگیریم خب همون اول کار میره کنار، چون به دردش نمیخوره و الکی کلی عملکرد هستش که به کار نمیاد.

۲) ممکنه performance رو توی توسعه‌پذیری یا extendibility در نظر بگیریم. یعنی بتونم هی توسعه‌ش بدم.

۳) وجه دیگر connectivity هستش. این اجزا به هم وصل هستن یا نه. قطع و وصل میشن یا نه.

۴) وجه دیگر flexibility هستش. یعنی سیستم من چقدر منعطفه.

۵) اینکه چقدر usability هستش هم یه فاکتور دیگس. یعنی میتونم ازش استفاده کنم یا نه... خوش دسته یا نه

۶) امنیت

۷) حریم خصوصی

۸) اطمینان‌پذیری یا reliability. سامانه مطمئنه یا نه. کاری که باید رو انجام میده یا نه.

Subgoal	Similar Terms
Extendibility	Openness, interoperability, permeability, compatibility, scalability
Security	Defense, protection, safety, threat resistance
Flexibility	Adaptability, portability, customizability, plasticity, agility, modifiability
Reliability	Stability, dependability, robustness, ruggedness, durability, availability, maintainability
Functionality	Capability, effectualness, usefulness, effectiveness, power, utility
Usability	Ease of use, simplicity, user friendliness, efficiency, accessibility
Connectivity	Networkability, communicativeness, interactivity, sociability
Privacy	Confidentiality, secrecy, camouflage, stealth, social rights, ownership

Permeability:

خیلی راحت می‌توانه نشت پذیر باشه و جابه‌جا بشه

Compatibility:

با عناصر خارجی و شرایطی که دارن تطبیق‌پذیر باشه.

Safety: ایمنی

Security: امنیت

safety و security تفاوت

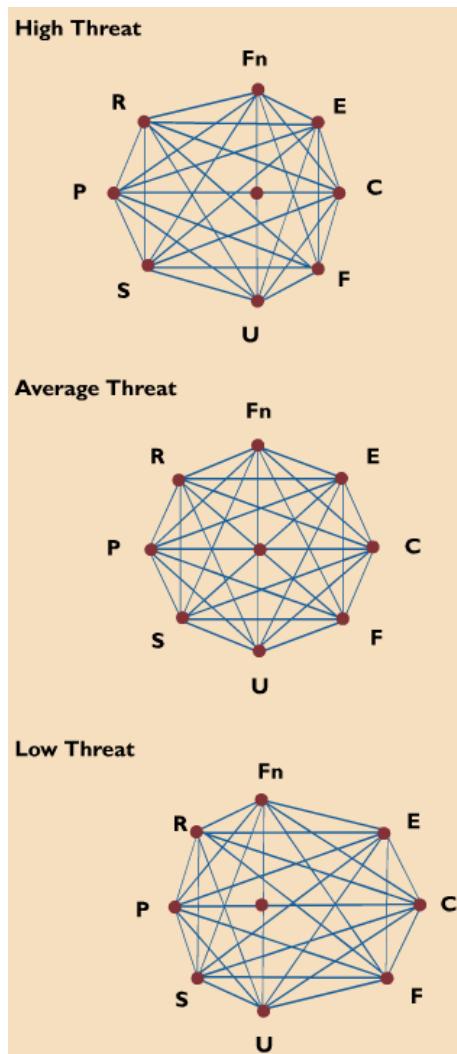
ایمنی یعنی crash نکردن سیستم که ربطی به security نداره. بهتره safety رو وجه جدایی از performance در نظر بگیریم و از زیر پرچم security بیرون بیاریم. همه دغدغه‌ی امنیتی دارن و کسی مشکل ایمنی نداره، چون پول توی security هستش.

Robust: تنومند یعنی بتونه این کارا رو بکنه و تو امنند باشه.

Ruggedness: سرسخت و چغر یعنی میخوره زمین پا میشه و با یدونه فن ضربه فنی نمیشه.

اتصال به اون سیستم و قابلیت همهگیری و اجتماعی پذیری بودن: Sociability

استئار: Camouflage



در گراف بالا اهمیت هر مورد رو میشه متوجه شد.

حالا این همه بحثا به چه دردی میخوره؟

برای اینکه در نهایت تو کاربرد خودش بیایم وزنی به هر موردی که برای ما اهمیت داره بدیم و در مجموع با جمیع مشخصون و رساندنشون به ۱۰۰ درصد به کارآمدی خوبی بررسیم. اگر امنیت مهمه بریم security analysis انجام بدیم و ... اطلاعات کلی در جدول زیر مشخصه.

Goal	Analysis	System Layer	Testing
Extendibility	Interoperability analysis	Import/Add-in	Compatibility
Security	Threat analysis	Security/Log-on	Penetration
Flexibility	Contingency analysis	Configuration/Control	Situational
Reliability	Error analysis	Fault Recovery	Stress/Load
Functionality	Task analysis	Application	Output
Usability	Usability analysis	Interface	User
Connectivity	Network analysis	Communication	Channel/Network
Privacy	Legitimacy analysis	Authorization Rights	Social

اننهای جلسه ۱۵

جلسه ۱۶

Consistency and Replication:

- یکسری موجودیت داریم که میتوانن داده، پردازه، داده + کد و ... باشد و هر موجودیت در سیستم توزیع شده تکثیر شده و نسخه های مختلفی ازش در سیستم وجود دارد و می خواهیم تمامی این نسخه ها (replica) با هم سازگار (consistent) باشند یعنی هر تغییری در هر نسخه روی تمامی نسخه ها اعمال بشه.

چرا به replica نیاز است؟

- در سیستم توزیع شده با تمامی ویژگی هاش که گستردنگی از نظر جغرافیا و تعداد و حجم داره و متفرق هست به نسخه برداری نیاز داره. به ۴ جهت از replica استفاده میکنیم:

1. Improve performance:

- کارآمدی رو افزایش بدیم. عملا منظور این است که کارایی یا efficiency رو افزایش بدیم یعنی سرعت دسترسی به replica ها رو بهبود دهیم.

- مثال هایی برای افزایش کارآمدی با وجود :replica

- در حوزه وب client ها در browser خودشون داده ها رو cache کنند، تا در ارتباط بعدی دسترسی در اسرع وقت صورت بگیره. در این حالت یک client side داریم و ممکنه n نسخه مختلف در server side داشته باشیم.

- مثلًا وقتی میخوایم نام resol کنیم، یعنی نام رو به آدرس فیزیکی تبدیل کنیم از IP Address استفاده می کنیم و میشه IP Address ها رو در سمت client ذخیره و cache کرد و خدمون رو

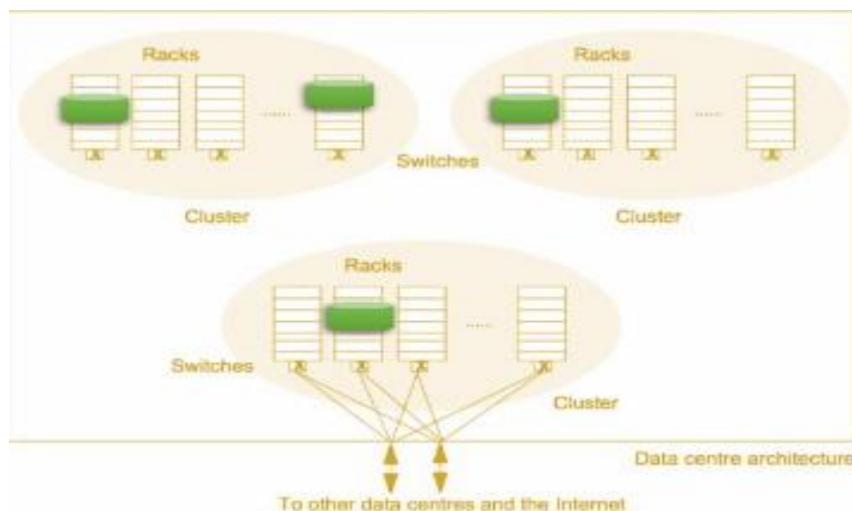
داشته باشیم و اگر امکان پذیر نبود با DNS محلی resol کردن صورت بگیره از لایه های دیگر استفاده کنیم.

- مثل در بحث Content Delivery Network = CDN برای بالا بردن سرعت عمل میشه در جای جای شبکه محتوا cache کرد و در مثلا کاربرد video streaming ازش استفاده کرد.

2. Increase Availability of services:

بهبود دسترس پذیری فراهم میشه و به جای داشتن یک نسخه قراره نسخه های مختلف در سرور های مختلف داشته باشیم.

مثالش میشه Chubby یا Google File System که داده ها در data center های مختلف در cluster های مختلف و در rack های مختلف به صورت replica جایگذاری میکنه و اگر یک data center دچار مشکل شد، بقیه در دسترس هستند. یا اگر یک cluster مشکل داشت در cluster دیگه نسخه موجودیت وجود دارد.



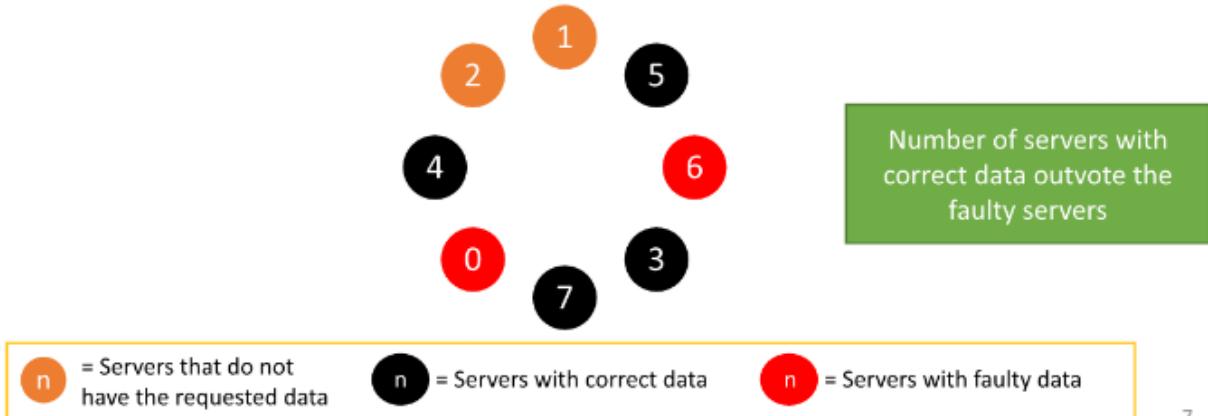
3. Enhancing Scalability of system

بهبود مقیاس پذیری فراهم میشه. اگر یک نسخه داشته باشیم در سیستم Single Point of Failure میشه و مشکل دسترس پذیری و مقیاس پذیری داره.

همه client ها که تعداد زیادی هم هستند اگر قرار باشند به یک نسخه دسترسی پیدا کنند در سیستم ایجاد گلوگاه میشه و با ایجاد نسخه های دیگه load balancer در سیستم توزیع میشه بین سرور های مختلف هر کس با نزدیکترین نسخه کنار خودش ارتباط میگیره و خیلی راحت میشه Scale رو کم و زیاد کرد.

4. Secure against malicious attack

با ایجاد نسخه های متعدد امنیت سیستم های توزیعی در برابر حملات داده ها حفظ میشه.

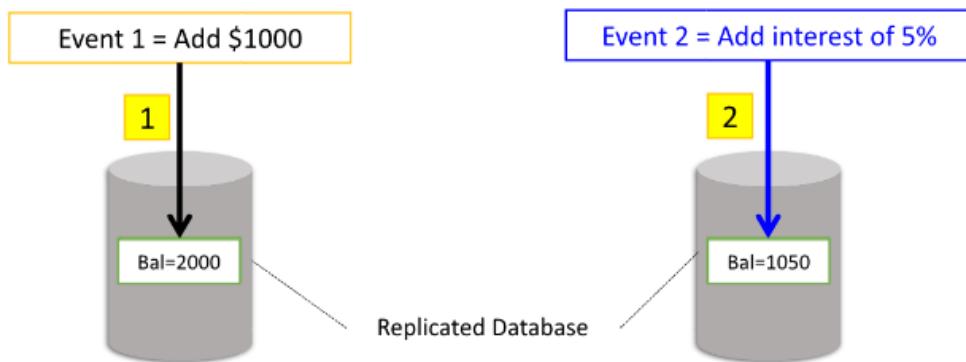


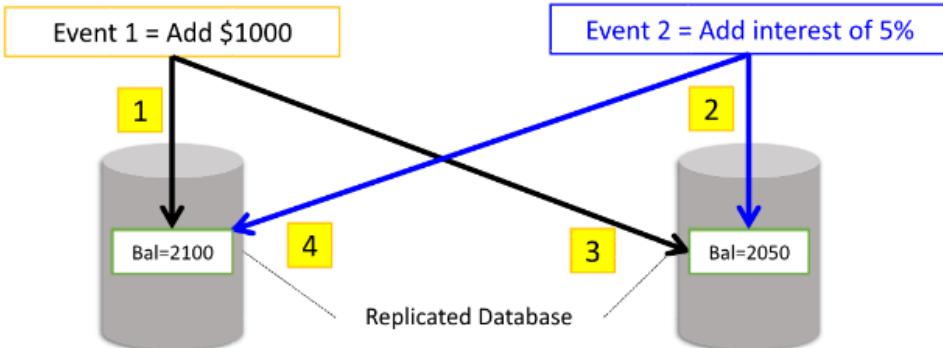
7

- در تصویر بالا ۸ تا سرور داریم که سرور ۱ و ۲ نسخه داده رو در اختیار ندارند و سرورهای مشکی رنگ نسخه سالمی از داده رو در اختیار دارند و قرمز رنگ ها هم نسخه خرابی از داده دارند یعنی یا اطلاعاتشون ناقصه و update های مورد نیاز بهش اعمال نشده و یا مورد حمله قرار گرفته و دادهاش خراب شده.
- در سیستم‌های P2P که فلسفه وجودیش به اشتراک گذاشتن همه چیز هست سعی میشه با coordination که بین گره‌ها صورت میگیره faulty data تحويل داده نشه و تمهداتی اندیشیده بشه. یعنی در شکل گره‌های مشکی با مشارکت هم داده صحیح ارائه بند و اگر درخواستی اوmd از داده‌های گره‌های قرمز استفاده نشه.

سازگاری یعنی چه؟

- اگر چند نسخه از یک چیز داشته باشیم و نسخه‌ها روی کامپیوترهای مختلف هستند و هر از چندگاهی نسخه‌ای روی یک کامپیوتر بروز رسانی بشه، بقیه اگر این بروز رسانی در شون اعمال نشه سازگاری از بین خواهد رفت.

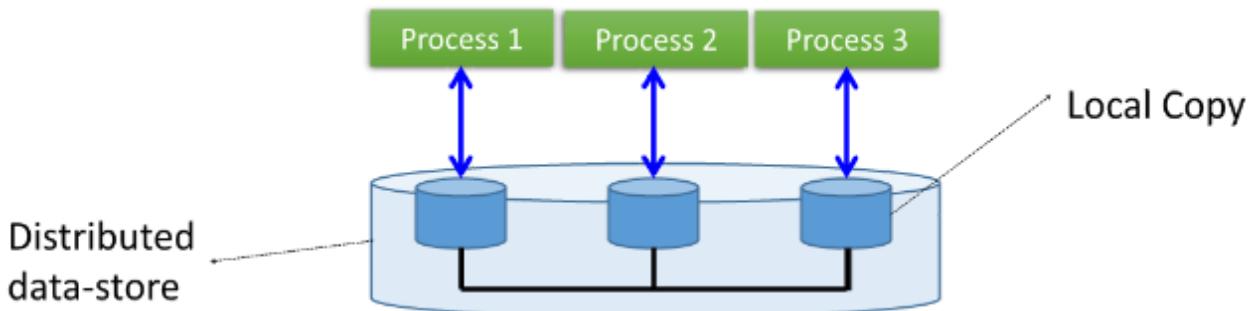




مثال: یک database replicated داشته باشیم که با حساب بانکی مثال زده شده. موجودی اولیه حساب \$1000 هست و رخداد اول که اضافه شدن \$1000 هست روی یک نسخه اجرا بشه و رخداد دوم که اضافه کردن ۵ درصد بهره به موجودی حساب هست روی یک نسخه دیگر اجرا بشه، حالا هر ۲ رخداد بایستی در تمام نسخه‌ها اعمال بشن اما چون ترتیب اجرашون روی نسخه‌ها فرق داره ایجاد ناسازگاری کرده و نسخه‌های مامون consistent نیستند.

مدل‌های مختلف سازگاری = Consistency Model

1. Data Centric Consistency Model = داده محور
2. Client Centric Consistency Model = کاربر محور



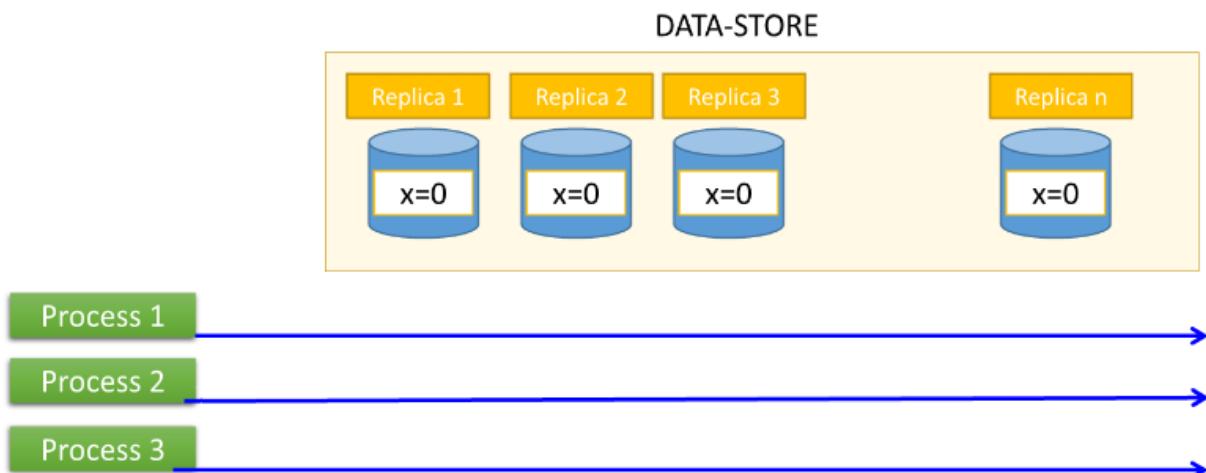
توضیح شکل بالا:

داده‌هایی که قراره به اشتراک گذاشته شود در یک Distributed Shared Memory یا Distributed File System باشند چه در memory یا در database system به همین عنوان Data Store یا Data Storage Unit میگیم

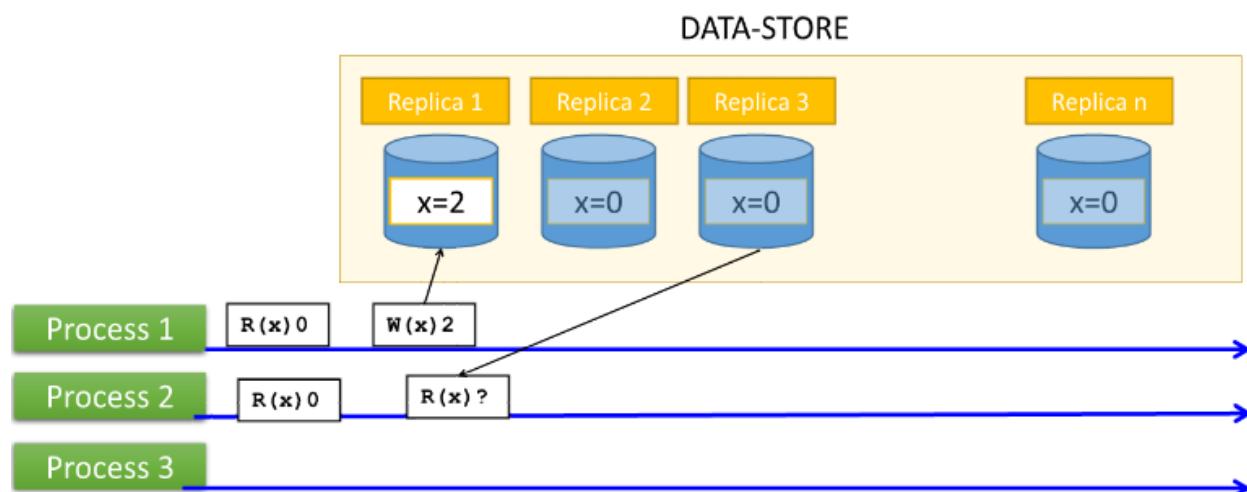
فرض این هست که همه پردازه‌ها به همه این نسخه‌ها حق access دارند.

- هر process در یک محیط توزیع شده دسترسی به data store ماشین خودش و ماشین های دیگر را هم دارد و یک میان افزار وجود داره که دید دسترسی مستقیم رو به پردازه ارائه میده.

:Consistency مثال

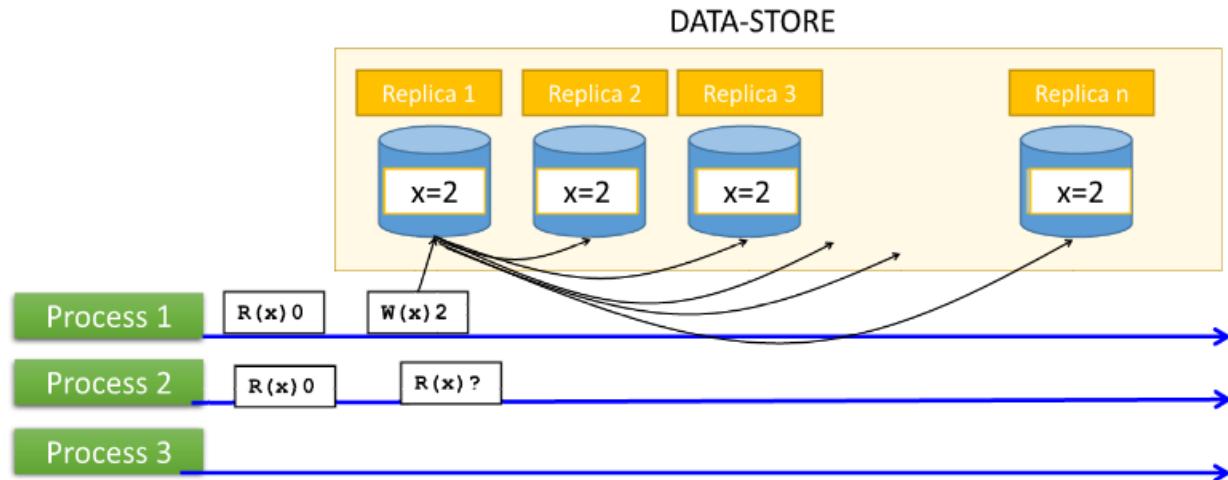


n تا replica داریم و ۳ تا پردازه و خط افق نشون از گذر زمانی هست و چیزی که به اشتراک گذاشته میشه متغیر x هست که در ابتدای کار مقدار یکسان ۰ رو در همه replica ثبت شده.

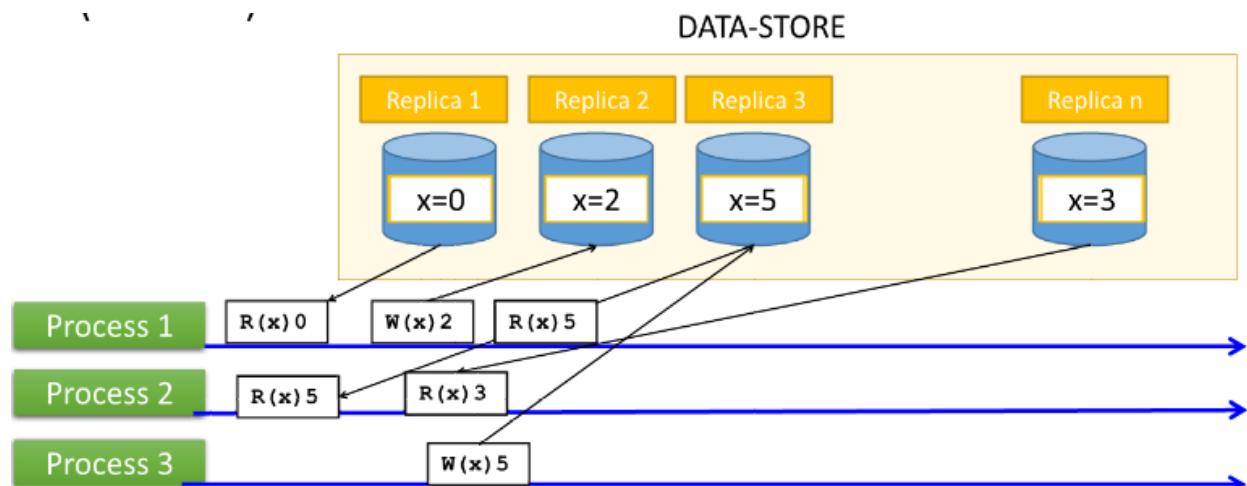


ابتدا پردازه ۱ از یک replica در سیستم، x رو Read میکنه که مقدارش هم ۰ هست. بعد پردازه ۲ از یک نسخه دیگه در سیستم همین کارو انجام میده، بعد پردازه ۱ مقدار x رو به ۲ به روز رسانی میکنه و در replica ۱ در سیستم Write ۱ اش میکنه اما بقیه replica ها هنوز مقدار ۰ رو دارند و به روز رسانی نشدن و سازگار نیستند.

حالا اگر پردازه ۲ بخواهد از یک replica ای در سیستم مقدار x را Read کردنش را معطل بکنیم تا update در replica ۱ روی همه نسخه های دیگه propagate بشد. مطابق شکل زیر:



به این روش **Strict Consistency** سازگاری اکید یا سازگاری محض میگن. یعنی سیستم توزیعی عین یک سیستم غیر توزیعی سنتی تک پردازنده و تک حافظه فارغ از این که از Data unit نسخه های مختلف وجود داره یا نداره به این پایبند هست که هر وقت دارم Read میکنم اون data به روزترین داده هست فارغ از این که pattern بروزرسانی روی نسخه های مختلف چی بوده و چی نبوده.



در تصویر بالا هر replica یک مقداری داره و الزامی برای propagate شدن به روزرسانی ها در تمامی replica ها نیست و به این حالت میگن **Loose Consistency** یا سازگاری غیراکید و شل.

- به صورت کلی یک مصالحه ای بین Strict consistency و Loose consistency وجود داره.
- برای developer Strict consistency خیلی خوبه اما پیاده سازیش مشکل هست و سربار بالایی داره و ممکنه کارایی رو کم کنه و inefficient باشه به این علت که باید update ها رو مدام propagate بکنه و اگر تعداد نسخه ها زیاد باشه این کار خیلی زمانبر میشه.

- پیاده سازی راحتی داره و efficient هست اما program کردنش سخته و منطق برنامه ای رو پیاده کردن در این مدل راحت نیست.
 - برای هر application باید دید update کدام روش به صلاح هست که استفاده بشه. مثلا اگر تعداد Strict باشه سربار فوق العاده بالایی داره و به درد نمیخوره. اگر تعداد Read ها زیاد باشه خوبه چون هم کارا هست و هم ساده پیاده سازی میشه اما اگر Write زیاد باشه میطلبه که به سمت consistency برمی.
- باید دید چه Consistency Model هایی وجود داره و برای پیاده سازی مدل ها چه کار میتوان کرد: عملا یک تفاهم بین پردازه و data store منعقد میشه و تعهدی هست که دو طرف به هم میدن و consistency model نام گرفته و من به عنوان پردازه فرض میکنم ترتیب به این صورتی و تو در نقش data store ضمانت میکنی که همین فرض من رو در ۲ تا operation یعنی Read و Write ارائه بدی.
- ۲ نوع Consistency Model داریم:

1. Data centric Consistency Model:

فرض این هست که این نوع مدل ها تعریف میکنند که چگونه داده هایی که دارند update می شوند در بقیه data store ها باید اعمال شوند یعنی از زاویه دید replica ها به بحث نگاه میکنیم.

2. Client centric Consistency Model:

فرض این هست که هر client میتونه در هر لحظه از یک نسخه متفاوت استفاده کنه و برای سازگاری چه باید بکنه. در اینجا از دید client به قضیه نگاه میکنیم و مهم نیست که کل data store ما consistent باشند. این هست، داده ای که client میخواهد این ویژگی رو داشته باشه. یعنی زمانی که داره از یک replica استفاده میکنه مطمئن باشه که نسبت به قبل به روز هست.

Data centric Consistency Model:

در مدل داده محور سازگاری به این معناست که replica ها رو چجوری سازگار نگه دارم و به روز رسانی رو روش propagate کنم. این قسمت ۲ سرفصل داره:

1. Consistency Specification Model:

تعیین مشخصات. این ها ویژگی نیستند و Spec هستند. یک تعریف هست که کمک میکنه ما سطح سازگاری مورد انتظارمون رو measure کنیم. مثلا Spec یک یخچال ۱۳ فوت هست. چگونگی و how نیست بلکه چیستی و what است. یعنی نحوه پیاده سازیش مهم نیست و حالت functional دارد.

2. Model of Consistent Ordering Operation:

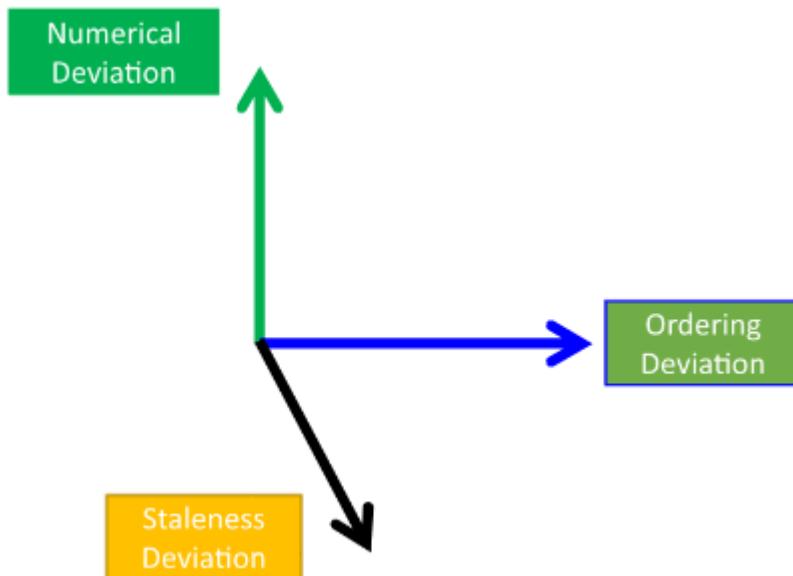
چطور میتونم یک ترتیبی از عملیات read و write رو اعمال بکنم تا specification ارضا بشه.

Consistency Specification Model:

از specification model استفاده میکنیم تا consistency level را تعریف کنیم و بتونیم محکشون بزنیم.

Continuous Consistency Model:

توسط آقای Yu و vahdat ارائه شد که یک چارچوبی ارائه کردند که data store consistency level را در ۳ سطح measure کرد. این ۲ نفر consistency level را تعریف میکنند: برای استخراج specification ها سازگاری در ۳ بعد قابل انجام هست شامل:



1. Numerical Deviation:

ارزش و value در replica ها نسبت به همیگر انحراف دارد یا خیر. انحراف به این جهت قید شده چون منظور tolerable deviation نیست بلکه exact are the same است. مثل: از قسمت یک سهام ۲ تا داریم که تفاوت مقدار اون ها نباید از ۰,۰۲ تجاوز بکند. پس انحراف ۰,۰۲ بالا پایین را سازگار حساب می کنم.

2. Ordering Deviation:

در بحث ترتیب رخدادهای صورت گرفته چه میزان انحراف وجود دارد. مثلًا برای یک اپلیکیشن تا ۶ message هم out of order بود مهم نیست.

3. Staleness Deviation:

یک replica نسبت به update قلی چقدر قدمی تر و کهن‌تر هست. مثل: اطلاعاتی که راجب آب و هوا داریم نباید بیش از ۴ ساعت کهنه باشند. اگر بیش از ۴ ساعت ازش گذشته باشه دیگه update نیست و انحراف دارد.

تعريف Conit

DB میتوانه conit باشد. یک رکورد فایل میتوانه conit باشد، بخشی از یک Conit = Consistency Unit یا کل DB یا چندتا DB با هم میتوان Conit باشند.

به واحد داده ای میگویند که قرار هست Consistency level اش اندازه گیری شود.

برای هر Conit میتوانیم ۳ سطح سازگاری تعریف کنیم:

۱. Numerical Deviation: برای هر replica به نام a ، چندتا update دیگر انجام گرفته که ازشون بی خبر هستم. (نگاه بیرونی دارد)

۲. Order Deviation: برای هر replica به نام a ، چه تعداد update هایی که من local برای این replica انجام دادم هنوز روی سایر replica ها propagate نشده. (دید درونی دارد)

۳. Staleness Deviation: برای هر replica آخرین update ای که روش انجام شده یا شده کی بوده و ببینیم چه قدر کهنه هست. (دید زمانی دارد)

مثال:

Example of Conit and Consistency Measures

Order Deviation at a replica R is the number of operations in R that are not present at the other replicas

Numerical Deviation at replica R is defined as $n(w)$, where $n = \#$ of operations at other replicas that are not yet seen by R, $w =$ weight of the deviation

= max(update amount of all variables in a Conit)

Replica A					Replica B				
x	y	VC	Ord	Num	x	y	VC	Ord	Num
0	0	(0,0)	0	0(0)	0	0	(0,0)	0	0(0)
0	0	(0,0)	0	1(2)	2	0	(0,5)	1	0(0)
2	0	(1,5)	0	0(0)	2	0	(0,5)	0	0(0)
2	1	(10,5)	1	0(0)	2	0	(0,5)	0	1(1)
2	1	(10,5)	1	1(1)	2	1	(0,16)	1	1(1)
3	1	(14,5)	2	1(1)	2	1	(0,16)	1	2(2)
3	4	(23,5)	3	1(1)	2	1	(0,16)	1	3(4)

Replica A	
Operation	Result
<5,B>	x=2
<10,A>	y=1
<14,A>	x=3
<23,A>	y=4

Replica B	
Operation	Result
<5,B>	x=2
<16,B>	y=1

$<5,B> =$ Operation performed at B when the vector clock was 5 $<m,n>$ = Uncommitted operation $<m,n>$ = Committed operation $x;y$ = A Conit

در این مثال Conit برابر با x, y هست.
۲ تا replica داریم به نام های A و B.

VC منظور vector clock است که عدد اولش نشان دهنده click A ای هست که به روز شده و عدد دوم برای replica B هست.

مرحله اول که x, y در هر دو replica با مقدار ۰ مشخص هستند.

در سطر دوم clock replica B میاد توی x متغیر ۵ میکنه ۱، چون من یک تغییری دادم که هنوز در بقیه replica ها اعمال نشده و از سمتی Num در replica A هم میشه ۱ توی پرانتر ۲. اون یک که داره تعداد به روز رسانی هایی که توسط من دیده نشده رو نشون میده و ۲ هم یک وزن هست و برابر هست با \max که در این مثال ۲ هست.

در سطر سوم در clock ۱ برای سیستم A، به روز رسانی replica A برای replica B به نوعی propagate میشه. در سطر بعدی در clock ۱۰ در سیستم A متغیر y مقدارش اپدیت میشه به ۱.

در سطر بعدی در clock ۱۶ خود سیستم B متغیر y رو اپدیت میکنه به ۱. (اینجا propagate نیست بلکه خودش update کرده است)

دو سطر بعدی هم به همین ترتیب رخدادها اتفاق افتاده است.

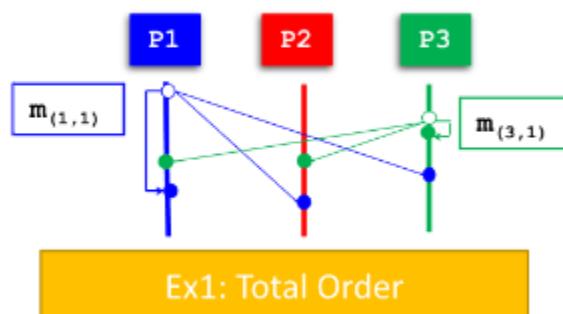
Why is consistent ordering required in replication?

با استنی رخدادها و بروزرسانی هایی که صورت میگیره order بشه و این update ها از طریق ارسال و دریافت پیغام انجام میشه. (message passing) پیغام ها ممکنه پس و پیش بر سه پس نیاز به ordering داریم که ۳ روش برای اون وجود داره:

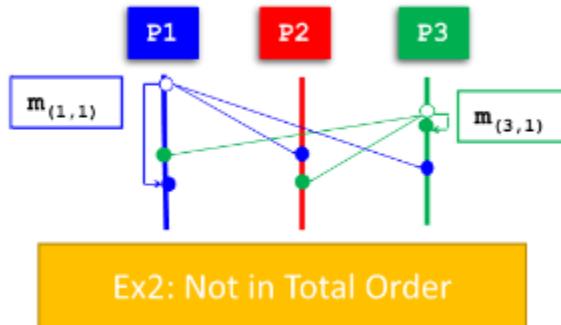
1. Total

بین پردازه و data store به این توافق و تفاهم برسیم که اگر پردازه i پیغام m_i را بفرسته و پردازه j پیغام m_j را بفرسته، اگر پردازه های دیگه m_i رو قبل از m_j بیان deliver بکن، هر پردازه صحیحی هم با استنی m_i رو قبل m_j بیاد deliver (Correct Process) بکنه حتما.

مثال:



در این شکل همه پردازه های P_1 و P_2 و P_3 اول پیام m_1 را دیدند و بعد پیام m_2 و این m_3 Total Order هست.

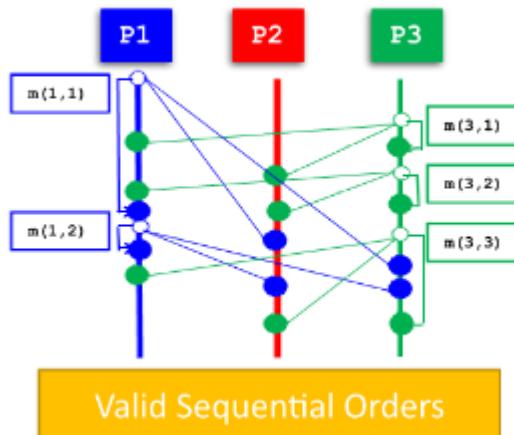


اما این شکل این طور نیست و پردازه P_2 کارو خراب کرده و چلی دیده.

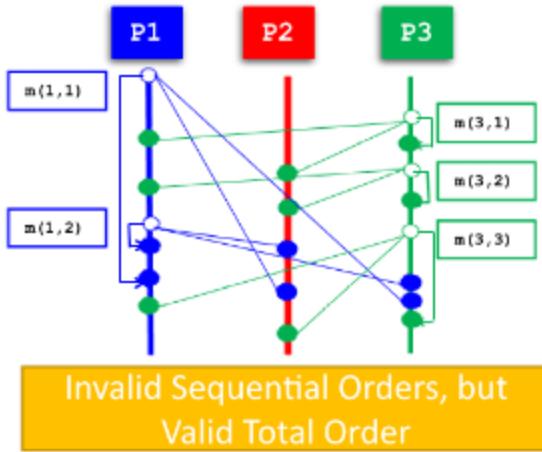
2. Sequential (sequential consistency model):

اگر برای هر پردازه یک توالی از message ها داشته باشیم همه باید این توالی رو به همین شکل ببینن. یعنی فرض کنیم پردازه P_i پیام های $m_{i,1}, m_{i,2}, \dots, m_{i,n}$ را تولید کرده و پردازه P_j پیام های $m_{j,1}, m_{j,2}, \dots, m_{j,n}$ را تولید کرده و این توالی نباید در هیچ پردازه دیگری بهم بخوره.

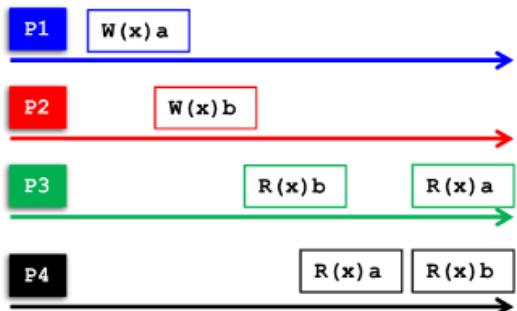
مثال:



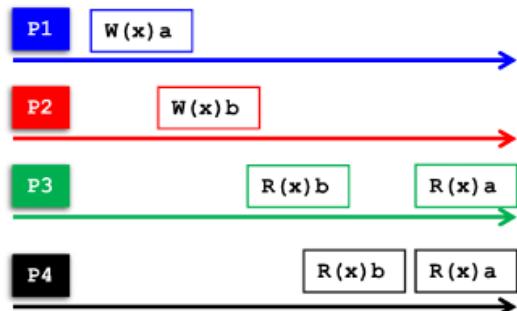
در این تصویر همه پردازه ها هم ترتیب ۳ تا پیام پردازه P_3 را حفظ کردند و هم ترتیب ۲ تا پیام P_1 را حفظ کردند و این حالت sequential order هست.



اما این تصویر هر ۳ تا پردازنده کارو خراب کردند و در مورد دو تا پیام P_1 اومدن اول پیام دومشو دیدن بعد پیام اولشو دیدن و دیگه sequential نیست. (با این که همچون یک جور دیدن اما چون چپ دیدن قبول نیست)
مثال:



(a) Results while operating on DATA-STORE-1



(b) Results while operating on DATA-STORE-2

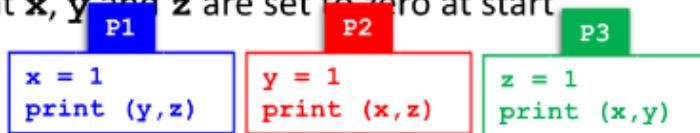
P1 =Process P1 **→** =Timeline at P1 **R(x) b** =Read variable x;
=Result is b **W(x) b** = Write variable x;
=Result is b

در این تصویر شکل سمت چپ sequential order نیست چون پردازه P_3 اول متغیر x را با مقدار b اومده کرده و بعد با مقدار a اما پردازه P_4 بر عکس این کارو انجام داده.
شکل سمت راست sequential هست. سوال: چرا با این که اول مقدار a روی x ما write شده و بعد b روی ما write شده اما ما شکل راست را sequential در نظر میگیریم؟ چون که این دو کار توسط ۲ تا پردازه جداگانه انجام شده و توسط یک پردازه نبوده و اگر هر دو روی یک پردازه انجام میداد اون وقت شکل راست هم sequential نبود.

مثال:

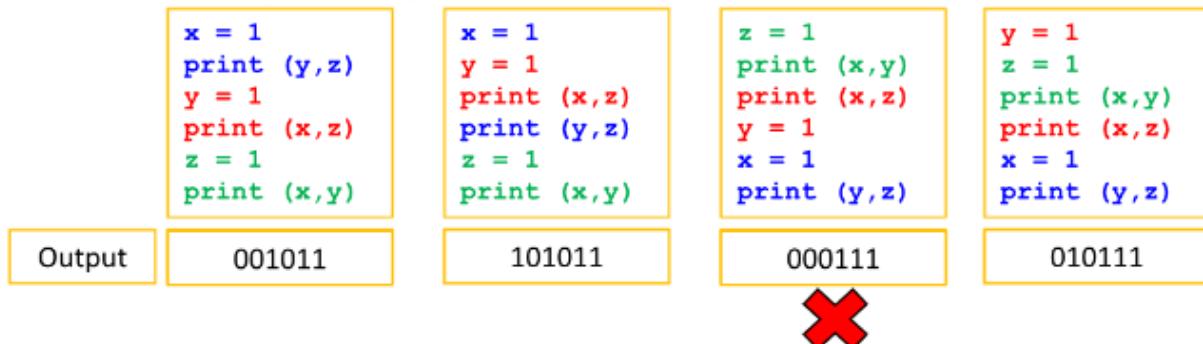
UNIFORM SHARED VARIABLES x , y AND z

- Assume that x , y and z are set to zero at start



- There are many valid sequences in which operations can be exec the replica respecting sequential consistency

- Identify the output



در این مثال هر پردازه ۲ تا عملیات داره که باید ترتیب این عملیات ها حفظ بشه و فقط در حالت سوم جای ۲ تا پیام پردازه P عوض شده و دیگه sequential نیست ولی باقی حالات قابل قبوله.

یعنی برای n تا متغیر در Conit میتونم $!n$ فاکتوریل حالت داشته باشم که یک تعدادیش قابل قبوله و یک تعدادیش قابل قبول نیست و program کردن این حالت نسبت به Strict سخت تر هست.

3. Casual

پایان جلسه ۱۶

جلسه ۱۸

Data centric Consistency Model:

مروری بر بحث جلسه قبل:

یک تعداد داده داریم که به دلایلی همچون افزایش کارایی میخواهیم نسخه های متعددی ازشون داده داشته باشیم. چالش این بحث سازگاری میان این نسخه ها هست. ۲ رویکرد اصلی برای این کار در نظر گرفته شده بود:

Data centric consistency model

Client centric consistency model

در مورد مدل هایی که برای consistency وجود دارد صحبت شد شامل:

- Strict Consistency

در این مدل هر پردازه فارغ از این که چند replica داریم، باید داده ای که میخواهد به روزترین داده موجود باشد.

P1:	W(x)a	
P2:		R(x)a
(a)		

P1:	W(x)a		
P2:		R(x)NIL	R(x)a
(b)			

در شکل بالا قسمت a در شکل strict consistency هست چون پردازه ۱P در متغیر x write مقدار a را کرده و با گذشت زمان پردازه ۲P که متغیر x را خوانده همان مقدار a read کرده است. اما شکل b این strict consistency را ندارد چون پردازه ۲P بار اول که متغیر x را خوانده از replica update ای خوانده که ۱P روی آن propagate نشده بوده و مقدار a را نخوانده. از دید developer بهترین حالت اما از لحظ سربار بیشترین سربار رو بر سیستم متحمل میکنه.

● Sequential Consistency

همه پردازه ها و read ها و write ها را در data storage یک جور ببینند که هر sequence از read و write در اون انجام شد همه پردازه ها یکجور ببینند.

P1:	W(x)a		
P2:		W(x)b	
P3:		R(x)b	R(x)a
P4:		R(x)b	R(x)a
(a)			

P1:	W(x)a		
P2:		W(x)b	
P3:		R(x)b	R(x)a
P4:		R(x)a	R(x)b
(b)			

در شکل بالا قسمت a در شکل sequential consistency هست چون در دو پردازه ۳P و ۴P ترتیب خواندن متغیر x یکسان بوده یعنی هر دو اول b را Read کرده اند و بعد a، اما قسمت b در شکل sequential consistency نیست.

● Linearizability and Sequential Consistency

در این حالت باید update به همون ترتیبی که اتفاق افتاده اند به همون ترتیب توسط همه پردازه ها دیده شوند. با این تفاسیر در شکل بالا نه قسمت a و نه قسمت b ویژگی linearizability and sequential را دارا نیست چون باقیستی حتما در دو پردازه ۳P و ۴P ابتدا a را میخوانندند و بعد b را. هر چه که sequential باشد حتما linearizable هم هست.

Process P1	Process P2	Process P3	
$x = 1;$ print (y, z);	$y = 1;$ print (x, z);	$z = 1;$ print (x, y);	
			در این مثل conit ما شامل ۳ متغیر x و y و z هست و ۳ پردازه داریم که هر کدام عملیاتی به شرح شکل بالا دارند
$x = 1;$ print ((y, z); $y = 1;$ print (x, z); $z = 1;$ print (x, y);	$x = 1;$ $y = 1;$ print (x, z); print(y, z); $z = 1;$ print (x, y);	$y = 1;$ $z = 1;$ print (x, y); print (x, z); $x = 1;$ print (y, z);	$y = 1;$ $x = 1;$ print (z, x); print (x, z); $z = 1;$ print (x, y);
Prints: 001011	Prints: 101011	Prints: 010111	Prints: 111111
Signature: 001011 (a)	Signature: 101011 (b)	Signature: 110101 (c)	Signature: 111111 (d)

در قسمت a اول دستورات P_1 اجرا شده و بعد دستورات P_2 و در آخر P_3 . در ۳ شکل b و c و d هم همین اتفاق افتاده و هر ۴ حالت **valid** هستند چون ترتیب اجرای اجزا در هر پردازه P_1 و P_2 و P_3 به هم نخورده است یعنی مثلا در هیچ کدام نیامده اول **print** صورت بگیره بعد **update**.

• Causal Consistency

$P1: W(x)a$	$W(x)c$		
$P2:$	$R(x)a$	$W(x)b$	
$P3:$	$R(x)a$	$R(x)c$	$R(x)b$
$P4:$	$R(x)a$	$R(x)b$	$R(x)c$

در این حالت اول P_1 آمده x را به مقدار a به روز کرده و بعد P_2 آمده و x را خوانده و بعد آن را به مقدار b به روز کرده. رابطه علت و معلوی اینجاست که **update** صورت گرفته در P_2 منبعث از خوندن a ای است که حاصل از بروزرسانی P_1 بوده است. این روند causal consistency هست چون بعد از این روند دیگر هیچکس x را با مقدار a خوانده است.

P1: W(x)a

P2: R(x)a W(x)b

P3: R(x)b R(x)a

P4: R(x)a R(x)b

(a)

P1: W(x)a

P2: W(x)b

P3: R(x)b R(x)a

P4: R(x)a R(x)b

(b)

در این مثال قسمت a در شکل causal consistency نیست چون $3P$ و $4P$ نباید کلا a را read میکردند و حق خواندن این مقدار را نداشتند. اما شکل b ویژگی causal consistency هست چون update ای که در $2P$ صورت گرفته قبلش read ای نبوده که تغییر منبعث از به روزرسانی $1P$ را فهمیده باشد و بنابراین پردازه ها اجازه دارند مقدار a را هم Read کنند.

- **FIFO Consistency**

همه update هایی که توسط یک پردازه انجام میشے باید به همون ترتیب توسط بقیه دیده بشه.

P1: W(x)a

P2: R(x)a W(x)b W(x)c

P3: R(x)b R(x)a R(x)c

P4: R(x)a R(x)b R(x)c

در این شکل $3P$ اول b رو دیده و بعد c رو دیده و او کی هست یعنی update های پردازه $2P$ را به همون ترتیبی که رخداده دیده در $4P$ همین اتفاق افتاده و OK هست. اما اگر در دو پردازه $3P$ یا $4P$ اول c خوانده میشد و بعد b دیگر FIFO consistency نبود.

<code>x = 1; print (y, z); y = 1; print(x, z); z = 1; print (x, y);</code>	<code>x = 1; y = 1; print(x, z); print (y, z); z = 1; print (x, y);</code>	<code>y = 1; print (x, z); z = 1; print (x, y); x = 1; print (y, z);</code>
Prints: 00	Prints: 10	Prints: 01
(a)	(b)	(c)

در این مثال چون ترتیب عملیات ها در پردازه بهم نخورده FIFO consistency در هر ۳ حالت برقرار هست. این مثال sequential consistency نیست.

Process P1	Process P2
<code>x = 1; if (y == 0) kill (P2);</code>	<code>y = 1; if (x == 0) kill (P1);</code>

در این مثال مقدار اولیه x و y که `conit` را شکل می دهند . بوده و هر پردازه ای اول اجرا شود بعدی را Kill می کند. آیا ممکنه که هر دو پردازه Kill بشن؟ بله اگر هر پردازه از replica ای که update پردازه قبلی را دریافت نکرده استفاده کنه هر دو می توانند هم رو Kill بکنند. یعنی با این که FIFO consistency برقراره یعنی ترتیب باید رعایت بشه اما ایراد نداره که یک پردازه update اون یکی رو نمیبینه و هر دو هم رو Kill بکنند.

FIFO خلی شل و ول تر از sequential هست. چون میگه فقط first in first out داشته باشیم کافیه ولی در sequential باید همه پردازه ها به یک ترتیب ببینند update هارو.

در هر مرحله که consistency رو داریم شل تر می کنیم داریم programmability اش رو سخت تر می کنیم. یعنی در واقع پیاده سازی consistency رو از عهده میان افزار می اندازیم بر عهده developer.

برای راحت تر کردن کار developer از تکنیک sequence variable استفاده میکنند که در اون developer با استفاده از متغیر های همگام سازی یک سری احکام در ک خوش صادر میکنه که میان افزار اون ها رو انجام بده تا مطمئن باشیم اون مدل consistency برقرار باشه.

1. Weak Consistency

```

int a, b, c, d, e, x, y;           /* variables */
int *p, *q;                      /* pointers */
int f( int *p, int *q);          /* function prototype */

a = x * x;                        /* a stored in register */
b = y * y;                        /* b as well */
c = a*a*a + b*b + a * b;        /* used later */
d = a * a * c;                   /* used later */
p = &a;                           /* p gets address of a */
q = &b;                           /* q gets address of b */
e = f(p, q)                      /* function call */

```

در این مثال یک pseudo code داریم و در اون متغیرهای a تا y تعریف شده و دو تا pointer و یک function که به عنوان ورودی دو pointer رو دریافت میکنه. در کد x واکنشی میشه و مقدار a حاصل میشه در یک register اما فعلاً نمیره توی memory مقدار a رو ثبت کنه. همچنین برای b. چون در خط سوم برای محاسبه c بھشون نیاز داشتیم. بعد محاسبه c هم نمیره memory update کنه و عجله نمیکنه چون بازم برای d بھشون نیازه و تو register داریم و نیاز به memory access نداریم. توی e که باید ادرس a و b رو به تابع بده باید به روز رسانی توی memory انجام بشه ولی تا این خط آخر این موضوع رو به تاخیر انداخته بودیم. از نظر کارایی خیلی بهبود داشتیم با این کار.

ایده Weak Consistency همین هست که با sequence variable ها در کد مشخص کند چه موقعي صورت بگیره و چه موقعي نیاز نیست.

P1:	W(x)a	W(x)b	S
P2:		R(x)a	R(x)b
P3:		R(x)b	R(x)a

(a)

P1:	W(x)a	W(x)b	S
P2:			S R(x)a

(b)

در شکل a این مثال P ۱ اول a و بعد مقدار b را در متغیر x به روز رسانی کرده و بعد S که استفاده کرده یعنی **synch** کرده به این منظور که آخرین update من که b باشه رو propagate بکن اما چون ۲P و ۳P چون **synch** نکردن توی **read** هاشون a رو دیدن. پس پردازه ۲P و ۱P در واقع **weak consisten**t نیستند. اما تصویر b که در آن P ۲ اول S رو استفاده کرده **weak** هست.
نکته: برای هر **conit** باید یک **sequence variable** مجزا تعریف می‌کرد.

2. Release Consistency

این روش بیشتر به developer کمک می‌کند.

P1:	Acq(L)	W(x)a	W(x)b	Rel(L)
P2:			Acq(L)	R(x)b
P3:				Rel(L) R(x)a

شیوه بحث locking است. در اینجا اول پردازه ۱P روی **lock conit** ما قرار داده و بعد به مقدار a و b آن را update کرده و بعد lock را Release کرده. تا زمانی که این پردازه Lock اش رو Lock نکنه پردازه دیگری نمیتوانه روی Lock ما Lock **conit** بزاره. پردازه ۲P که lock گذاشته حتماً باید آخرین update ای که روی **conit** اعمال شده رو ببینه که مقدار b هست اما پردازه ۳P کلا حرجی بهش نیست چون اصلاً نیومده lock بزاره و ممکن است update ای رو ببینه و به صورت کلی این مثال **Release consistency** هست. من که Release میکنم یعنی برو propagation رو انجام بده و هر پردازه ای که میخواهد از این آخرین update برخوردار بشه باید lock بزاره مگرنه حرجی بهش نیست.

3. Entry Consistency

P1: Acq(Lx) W(x)a Acq(Ly) W(y)b Rel(Lx) Rel(Ly)

P2:	Acq(Lx)	R(x)a	R(y)NIL
P3:	Acq(Ly)	R(y)b	

اینجا lock قرار دادن ریزدانه نر شده و روی کل conit قفل نگذاشته و روی اجزای آن lock قرار داده. در این مثال وقتی انجام میشه که پردازه بعدی روی اجزایی lock بزاره اما در حالت قبلی propagation وقتی انجام می شد که lock روی conit برداشته و Release می شد. در پردازه ۲P که روی y قفل نگذاشته آخرین اش هم دریافت نکرده است.

خلاصه :

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order

(a)

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered.

(b)

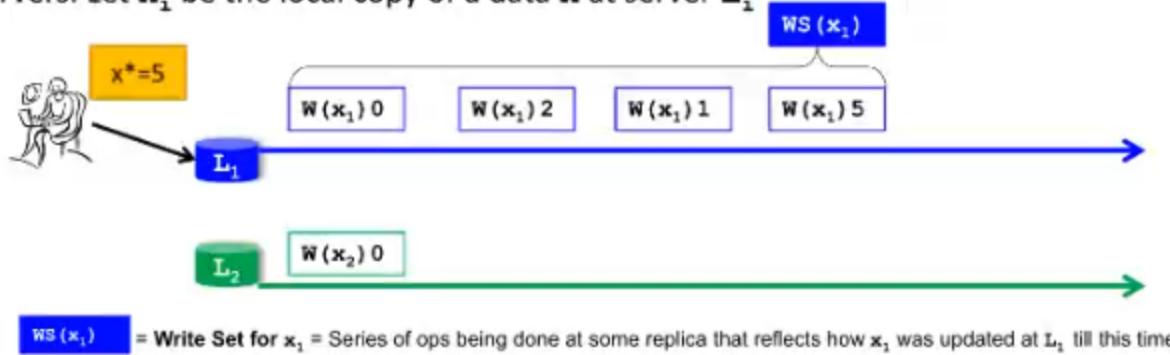
- a) Consistency models not using synchronization operations.
- b) Models with synchronization operations.

منظور از ناحیه بحرانی یا critical region چیست؟ بخشی از کد اجرایی یک پردازه که ممکن است بخواهد از یک منبعی استفاده کند که بین تعدادی پردازنده مشترک باشد.

Client centric Consistency Model

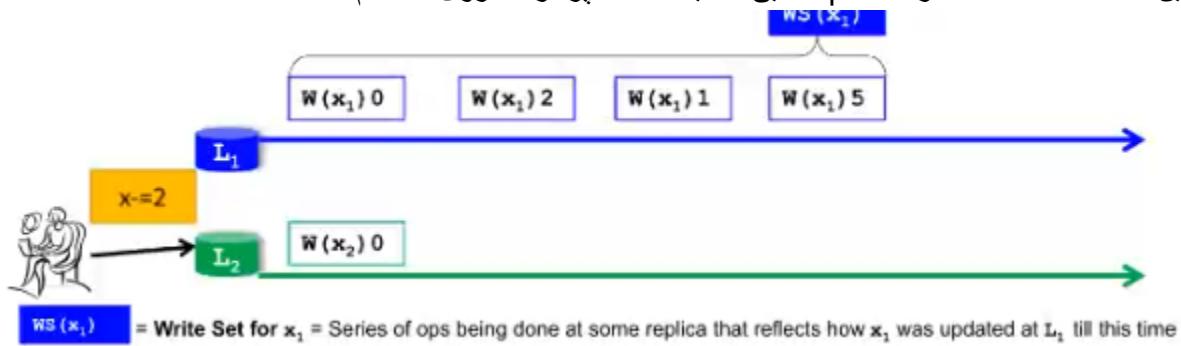
این مدل گارانتی کند که یک client هر وقت به داده access پیدا میکنه داده consistent باشد.

servers. Let \mathbf{x}_i be the local copy of a data \mathbf{x} at server L_i

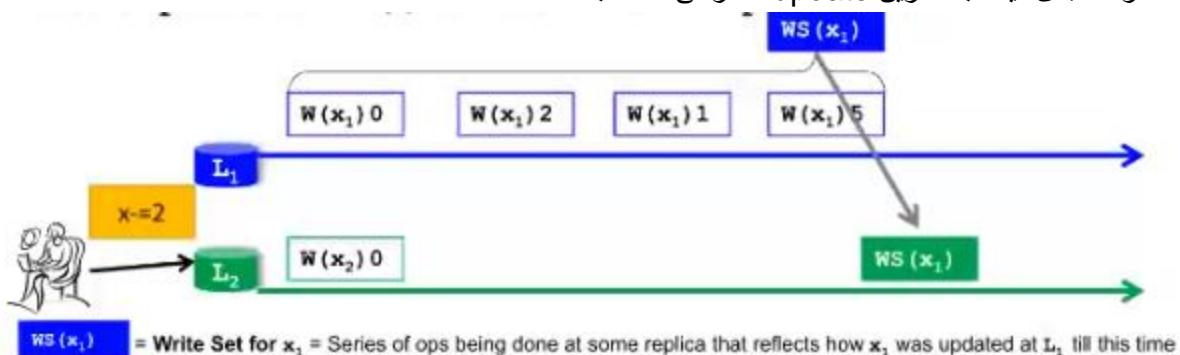


L_i	= Replica i	$R(x_i)b$	= Read variable x at replica i ; Result is b	$W(x)b$	= Write variable x at replica i ; Result is b	$WS(x_i)$	= Write Set
-------	-------------	-----------	--	---------	---	-----------	-------------

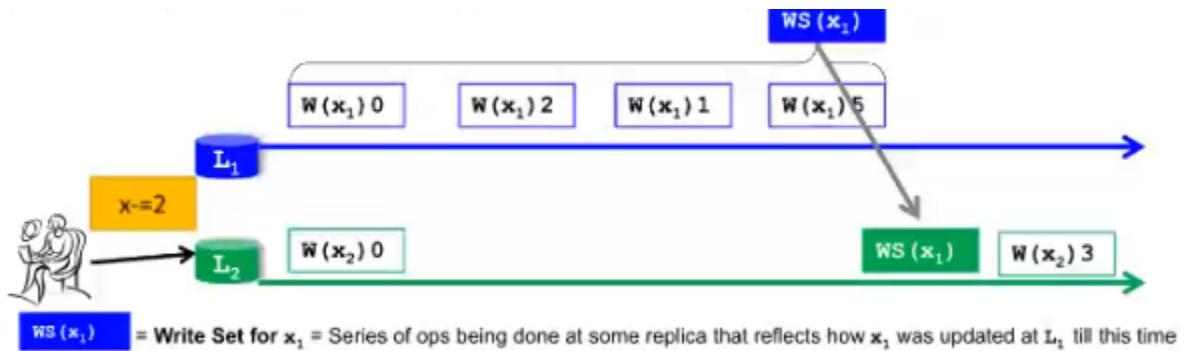
در این مثال update ما از update هایی تشکیل شده که پردازه L روی x انجام داده است.



در replica دوم میخواست x را واحد کم کنے در این update اعمال نشده و من میخواست x را کم کریم. حرفی رفت جای دیگه به آخرین update دسترسی داشته باشه.



یعنی قبل write ما باید read کنیم که x 5 هست. برای این که این اتفاق بیوقتی write set که در قلی بوده باید transmit شود به location جدید و بعد اون میتوانه update صورت بگیره:



٤ نوع مدل برای client centric وجود دارد:

1. Monotonic Read
2. Monotonic Write
3. Read your Writes
4. Write Follow Reads

توضیح این مدل ها حذف هستند. (()

پایان جلسه ۱۸