در جلسات گذشته گفتیم که یک سری الگوها به مرور زمان شکل گرفتند و این الگوها موضوع مورد بحث ما در این درس هستند و لازمه که هر کدام را بشناسیم و مزایا و معایب هر کدام را بدانیم.

سیستم ما میتواند معماری داشته باشد اما لزوما الگوی معماری نداشته باشد ولی استفاده از الگوی معماری میتواند به ما کمک کند که از تجربیات دیگر برنامه نویسان استفاده کنیم. الگوهای معماری که بررسی شد Pipe & Filter، Client-Server، Repository، P2P بود در آخر رسیدیم به معماری Time-Critical System ها.

Time-Critical Systems

A time-critical (real time) system is a software system whose correct functioning depends upon the results produced and the time at which they are produced.

- A hard real time system fails if the results are not produced within required time constraints

  e.g., a fly-by-wire control system for an airplane must respond within specified time limits

- A soft real time system is degraded if the results are not produced within required time constraints

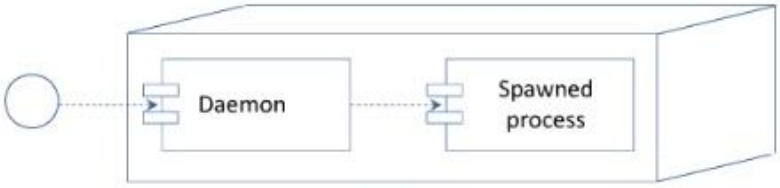
  e.g., a network router is permitted to time out or lose a packet

گفتیم این سیستم ها که به real-time سیستم ها نیز معروف اند، سیستم هایی هستند که عملکرد صحیح آنها باید در یک بازه زمانی کوتاه و مشخص اتفاق بیفتد. یعنی ما توقع داریم که چنین سیستم هایی دریک بازه زمانی مشخص یک response تولید کنند و اگر چنین response ای دریافت نشود این سیستم عملکرد درست خودش را از دست داده است.

این سیستم ها دو دسته بودند : hard real time یا soft real time . سیستم های hard آنهایی هستند که اگر پاسخ مورد نظر در آن بازه زمانی داده نشود یک فاجعه ای اتفاق می افتد اما در سیستم های soft در صورت عدم دریافت پاسخ، عملکرد دچار مخاطره میشود اما شکست وحشتناک و اتفاق ناگواری رخ نمیدهد.

Time Critical System: Architectural Style - Daemon

A daemon is used when messages might arrive at closer intervals than the the time to process them.



Example: Web server

The daemon listens at port 80

When a message arrives it:

spawns a processes to handle the message

returns to listening at port 80

اولین الگویی که در سیستم های Time Critical دیدیم الگوی Daemon بود. Daemon یک مولفه نرم افزاری است که در یک پورتی قرار میگیرد (برای مثال پورت 80) و request ها را تک تک دریافت میکند و وظیفه آن این است که درخواست های مختلف را به مقصد های مورد نظر هدایت کند تا درخواست پردازش شود. گفتیم که این الگوی معماری میتواند در زمینه کاهش زمان پاسخدهی به درخواست ها برای ما موثر باشد و Daemon پس از هدایت یک درخواست مجددا روی پورت گوش میدهد و به درخواست های دیگر پاسخ میدهد.

Architectural Styles for Distributed Data

Replication:

Several copies of the data are held in different locations.

Mirror: Complete data set is replicated

Cache: Dynamic set of data is replicated (e.g., most recently used)

With replicated data, the biggest problems are concurrency and consistency.

Example: The Domain Name System

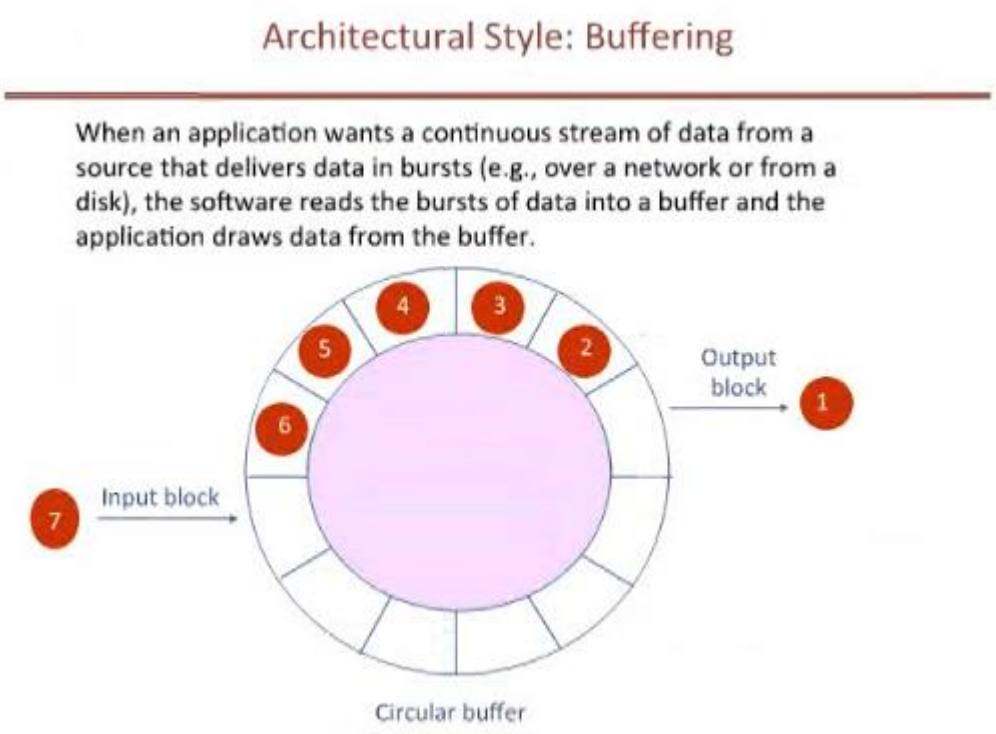
For details of the protocol read:

Paul Mockapetris, "Domain Names - Implementation and Specification". IETF Network Working Group, *Request for Comments*: 1035, November 1987.

http://www.ietf.org/rfc/rfc1035.txt?number=1035

بعد از آن درمورد الگوهای معماری برای داده های توزیع شده صحبت کردیم و گفتیم در اینجور سیستم ها ما میتوانیم دو نوع Replication داده داشته باشیم : Mirroring و Caching. Mirroring برای وقتی است که ما کلا داده را Replicate میکنیم و عینا یک Mirror دیتاهای یک سرور دیگر را در خودش نگهداری میکند. اما Caching زمانی است که ما فقط دیتا هایی که اخیرا استفاده شده را مدنظر داریم و از آنها استفاده مجدد میکنیم.

گفتیم طبق نظریه CAP ما نمیتوانیم چند پارتیشن از دیتا را به صورت Caching داشته باشیم و همزمان هم Availability و هم Consistency را از آن انتظار داشته باشیم. چون Consistency زمانی از بین میرود که write همزمان داشته باشیم و برای حفظ Consistency حتما باید عملیات Locking انجام شود که اگر این عملیات هم انجام شود دیگر Availability نخواهیم داشت. اینها موضوعاتی بود که درجلسه قبل به آن پرداخته شد.



الگوی معماری بعدی Buffering است. در این الگو دیتایی که گرفته میشود در یک حافظه بافر قرار داده میشود تا بتوان در مقابل حجم بالای داده های دریافتی آنها را موقتا در یک حافظه ذخیره کرد و به موقع به هرکدام از آنها رسیدگی کرد. بنابراین مهم ترین مورد استفاده از این معماری پاسخدهی به حجم بالای درخواست ها می باشد.

An Old Exam Question

A company that makes sports equipment decides to create a system for selling sports equipment online. The company already has a product database with description, marketing information, and prices of the equipment that it manufactures.

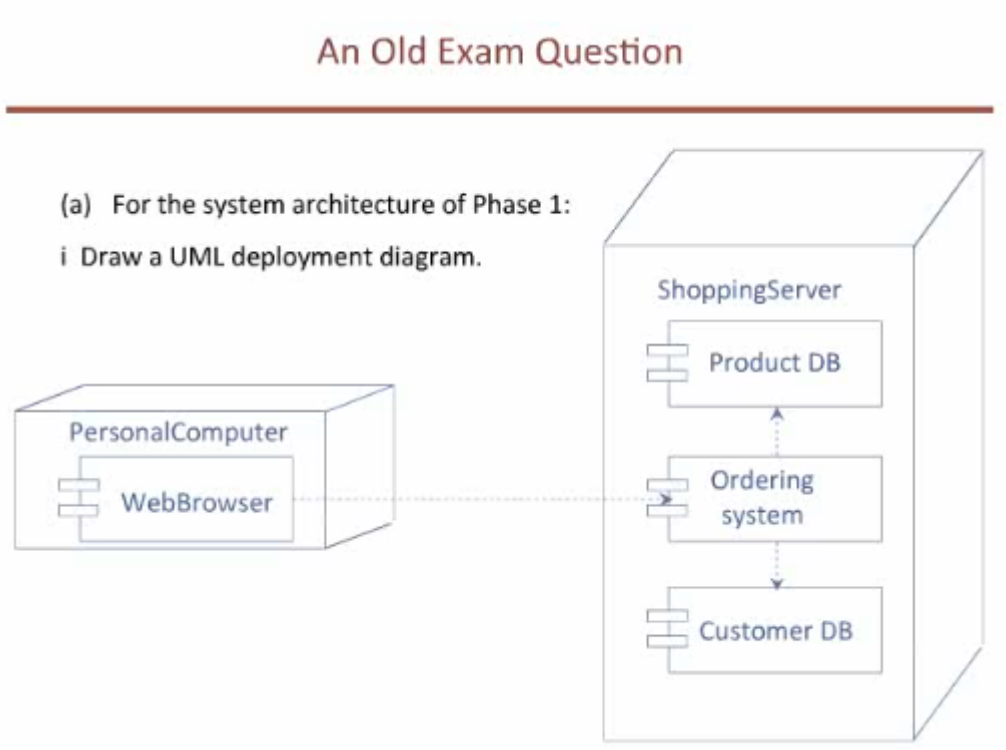
To sell equipment online the company will need to create: a customer database, and an ordering system for online customers.

The plan is to develop the system in two phases. During Phase 1, simple versions of the customer database and ordering system will be brought into production. In Phase 2, major enhancements will be made to these components.

در اسلاید بالا یک نمونه سوال معماری نرم افزار مطرح شده است. سوال این است که یک کمپانی ورزشی، ابزارآلات ورزشی را میفروشد. این کمپانی به دلیل افزایش فروشش نیازمند تولید و توسعه یک نرم افزار فروشگاه آنلاین میشود. این شرکت یک پایگاه داده محصول دارد که در این دیتابیس اطلاعات محصول، قیمت و هر اطلاعاتی که مرتبط با محصول ورزشی باشد در آن قرار گرفته شده است. برای ایجاد این فروشگاه آنلاین نیاز به یک دیتابیس مشتریان و یک سیستم پذیرش سفارش نیز می باشد. این سیستم قرار است در دو فاز توسعه پیدا کند. در فاز اول ساده ترین حالت ممکن قرار است پیاده سازی شود و در فاز دوم، طراحی نرم افزار بهتر و مهندسی تر خواهد شد. (اطلاعات این سوال خیلی کلی است و در امتحان این درس مسئله ای که داده میشود با جزئیات بیشتر است و متناسب با آن اطلاعات بیشتری هم باید برای پاسخ داده شود!)

حال کاری که باید برای این مسئله انجام شود این است که لیست Feature هایی که از ما خواسته شده را بر اساس اطلاعاتی که میدانیم آماده کنیم. Feature های ما Usecase ها هستند و برای هر Usecase ای سناریو میکشیم و برای هر سناریو Sequence Diagram ترسیم میکنیم و براساس Sequence Diagram ، نمودار کلاس که همون معماری سیستم نیز هست استخراج میشود.

خیلی وقت ها هم گفته میشود که از یک الگوی معماری استفاده کنیم. در اینصورت قبل از هر چیز باید با مرور کردن معایب و مزایای الگوهای معماری که میدانیم یکی را که مناسب ترین مورد برای مسئله ما هست انتخاب کنیم.



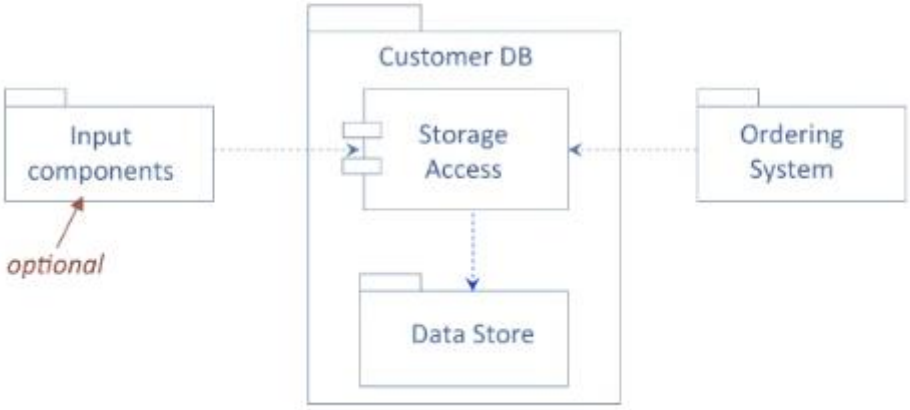
الگوی معماری که برای این مسئله پیشنهاد داده میشود الگوی Repository می باشد که در آن درخواست ها به سیستم سفارش فرستاده میشود و سیستم سفارش نیز بر حسب نیاز به دو دیتابیسی که در اختیار دارد رجوع میکند. این دیاگرام مربوط به فاز یک است (در امتحان یک لایه بیشتر باید توضیح دهیم. یعنی کلاس های هرکدام از این مولفه ها را نیز مشخص کنیم و بعد از آن برحسب ارتباطاتی که بین کلاس ها وجود دارد، ارتباط مولفه ها نیز مشخص شود)

در این طراحی چون سیستم ما دیتا محور میباشد و دو دیتابیس داریم از معماری Repository استفاده کردیم. برای آنکه بدانیم کدام معماری برای برای سیستم نرم افزاری ما مناسب تر است. ابتدا بررسی میکنیم که کدام معماری ها به وضوح غلط هستند (مثلا برای سیستم کافه بازار معماری pipe&filter غلط است) حالا از بین معماری هایی که باقی مانده و احتمال میدهیم که درست باشد، یکی از آنها را با دلایل منطقی که از نظر مزایا و معایب مناسب تر است انتخاب میکنیم در این زمینه ممکن است که پاسخ 100 درصد صحیحی وجود نداشته باشد اما مهم آن است که بتوان با تحلیل معماری و با توجه به شرایط موجود بهترین را انتخاب کرد.

An Old Exam Question

(b) For Phase 1:

iii Draw an UML diagram for this architectural style showing its use in this application.



شکل بالا مولفه های سیستم را بر مبنای معماری Repository نشان میدهد. لایه وسط، لایه مربوط به دیتا می باشد و سیستم سفارش و input component ها تنها با دنیای Storage access آشنا هستند و این لایه Storage access نقش یک interface را دارد که خودش به لایه Data Store متصل است.

ممکن است که بگوییم معماری MVC هم برای این مسئله مناسب است و مزایا و معایبش را بررسی کرد اما وقتی میخواهیم الگوی معماری آن را همچون شکل بالا ترسیم کنیم حتما باید در بستر معماری MVC کشیده شود.

System Design Study 1
Extending the Architecture of the Web

The basic client/server architecture of the web has:

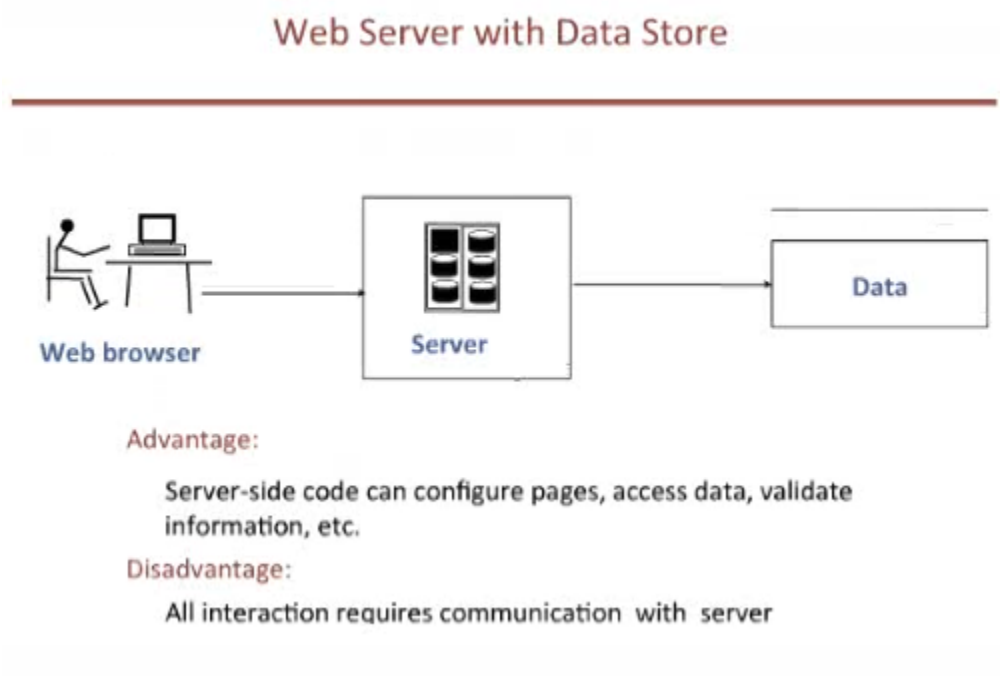
- a server that delivers static pages in HTML format
- a client (known as a browser) that renders HTML pages

Both server and client implement the HTTP interface.

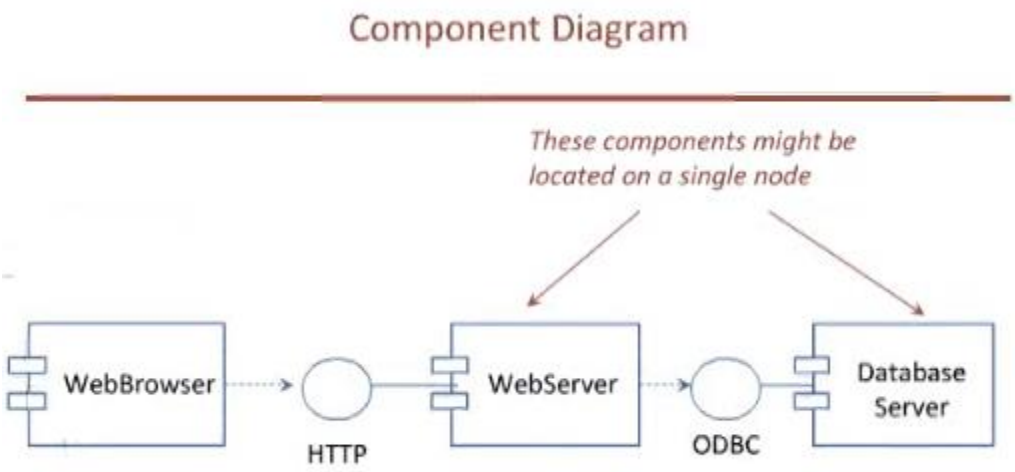
Problem

Extend the architecture of the server so that it can configure HTML pages dynamically.

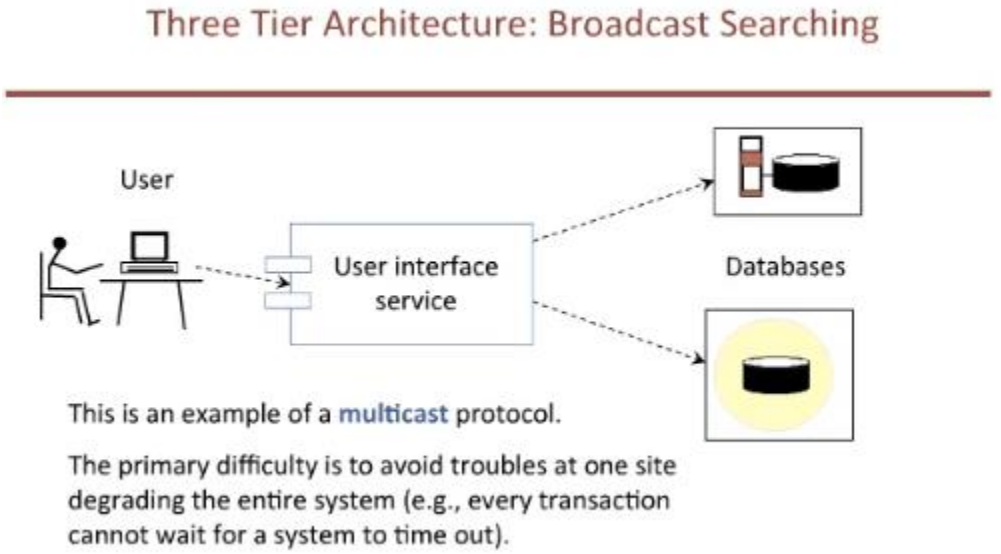
اگر به یاد داشته باشید ما بحث را از یک معماری ساده که یک Web Browser به یک Web Server درخواست میداد و صفحات استاتیک HTML را بر میگرداند شروع کردیم. در این حالت ما دو محدودیت داشتیم اول آنکه همه‌ی صفحات استاتیک بود و دوم آنکه کلاینت برای نمایش یک صفحه حتما باید برای تمام بخش های آن صفحه به سرور درخواست میداد. حال میخواهیم بر اساس معماری هایی که یاد گرفتیم، یک الگوی جدید برای این سیستم پیشنهاد دهیم و بر اساس آن بتوانیم دیتا های داینامیک نیز داشته باشیم.



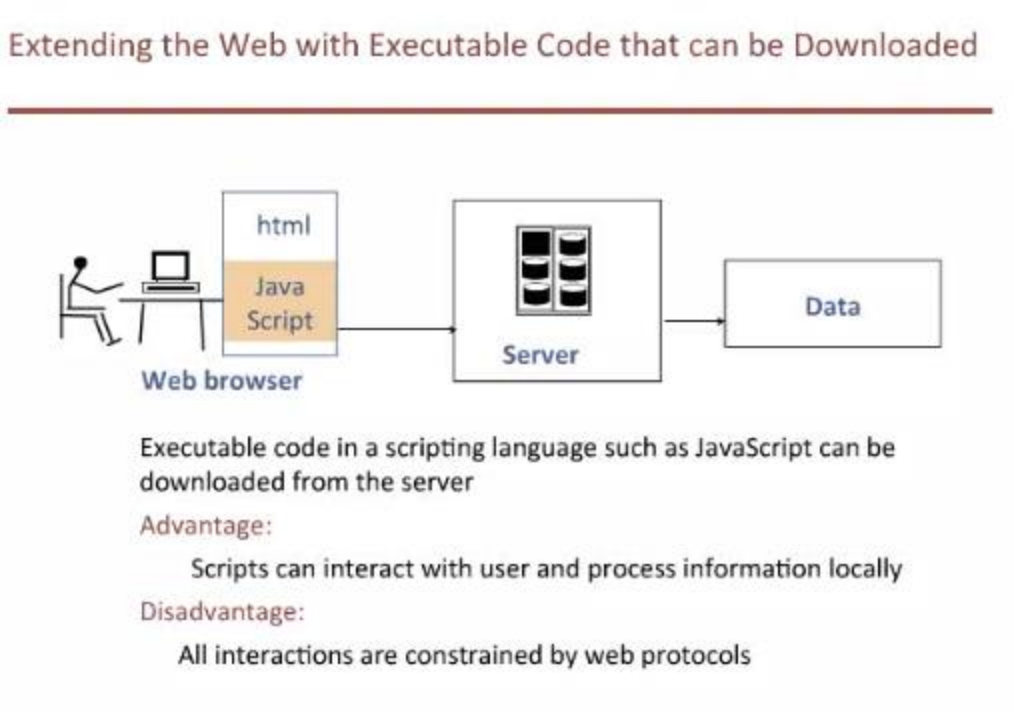
برای داینامیک کردن دیتا باید یک مولفه ای را برای تولید محتوای درخواستی با استفاده از دیتای به دست آمده در نظر بگیریم. به همین علت قسمت دیتا نیز در شکل بالا اضافه شده است. این معماری جدید برای ما این مزیت را خواهد داشت که بتوانیم تولید محتوا را پویا کنیم ، از سمت سرور روی کانفیگ صفحات تغییر ایجاد کنیم و در نهایت بر روی داده ها مدیریت کنیم. اما همچنان آن عیب قبلی که برای دریافت تمام اطلاعات ،ارتباط باید از سمت کلاینت رخ دهد وجود دارد.



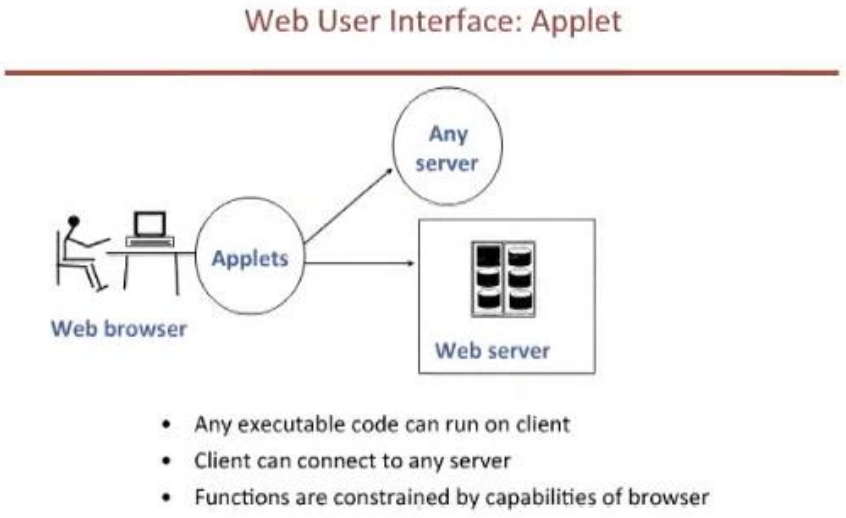
این سیستم را میتوان با یک معماری سه لایه مدل کرد. سمت چپ ترین لایه که همان Browser می باشد لایه UI یا همان Boundary است. قسمت Web server همان لایه Logic و در آخر Database server همان لایه دیتا می باشد. ارتباطات میان این لایه ها توسط اینترفیس برقرار میشود که همان HTTP و ODBC است.



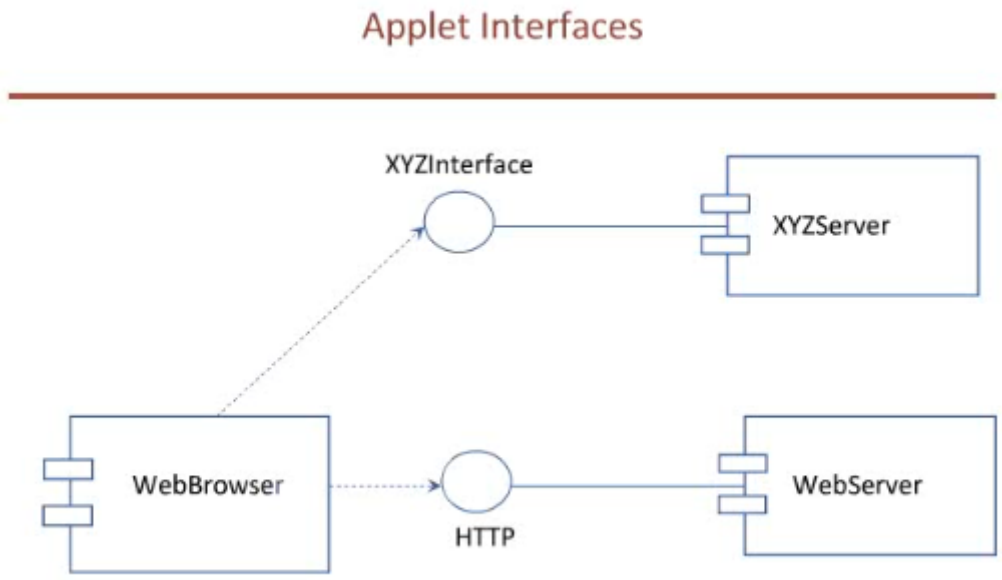
ما میتوانیم چند الگوی معماری را نیز برای همان مسئله ترکیب کنیم و به نتیجه بهتری برسیم. برای مثال میتوان چند دیتابیس استفاده کرد و از الگوهای Replication ،Caching, Daemon, Buffering نیز استفاده کرد. در این صورت پروتکل ارتباطی به Multicast تغییر خواهد کرد چرا که داده قرار است به چندین مقصد مختلف فرستاده شود. بنابراین الگوهای معماری نیز میتوانند به راحتی با هم ترکیب شوند.



حال میخواهیم مشکل باقی مانده را حل کنیم و آن این است که تمام درخواست ها باید به سرور فرستاده شود و نمیتوان در سمت کلاینت کار خاصی را انجام داد. برای حل این مشکل میتوان تکنولوژی های کلاینتی نظیر Ajax و زبان Javascript استفاده کرد که بتوان در سمت کلاینت نیز کد اجرا کرد (برای مثال validate کردن نام کاربری و رمزعبور). این کار باعث میشود که thin client ای که داشتیم به یک fat client تغییر کند. مزیت این تغییر کاهش بار پردازشی سرور (با انتقال بخشی از پردازش قابل تحمل به کلاینت) می باشد و عیبی که ایجاد میکند ضرورت استفاده از یک پروتکل برای ارتباطات می باشد (مانند HTTP).



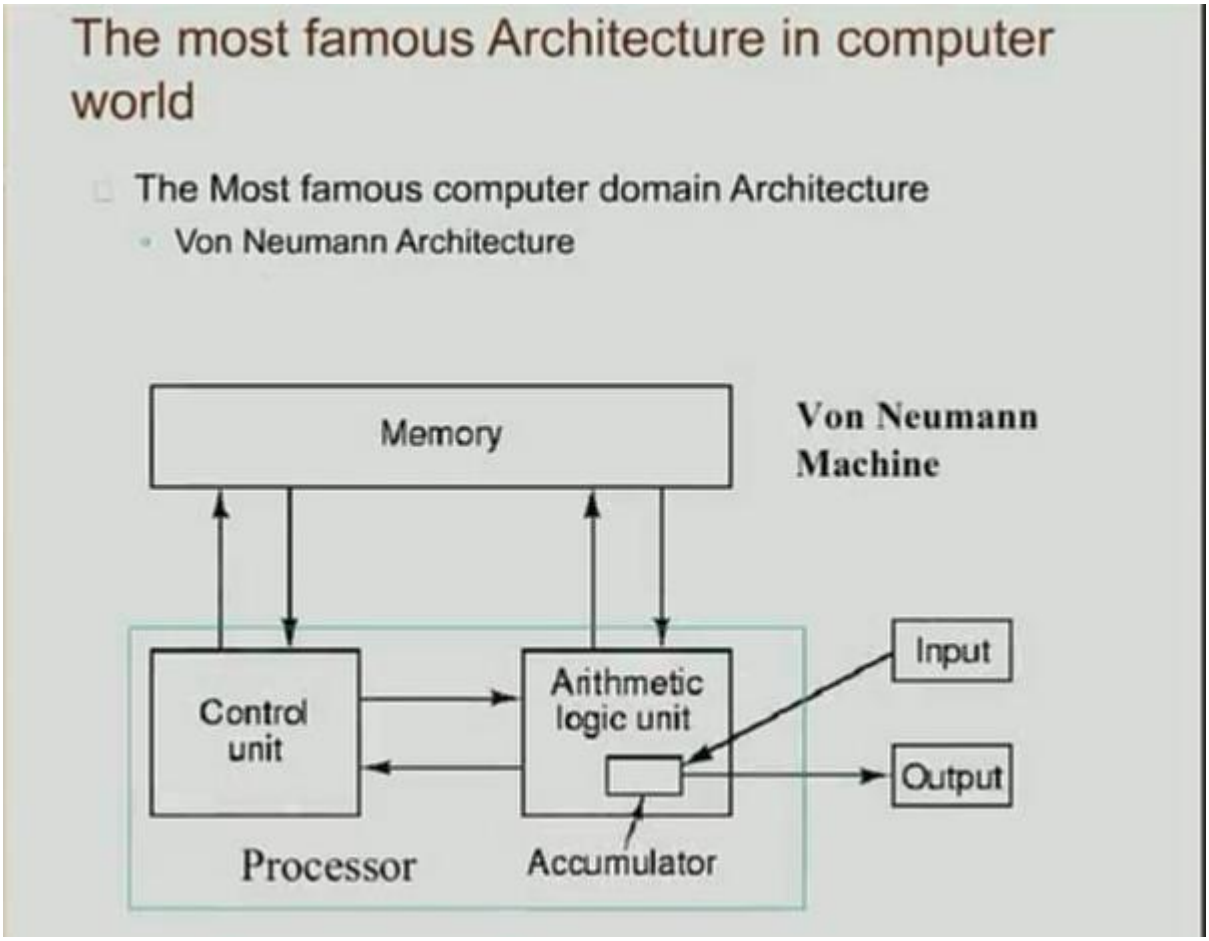
برای آنکه بتوان از چند پروتکل استفاده کرد میتوان تکنولوژی های متفاوتی را به کار برد. در اینجا تکنولوژی Applet استفاده شده است که میتواند توسط اینترفیس های مختلف با سرور صحبت کند.



امروزه به جای Applet ها از تکنولوژی های جدیدتری مانند Microservice ها، Server component ها و ... استفاده میشود که میتوانند با اینترفیس های مختلفی تعامل برقرار کنند.

تا به اینجای کار ما توانستیم الگوهای معماری مختلف را با ذکر مثال web browser – Web server بررسی کنیم.

میخواهیم با جزییات بیشتر و با تمرکز به الگوهای قبلی و جدید بپردازیم.



معماری یعنی چی ؟ مولفه ها و اجزا و ارتباط بین آنها را معماری میگویند.اولین معماری که در سیستم های کامپیوتر معرفی شد و بیس همه معماری ها ست که این معماری پایه و اساس سیستم های امروزی است به نام Famous Neumann میشناسیم که این معماری را در درس معماری سیستم های کامپیوتر دیده اید در اون درس بهتون گفتند که ما در یک سیستم کامپوتری یکCpu داریم که Processor یا پردازنده مرکزی است و شامل دو بخش اصلی ALU(Arithmetic logic Unit) که کار محاسبات و منطق در آن رخ میدهد و بخش دیگر Control Uint است نقش خوندن و نوشتن را مدیریت میکند. Processor با memory در ارتباط است و داده ها را در memory میخواند و مینویسد و از این طریق ما به اطلاعات دسترسی داریم پس این معماری سه مولفه دارد و مولفه ها به این شکل با هم ارتباط دارند که در شکل بالا مشاهده میکنید.

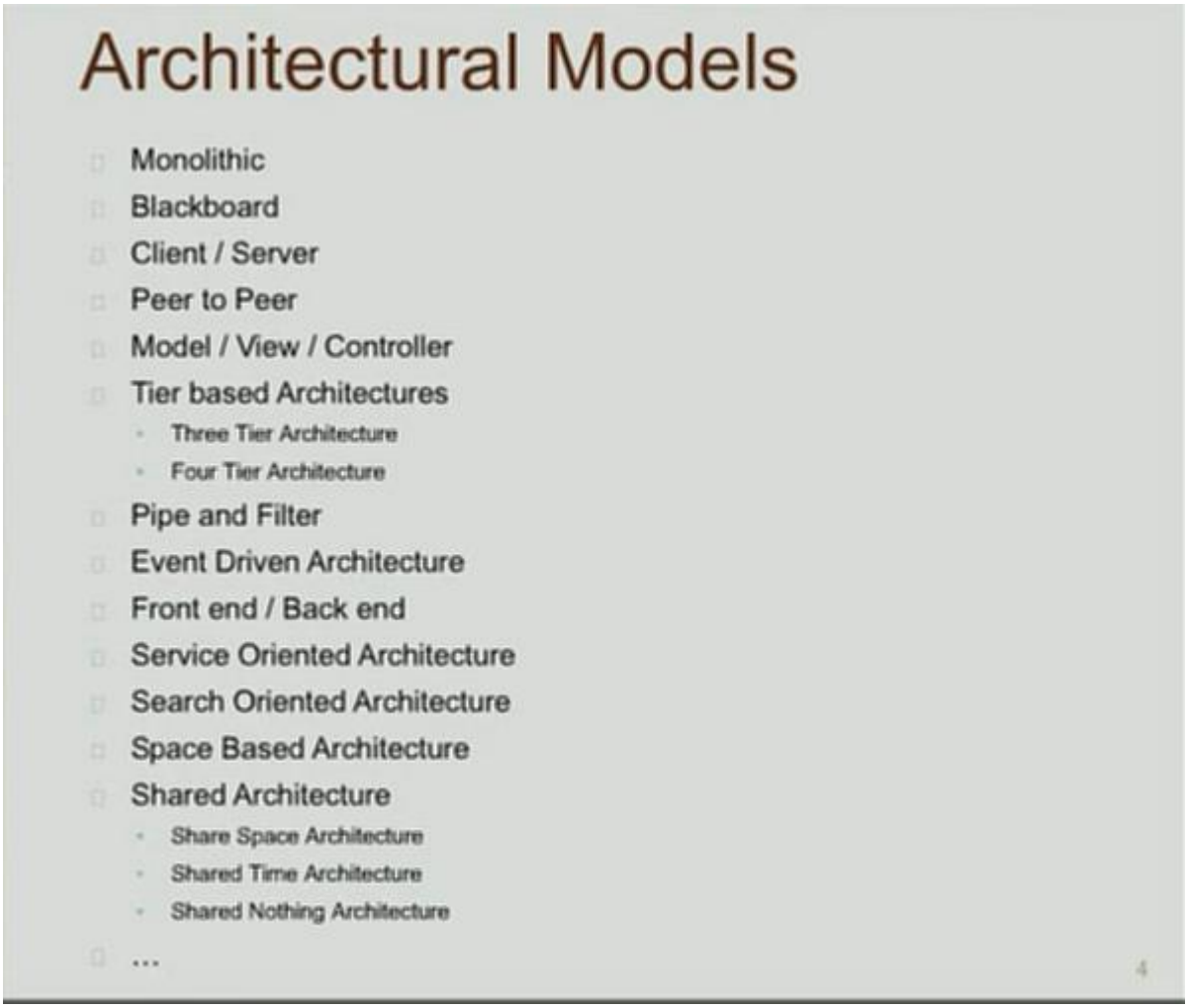
معایب و مزایای این معماری :

- واحد Processor و از واحد mermory جدا است ارتباط زیاد بین Processor و Memory است و دائما با هم حرف میزنن پس Coupling بالایی دارند و Processor روی Memory داده

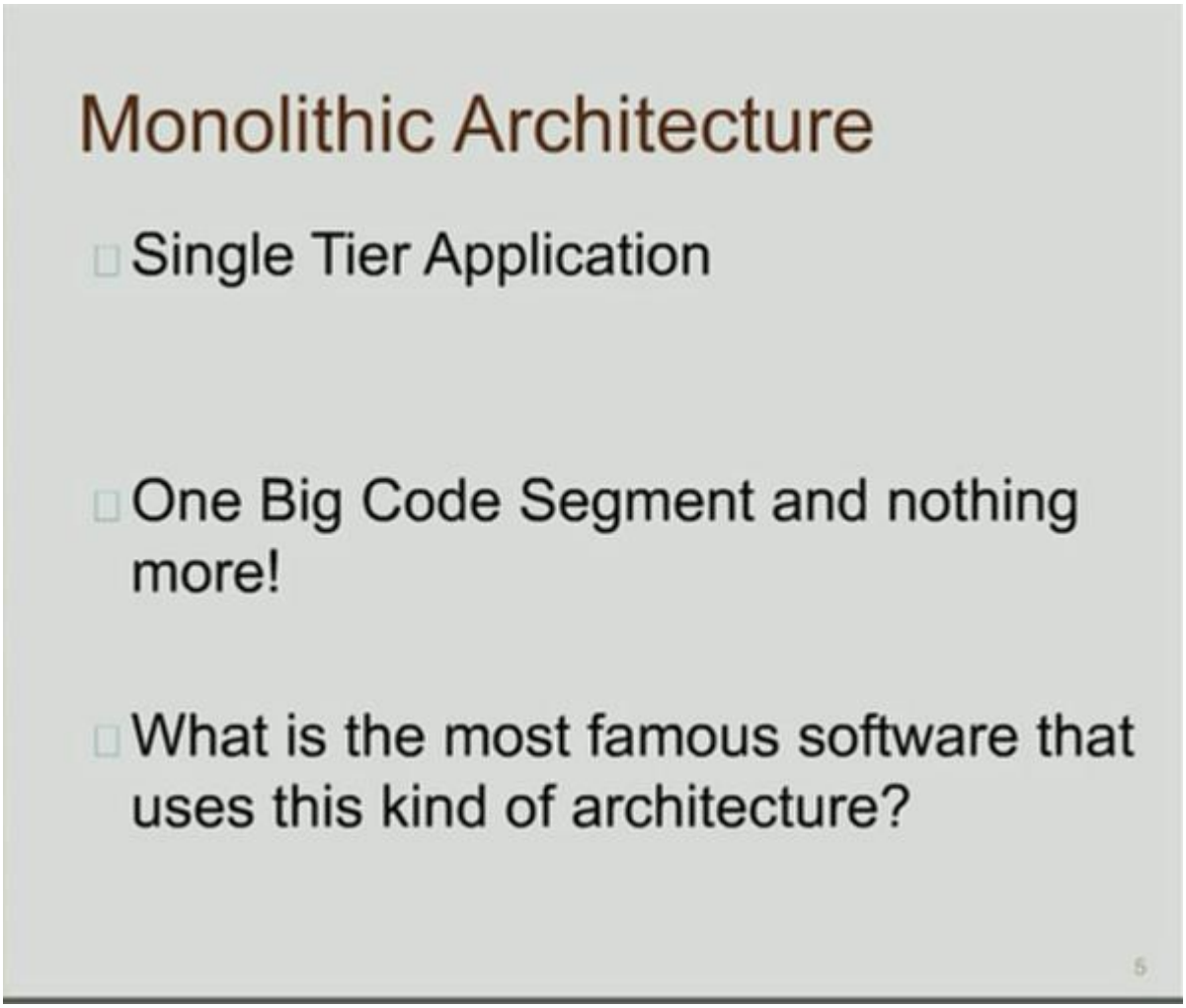
مینویسد و داده میخواند و همچنین سرعت Memory نسبت به Cpu بسیار بسیار پایین تر است با وجود راه حل های cach و لایه اما هنوز مشکل حل نشد و این راه خل ها اساسی نیستند.

پس اگر در شکل نگاه کنید ما سه مولفه داریم که کاملا به هم وابسته هستند (Dependency) دارند اگر بخواهیم درست طراحی کنیم باید هر سه یک جا باشند اما به دلیل محدودیت سخت افزار تصمیم برا آن شده که جدا از هم باشند و از طریق یک Bus انتقال اطلاعات Processor و Memory انجام میشود. و یک محدود کند عمل می کند.

پس به عنوان اولین معماری در سیستم های کامپیوتر مطرح شده است و کوانتوم کامپیوتر ها و DNA ها که اساس فکرشون این بود که این معماری پایه را کنار گذاشته شود و همه در یک جا باشند و در این معماری قرار نیست که از طریق یک Bus با هم صحبت کنند. به این صورت که یک رشته DNA هست که در داخل خودش Processor و Memory دارد و هر دو در یک حا هستند و از Bus در آن خبری نیست و هر دو با هم تو یک جا با هم حرف میزنند و این معماری ها جدید تر هستند.



معماری ها می خواهیم بهش بپردازیم.

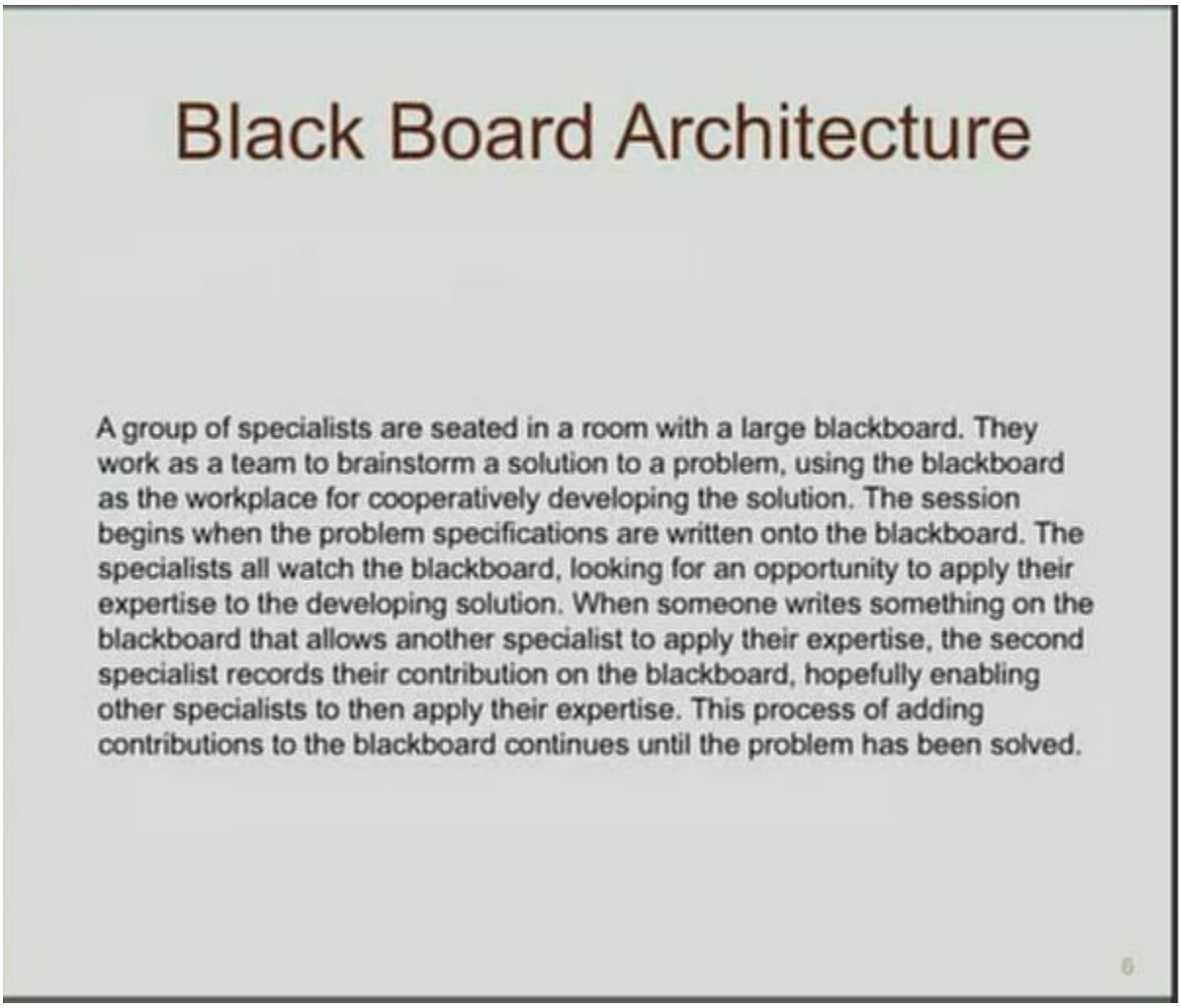


معماری Monolithic :

در این معماری با یک قطعه کد بزرگ سر و کار داریم که تقسیم بندی نشده و انگار هیچ مولفه وجود ندارد و همه چیز در یک لایه است .

از معایب این معماری مدیریت پیچیدگی و نگهداری بسیار سخت است ولی چرا و چه مزیتی دارد که بعضی ها از این معماری استفاده میکنند.مثلا kernel linux از این معماری استفاده کرده است Performance و کارایی است با اضافه شدن لایه روی سرعت تاثیر میگذارد بنابراین در مواقعی که نیاز است در زمانهای کم باید کاری انجام داده شود این معماری کمک میکند. درست است که سرعت را بدست می آوریم ولی یه سری چیزها را از دست می دهیم اگر معماری های Monolithic را نگاه کنیم و کسانی که با معماری Kernel Linux آشنا هستند و کار کرده اند و میدونند که تغییر در کدها خیلی سخت است و با یک تغییر خطایی ایجاد نشود و

معمولا خطایابی و نگهداری کار سختی به حساب میاد و اولین و ساده ترین معماری که به ذهن میرسد همین معماری Monolithic است.کدهای اسمبلی هم از این معماری استفاده کرده است.

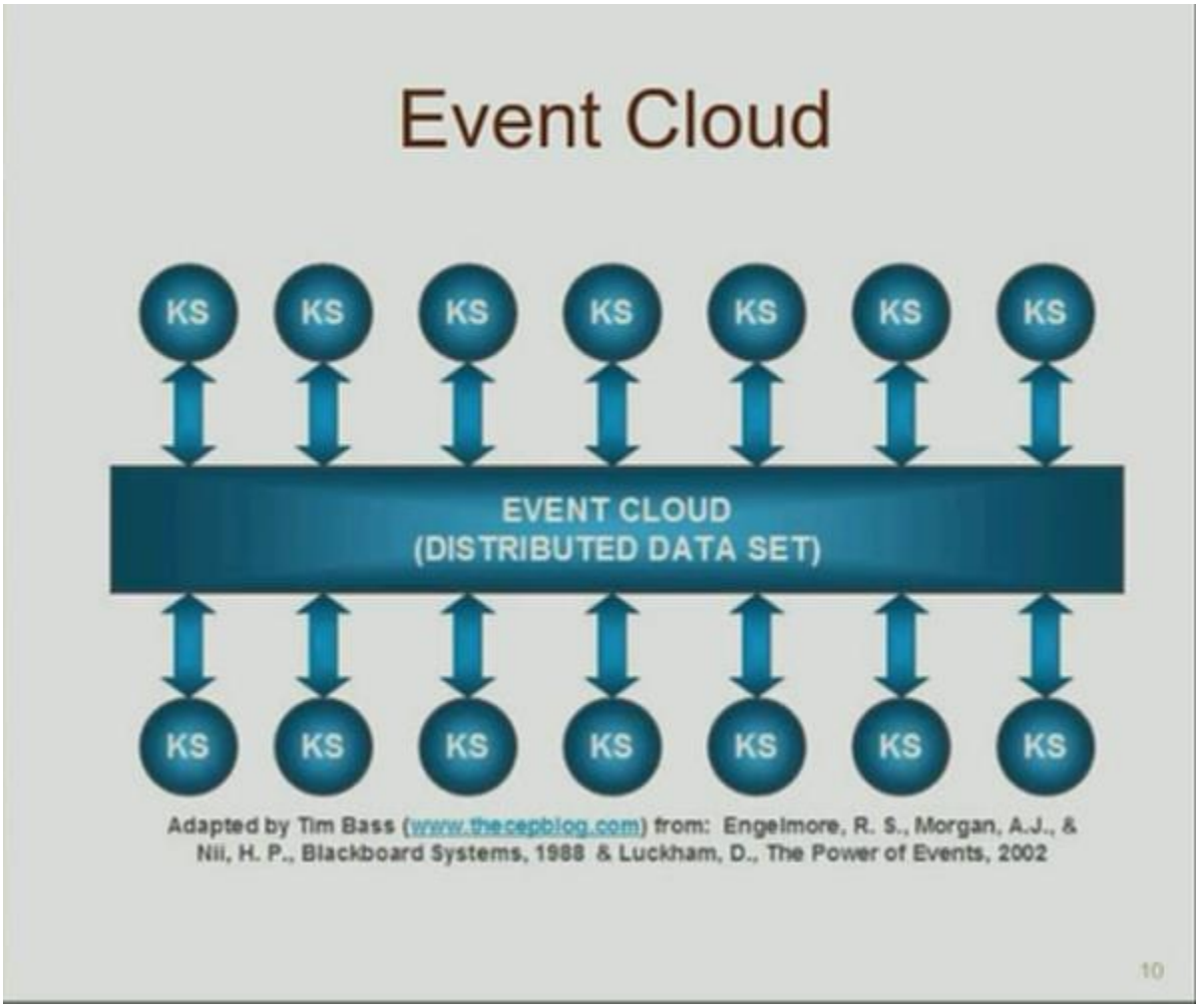


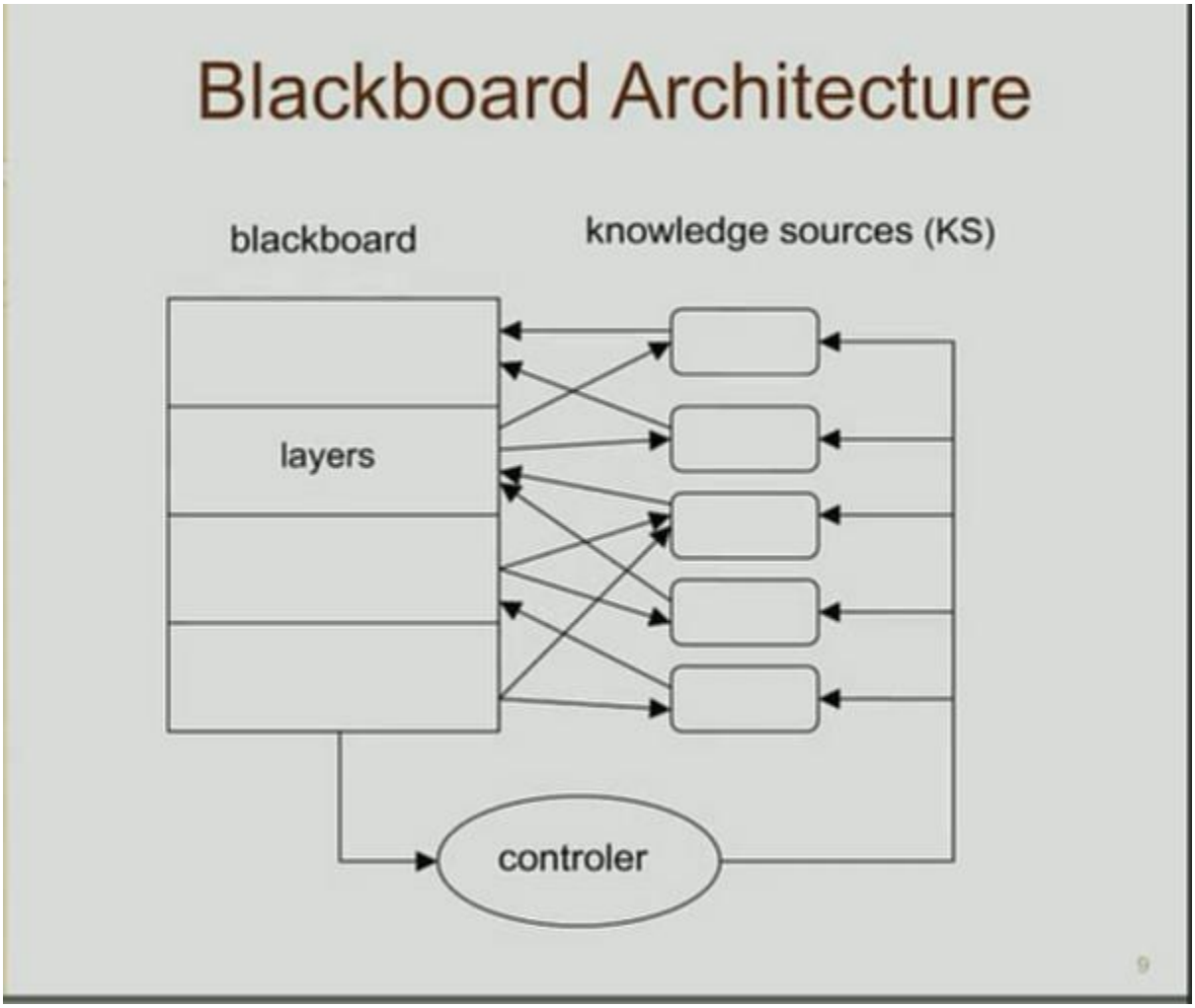
معماری Black Board :

معماری تخت سیاه یا همان Repsitory است ما یک Repository داریم که در آن مینویسیم و دیگران شروع به خواندن و استفاده کردن از آن هستند برای کار کردن با این معماری یک سری Black Board و یک سری Knowlage source داریم که شروع میکنیم روی Black Baord نوشتن و یا از آن خواندن برای اجرای Balck Baordچه مکانیزمی وجود دارد؟ یا به چه طریقی این کار را انجام بدهیم برای پیاده سازی از چه روشی استفاده کنیم ؟

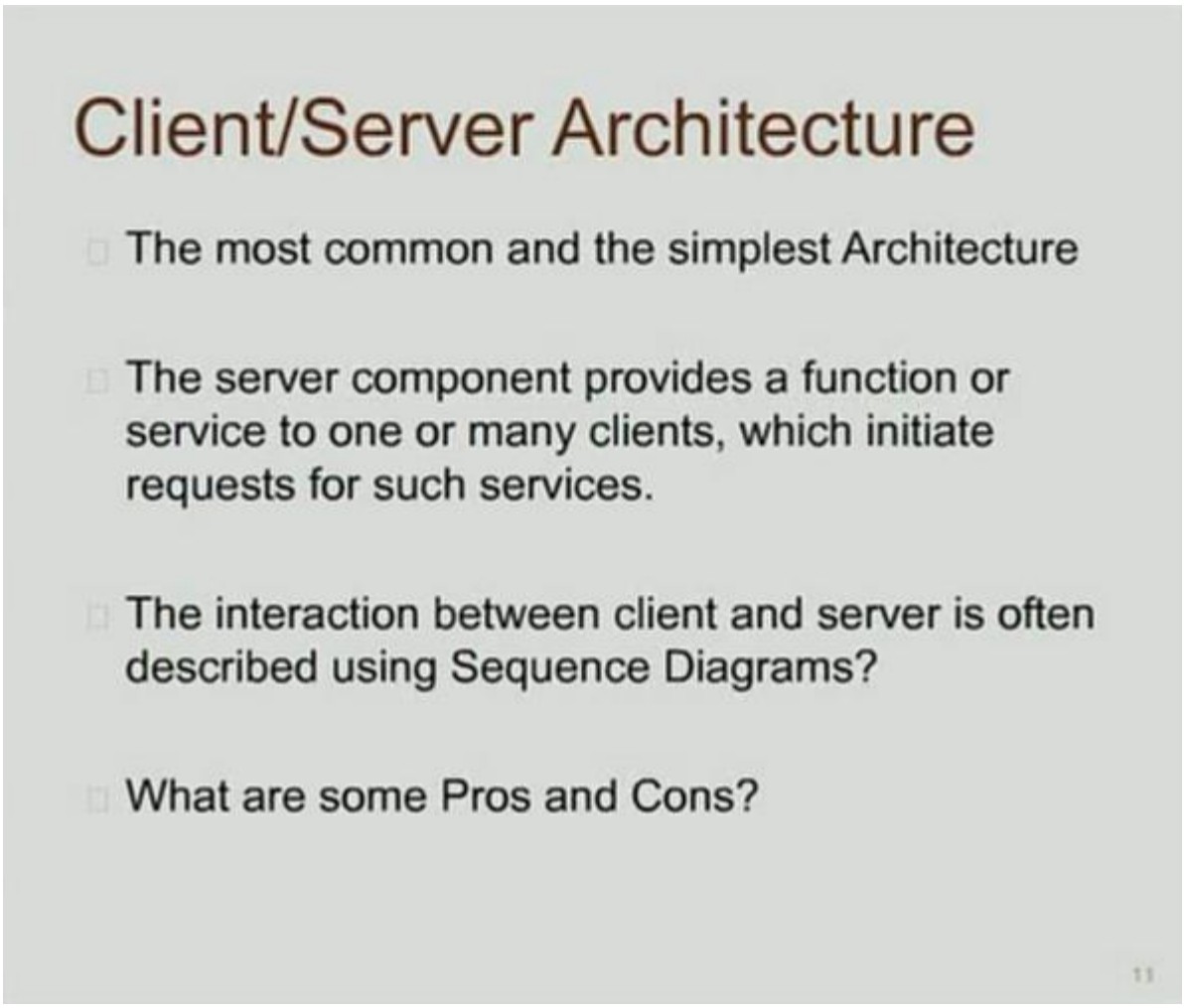
یکی از روش ها برای اطلاع رسانی abserver است استفاده دیگه آن در دیتابیس است که در دیتابیس چیزی بنویسیم و دیگران از دادهای دیتابیس استفاده کنند.استفاده دیگر آن در Assocaite Memory است یعنی این که من یک Memory تعریف میکنم که همه مولفه های من آن را ببینند و بتوانند آدرس دهی کنند و بتونند در آن آدرس چیزی بنویسند و از آن بخوانند یا می توانیم یک فایل داشته باشیم یا یک فایل توزیع شده مثل HFC یا XML که با یک مکانیزمی این فایل را در اختیار همه قرار بدهیم و همه بتوانند از آن بخوانند یا بنویسند یعنی در اختیار همه قرار بدهیم.

پس ما در repository ما تبدیل به یک Assocatie Memory بشود و Source Controle ها نمونه بارزی از این سیستم ها هستند مثال دیگر Event Cloud است که شما می توانید یک عالمه Knowlage Source داشته باشید و همه بتوانند Event هایی روی آن قرار دهند و دیگران را از رخ داد Event با مطلع کنند یا Wikipdia که در داخل آن یک سری دانش نوشته می شود و دیگران از آن استفاده میکنند.

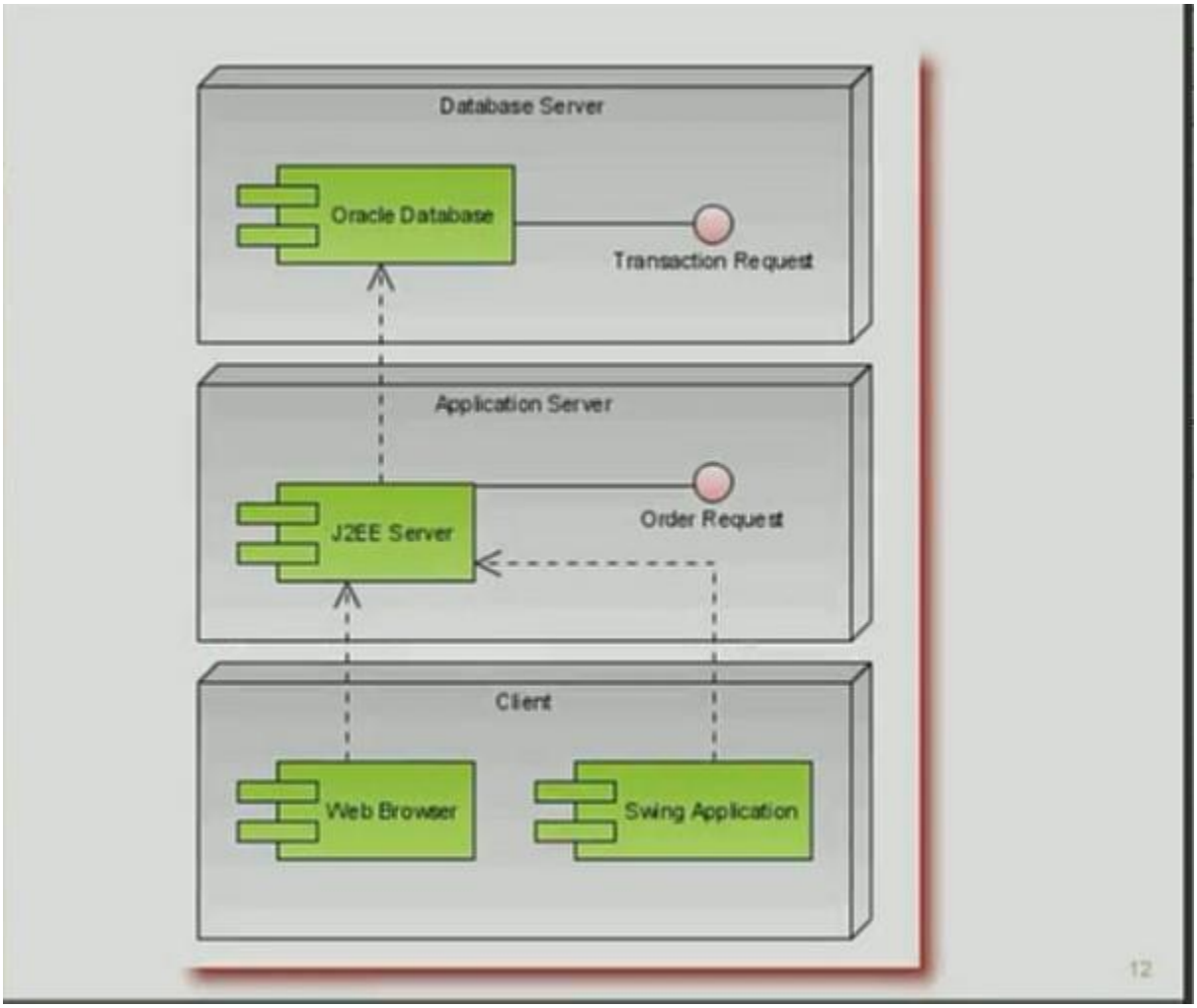




معماری Client-Server :

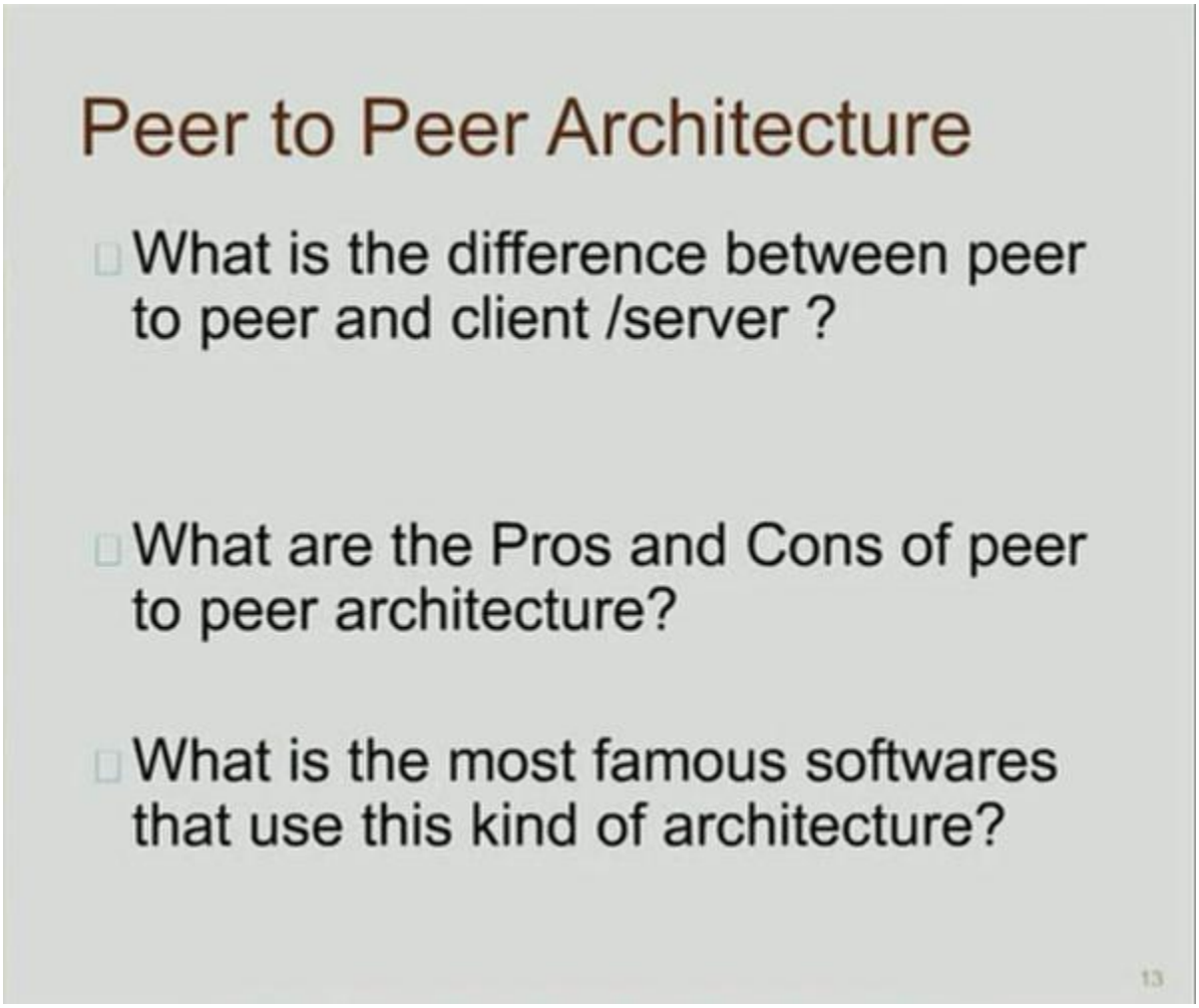


معماری یک معماری کلی است ما می توانیم برای Client یک معماری داشته باشیم برای Server هم همینطور ، معماری ها هر دو طرف میتوانند کاملا متفاوت باشند.

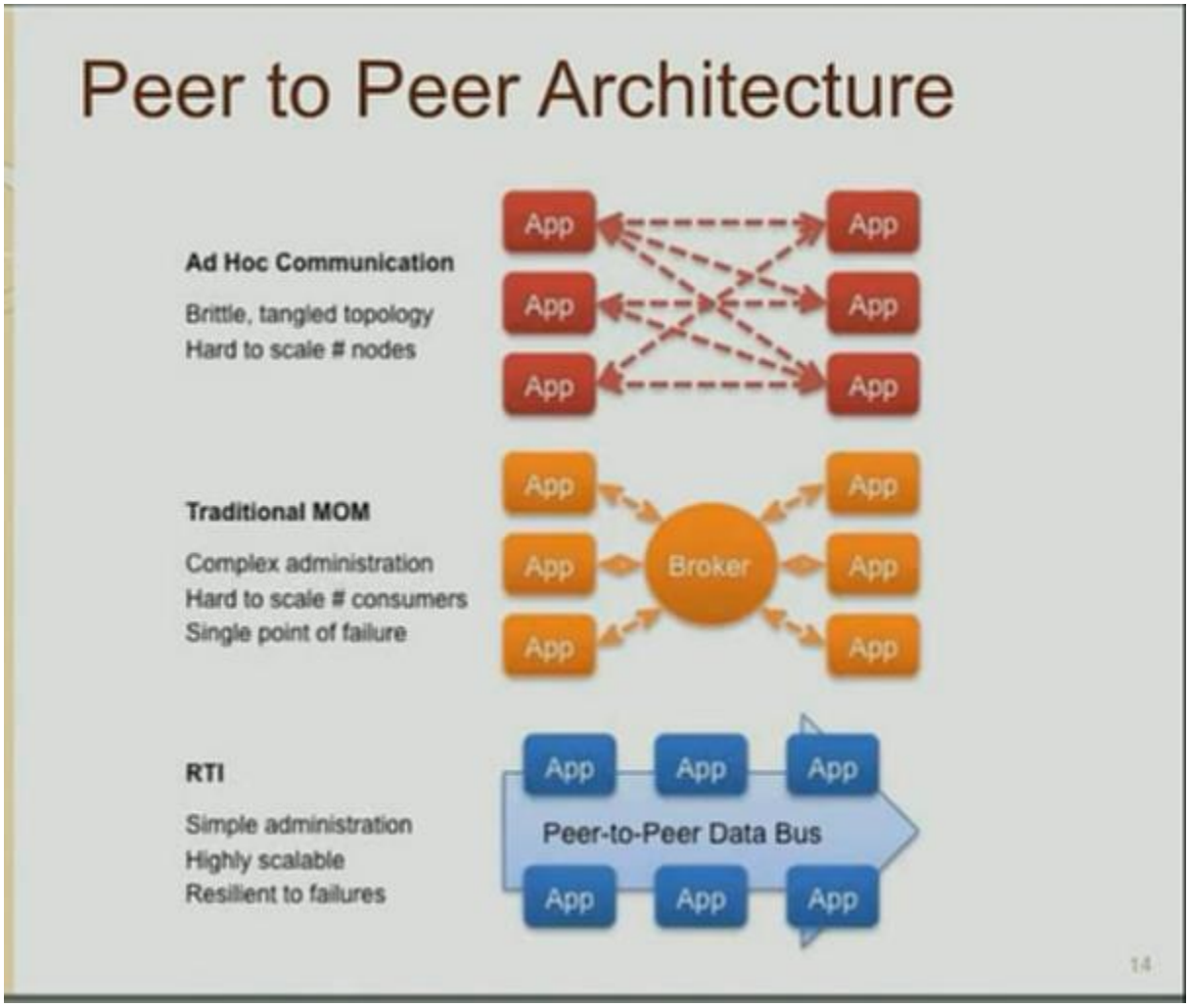


یا معماری Client – Server را در بستری معماری های دیگر ببینیم مثلا یک معماری سه لایه داشته باشیم Client، Applicatio Server وDatabase Server و هر لایه Client لایه بالاتر خودش باشد همان طور که قبلا گفته بودیم ما می توانیم معماری های باز یا بسته داشته باشیم. در معماری لایه باز هر لایه می تواند لایه بعدی خود را Skip کند و به لایه پایین تر خود برود ولی در معماری لایه بسته فقط و فقط به لایه پایین تر از خودش دسترسی و سر کار داشته باشدو به اصطلاح لایه Client لایه پایین تر است.

معماری Peer to Peer :



همانطور که قبلا گفتیم بزرگترین تفاوت Peer to Peer با Client-Server این هست که هر نود علاوه بر اینکه می تواند Client باشد Server هم میتواند باشد.



حالت اول Pure یا محض (شکل اول):

سرور مرکزی وجود ندارد و هر نود فقط و فقط از طریق ارسال پیام بهم و نودهای همسایه با استفاده از پروتکل شایع پراکنی (Gossip) که بتوانند با نودهای دیگر حرف بزنند .

در مورد معایب و مزایا قبلا صحبت کردیم و گفتیم خیلی وقتها ممکن است این نود ها شبکه را Flod بکنند و یا یک نود اساساً وجود نداشته باشد.

حالت دوم (شکل دوم) :

راه حل اینکه به جای Pure Peer to peer داشته باشیم semi - peer to peer داشته باشیم به این صورت که یک Broker یا سرور مرکزی داشته باشیم همه Request ها به سمت این Broker بروند د رواقع آدرس IP آن نود که peer ما دنبال آن است را به آن Peer برگرداند و بعد از این که آدرس Ip برگشت داده شده Application برود سراغ آن Application متناظر خودش و حالا شروع به صحبت کردن بکنند شکل وسطی نمونه از semi –Peer to Peeer است یک سرور میانی یا Broker در وسط قرار دارد و وظیفه آن فقط نودهایی که میان سراغ آن IP متناظر آن را برا گرداند و بعد ش خودش کنار برود تا نودها با همدیگر صحبت بکنند.

مزایا و معایب حالت دوم :

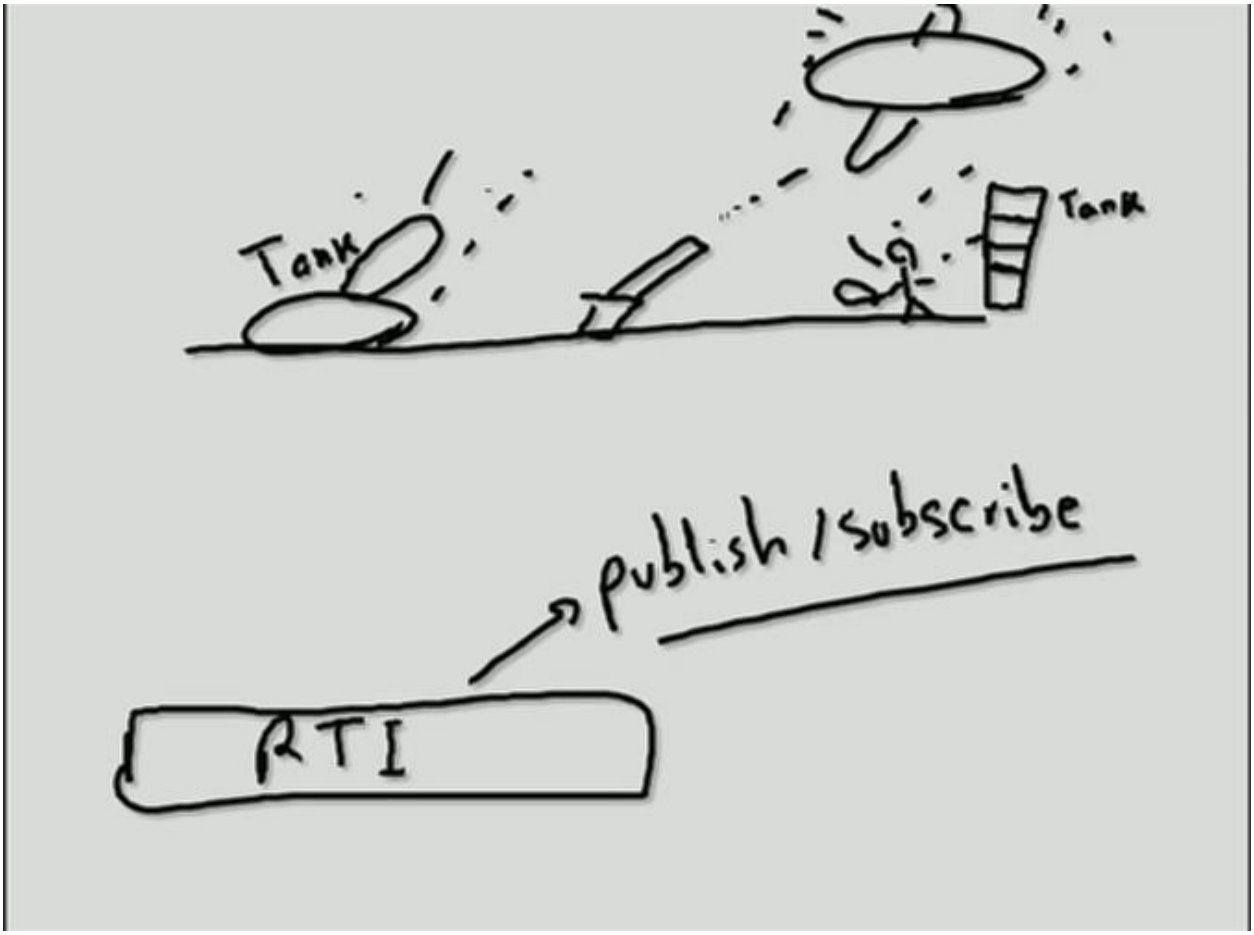
Scale کردن آن به خاطر وجود سرور کار سختی است.

Single Point of Failure به هر حال اگر این Broker به هر دلیلی دچار مشکل شود کل شبکه دچار مشکل میشود.

راه حل سوم (شکل سوم RTI) :

این روز ها این روش خیلی مرسوم است فرض کنید Application ها روی یک Bus قرار دارند یا Association Memory قرار دارند که میتوانند روی Bus بنویسند و دیگران به محض نوشتن از آن مطلع شوند مثلا یک app به یک app دیگر بخواهد داده ایی ارسال کندکافی است که داده ها را روی Bus بنویسیم و به بگوید با app فلان کار دارم و همه app اون پیام رو میخوانند و اعلام میکنند که ما اون app نیستیم و آن app میرود سراغ app برای پاسخ گویی که مخاطب مورد نظر باشد.

به این Approach میگویند run time structure یا RTI این روش هم Administration ساده دارد چون که سرور و نگهداری از سرور را نداریم و هم به شدت Scalable است چون نود مرکزی و سرور در وسط وجود ندارد و در برابر خطا failure مقاوم است اگر یک نود faile شود اتفاق وحشتناکی رخ نمیدهد. پس این راه حل برای سیستم های peer to peer مناسب است یک مثال برای RTI سال های پیش یک تکنولوژی کنار HLA مطرح شد مخفف High Level Architecture این معماری توسط D.D وزارت دفاع آمریکا استانداردسازی شد برای شبیه سازهای نظامی هر کسی که میخواهد شبیه سازی انجام دهد از زوج یا مبنای استاندارد HLA که زیر بنای آن RTI است را توسعه بدهد RTI امکان ایجاد یا اجرای شبیه سازی توزیع شده را میدهند.



برای مثال مثلا می خواهیم شبیه سازی مانورنیروی زمینی را انجام دهیم مثلا یک تانک ، هواپیما ، توپ خانه و هر کدام از این شبیه سازی ها بلاقوه می توانند به دریافت رویداد موجودیت دیگر علاقه مند باشند مثلا تانک هیچ علاقه مند به دریافت event های هواپیما ندارد چون تانک نمی تواند هواپیما را ببیند و به هیچ طریقی به آن شلیک کند ضد هوایی دریافت event های هواپیما علاقه مند است و از طرفی تانک به دریافتEvent های آرپیچی زن علاقه مند است و تمایل دارد یک سری اطلاعات از آن دریافت کند.

پس تو این حالت در هر لحظه یک عالمه Event داریم که به راحترین حالت Event میتواند تعریف تغییر جایگاه و موقعیت باشد پس RTI میگوید من یک مکانیزم Publish/ Subscribe دارم که به صورت توزیع شده است یعنی هر کدام از Entity ها وقتی وارد شبیه ساز میشوند خود را Subscribe میکنند برای دریافت event ها یک سری از Agent خاص مثلا اشتراک یک روزنامه را میگیریم وقتی نسخه ای از روزنامه بیرون

آمد منتشر شد اولین کسی که از این اتفاق مطلع میشود ما هستیم چون ما Subscribe کردیم و خودمان را علاقه مند به دریافت اطلاعات جدید آن روزنامه نشان داده ایم.

پس هر کدام از entity ها وارد شبیه ساز میشوند در واقع وارد یک لیست میشوند که این لیست Agent هایی است که به دریافت رویدادها علاقه مند هستند مثلا A1 به دریافت رویدادهای تانک علاقه مند است و A2 به رویدادهای هواپیما علاقه مند است.

هر کدارم از این ها که Event تولید میکند Event خود را میفرستند به RTI ، RTI نگاه میکند وکه این Event از چه Entity یا Agent تولید شده تا Event را redirect میکند به Entity هایی که در لیست Subscribe ها علاقه مند به دریافت Event هستند مثلا من علاقمند به دریافت Event تانک هستم که یک Event تانک ایجاد شده فرستاده شده به RTI و RTI این Event را میفرستد برای همه نودهایی که تو لیست Subscribe دریافت Event از Agent هستند این request را دریافت کنند و بعد از پردازش پاسخ دهند این شبیه سازی می تواند به صورت توزیع شده اجرا شود.Agent ایی که Event تانک را پردازش میکند میتواند در یک نود کاملا متفاوت باشد و روی یک نود سخت افزاری وجود داشته باشند این هواپیما روی یک نود دیگر باشد به همین صورت نودها با هم انتقال پیغام به همدیگر داشته باشند در واقع کار خودشان را پیش ببرند و کار شبیه سازی شده انجام شود.

ما در حالت سوم (شکل سوم)یک سری peer داریم که از طریق Run Time Structer (RTI) با همدیگر صحبت میکنند هر Peer یک لیست از SubScript ها دارد که علاقمند به دریافت Event یا پیغام هایی از peer هایی استت به محض اینکه یک peer می خواهد با یک peer دیگر صحبت کند این peer می فرستد به RTI و RTI آن را Redirect میکند به Peer که در لیست Subscribe های آن Peer هستند .

سوال این Bus بین Application هایی است چه جوری این مسیر مشترک بین app است؟

یک مولفه همان RTI است RTI به عنوان یک مولفه نرم افزار در همه App ها وجود داشته باشد به اصطلاح Application ها باید RTI-Enabled باشند در واقع RTI باید یک Pattern Observer توزیع شده در این معماری برقرار کند.

تمرین :

برای Snap تمام Feature ها و مولفه های آن را پیاده سازی کنید و یک معماری پیشنهاد بدهید.