Network Security

- شبکه های کامپیوتری با این ذهنیت که مورد حمله قرار بگیرن، ساخته نشدن و طراحی های این شبکه ها برای مقابله با اعمال خرابکارانه :/ نبوده، این باعث شده در شبکه های کامپیوتری نسبت به کسایی که این اعمال رو انجام میدن عقب باشیم و به اصطلاح یه حالت catch up (تعقیب و گریز) نسبت به این افراد داریم. همیشه یه سری حملاتی انجام میشه و بعد یه سری تغییراتی در شبکه ایجاد می کنیم تا اون حملات دفع بشه.
- در مورد امنیت شبکه های کامپیوتری ، باید بدونیم که افراد خرابکار به چه نحو به شبکه های کامپیوتری حمله می کنن، ما چطور می تونیم در مقابل این حملات دفاع کنیم ، و اینکه چطور پروتکل ها مون رو با ملاحظات امنیتی طراحی کنیم.
- Packet sniffing : ممکنه وقتی داریم یه بسته ای رو به یه مقصدی می فرستیم ، وسط کار یه شخص ثالثی بسته ی ما رو دریافت کنه و به اطلاعاتش دسترسی پیدا کنه .

این روش در broadcast media ها مثل Shared Ethernetو شبکه هایwireless خیلی راحت تره به خاطر اینکه بسته به تمام نود ها در مسیر فرستاده میشه. ما با تنظیماتی که روی کارت شبکه انجام میدیم به کارت شبکه بگیم که علاوه بر بسته هایی که آدرسشون آدرس

خودمون هست ، سایر بسته هایی هم که دریافت می کنه رو هم به لایه های بالای پروتکل انتقال بده و ما هم بتونیم در اون لایه ها داخل بسته ها رو بررسی کنیم.

اون mode ای که کارت های شبکه معمولا کانفیگ میشن که packet sniffing رو انجام بدن ، بهش promiscuous گفته میشه و البته نرم افزار wireshark هم میتونه این کار رو انجام بده.

- IP Spoofing (جعل آی پی) : وقتی شخص ثالثی میاد آی پی یوزر معتبری رو به عنوان source IP خودش در داخل بسته قرار بده و برای یه سروری ارسالش کنه ، اگر مثلا مکانیزم احراز اصالت در اون سرور مبتنی بر IP Address ای باشه که بسته توسط اون host ارسال شده، می تونه این مکانیزم احراز اصالت خدشه دار بشه.
- Denial of Service(DoS) : در این حمله یه attacker میاد منابعی که در اختیار یه سرور هست مثل پهنای باند اون لینکی که به سرور متصله، منابع سخت افزاری مثل حافظه ای که در اختیار اون سرور هست و ... رو به ترافیک های جعلی اختصاص بده و به این ترتیب برای کاربرای legitimate (کاربرای اصلی) منابعی باقی نمونه که سرور بخواد به اون ها اختصاص بده.

برای اینکه یه حمله ی DoS اتفاق بیفته : ۱- target (هدف) توسط شخصی حمله کننده انتخاب می شه ، ۲- شخص حمله کننده به یه سری host ها و end system ها نفوذ می کنه و یه سری بد افزار در

داخل اون ها قرار می ده ، و به این ترتیب اون host ها در اختیار اون شخص قرار می گیرن . بنابراین این شخص می تونه یه سری بسته ها و ترافیک ها رو برای hostها و end system ها ارسال کنه. به مجموعه ای از دستگاه ها که یه attacker به اون ها حمله کرده و در اختیارشون گرفته ، botnet می گن و به هرکدوم یه bot (رباتی که تحت کنترل یه هکر قرار گرفته) گفته میشه.

۳-در آخر هم اجرای حمله هست که شبکه ی bot ما ، میاد بسته هایی رو برای تارگت ارسال می کنه . بنابراین منابع سخت افزاری یا پهنای باند لینکش صرف ترافیکی میشه که توسط botnet براش فرستاده شده و اون کاربر های واقعی(legitimate) ای هم که باقی موندن دیگه نمی تونن از اون سرور ، سرویس بگیرن.

Chapter 2 : Application Layer

- برای شبکه های کامپیوتری ، تک تک لایه های protocol stackرو با رویکرد از بالا به پایین بررسی می کنیم .(Top down approach)
- اولین لایه ، application layer هست و میشه گفت مهم ترین لایه هم هست. ما شبکه های کامپیوتری رو می سازیم که بتونیم یه سری اپلیکیشن هایی داشته باشیم که توسط اون شبکه ها بتونن اجرا بشن.

- برنامه های کاربردی ای که در اینترنت هستن طیف وسیعی دارن. مثل social networking ، web ، text messaging ، email، streaming stored video ، multi-user network games (YouTube, Hulu, Netflix) ، P2P file sharing ، voice over IP(skype , ...) و real-time video conferencing که سرویس صوت یا ویدیو کنفرانس رو می تونیم از طریق اینترنت با استفاده از این سرویس ها داشته باشیم ، Internet search ، remote login یا دور کاری و .....
- اهداف ما برای بررسی این لایه اینه که ۱- با مفاهیم اولیه و اصولی لایه ی اپلیکیشن آشنا بشیم ۲- با کاربرد لایه ی transport هم آشنا باشیم ۳- با معماری های client-server و peer-to-peer آشنا می شیم ۴- با یه سری از پروتکل های مهم لایه ی اپلیکیشن مثل HTTP، DNS ، و video streaming system آشنا میشیم. ۵- با نوشتن برنامه های شبکه که بهشون network applications یا socket programming هم گفته میشه ، آشنا میشیم.

• Network Applications

- برنامه های تحت شبکه برنامه هایی هستن که توسط دو یا چند end system ران می شن و توسط شبکه با هم ارتباط دارن و پیام رد و بدل می کنن.مثلا توی وب اپلیکیشن ها ، یه نرم افزاری روی سرور در

حال اجرا هست و سمت کلاینت هم یه browser داریم که در واقع نرم افزار سمت کلاینته و این دوتا نرم افزار از طریق شبکه با هم ارتباط برقرار می کنن تا ما وب اپلیکیشن رو داشته باشیم.

- نکته ی مهم در مورد برنامه نویسی تحت شبکه اینه که اون توسعه دهنده، لازم نیست که برنامه ای بنویسه که روی device های داخل شبکه(network core devices) مثل روتر ها اجرا بشه و تغییری در بستر شبکه صورت نمی گیره. برنامه های کاربردی فقط روی end system ها اجرا میشن . این قابلیت خوبیه از این جهت که وقتی داریم برنامه ای می نویسیم نیازی نیست که کل شبکه تحت تاثیر قرار بگیره و نیازی نیست که مدت زمانی زیادی رو صرف کنیم تا به یه اپلیکیشن جدیدی از طریق اینترنت برسیم، و این کار راحت با نوشتن برنامه روی end system ها محقق میشه .این قضیه باعث توسعه ی سریع و سهولت برنامه نویسی تحت شبکه میشه .
- برای اینکه بتونیم برنامه ی تحت شبکه بنویسیم ، باید با یه سری مفاهیم آشنا باشیم :

۱- باید انتخاب کنیم از چه معماری ای می خوایم استفاده کنیم (از معماری client/server یا p2p)

۲-باید تصمیم بگیریم از چه سرویس لایه ی transport می خوایم استفاده کنیم.

۳- باید داخل برنامه مون ، از Socket API استفاده کنیم و متناسب با اون سرویس لایه ی transport تنظیم و مقدار دهی کنیم.
۴- با استفاده از Socket API شروع به برنامه نویسی کنیم :)

- معماری client-server :

- در این معماری ، یه سری end system های قدرت مندی به اسم server داریم که قسمتی از برنامه توی این سرور ها اجرا میشه . این سرور ها :
  1 - همیشه روشن هستن. (always-on)
  2 - آدرس IP شون همیشه یکسانه و بقیه ای آدرس رو میدونن و می تونن به این سرور ها متصل بشن .(Permanent IP address)
  3 - معمولا یکی نیستن ، وقتی تعداد مشتری ها زیاد بشه ، تعداد زیادی سرور فیزیکی هم در دیتا سنتر ها قرار میدن که مجموعه ی این ها ، یک سرور مجازی قدرتمند می سازه.
- یه قسمت دیگه از برنامه در داخل end system هایی به اسم client اجرا میشه که از لحاظ توان پردازشی میتونن به مراتب ضعیف تر از سرور ها باشن .
  1 - این کلاینت ها با سرور ها ارتباط برقرار می کنن.

2 - ارتباط کلاینت ها با سرور می تونه قطع و وصل بشه و این طور نیست که همیشه به اینترنت وصل باشن.

3 - آدرس IP شون داینامیکه و هر بار که به اینترنت وصل میشن آدرس IP شون ممکنه متفاوت باشه .

4 - این کلاینت ها ارتباط مستقیمی بین خودشون وجود نداره و اگر بخوان مرتبط بشن از طریق سرور این کارو انجام میدن.

- پروتکل HTTP توی وب، پروتکل IMAP توی ایمیل و پروتکل FTP توی file transfer ،نمونه هایی ازین معماری هستن.

- معماری peer-to-peer

- توی این معماری (توی ورژن pure)

1 - سروری نداریم که همیشه روشن باشه .

2 - End system ها به طور مستقیم با هم در ارتباط هستن.

3 - چون همه ی device هم هم رده هستن و هیچ سلسله مراتبی وجود نداره ، به هر کدوم از این device ها و end system ها می گن peer. به هم دیگه سرویس میدن و از هم سرویس می گیرن و این باعث میشه شبکه هایی که این نوع معماری رو دارن ، self scale باشن ، یعنی با افزایش تعداد کاربران مشکلی پیش نیاد و همچنان اون سرویس ارائه بشه.

یه مثال ، p2p file sharing یا bit torrent هست. در اپلیکیشن file sharing یه فایلی هست که بعضیا میخوان اونو دانلود کنن، یه روش اینه که با استفاده از bit torrent اون فایل رو دریافت کنیم. توی bit torrent ، هر کدوم از نود هایی(peer ها) که شروع می کنن به شبکه وصل بشن ، در ابتدای کار هیچ قسمتی از فایل رو خودشون ندارن و میتونن از همسایه هاشون تقاضا کنن هر کدوم قسمت هایی از فایل رو براشون بفرستن. به تدریج در اون نود ، قسمت هایی از فایل نهایی ذخیره شده . اگه یکی از peer های دیگه ای که در شبکه هست از این نود درخواست کنه که قسمتی از فایل رو براش بفرسته ، طبق پروتکل برنامه ی bit torrent میتونه این کارو انجام بده. به طور کلی اینجوری نیست که هرکدوم از peer ها فقط مصرف کننده یا فقط دریافت کننده ی سرویس باشن و میتونن هردو کار رو انجام بدن. این نقش توأمان سرویس دهندگی و سرویس گیرندگی باعث میشه که این معماری self scale باشه ، برخلاف معماری client-server که اگر تعداد کلاینت هامون زیاد میشد باید سرمایه گذاری زیادی می کردیم و تعداد سرور هامون رو زیاد می کردیم تا بتونیم بهشون پاسخ بدیم.

4 - البته چالش هایی هم در این معماری داریم : peer ها همیشه وصل نیستن و IP Address فیکسی هم ندارن ، از لحاظ امنیتی هم

اطمینان کردن به peer ها نسبت به اطمینان کردن به سرور ها سخت تر هست و چالش های خودش رو داره.
همچنین ما باید به نحوی پروتکل هامون رو طراحی بکنیم که peer ها تمایل به سرویس دادن هم داشته باشن .یعنی مجبور باشن در کنار سرویسی که از بقیه می گیرن ، سرویس هم بدن. (باید incentive بشن.)

- Process Communicating

- اپلیکیشن ها در قالب برنامه هایی هستن که وقتی در end system ها به اجرا در میان ،بهشون Process گفته میشه . بنابراین هدف غایی یه شبکه ی کامپیوتری این هست که process هایی که در داخل end system های مختلف هستن بتونن با همدیگه ارتباط داشته باشن.
- بنابراین طبق این تعریف ، در یه host یا end system ، میتونه تعداد زیادی process وجود داشته باشه که یا این process ها توی یک end system ، با همدیگه از طریق روش inter-process communication در ارتباط هستن(که سیستم عامل ها مکانیزمی دارن که میتونن ارتباط بین این process هارو برقرار کنن) ، یاااا اینکه process هایی که از end system های مختلف هستن میخوان با همدیگه ارتباط برقرار کنن . اینجاست که پروسس ها در

داخل end system های مختلف از طریق شبکه های کامپیوتری با هم پیام هایی رد و بدل می کنن.

- توی ادبیات مربوط به برنامه نویسی شبکه ، در سطح دیگه ای هم (در سطح process ها) اصطلاح client وserver رو استفاده می کنیم.
موقعی که داریم یه برنامه ای تحت شبکه می نویسیم به اون بخشی از برنامه که در سمت کلاینت اجرا میشه می گن client و به اون بخشی از برنامه که سمت سرور اجرا میشه می گن server .
در واقع به process ای میگن client که شروع کننده ی یه ارتباط هست و به process ای میگن server که منتظر تقاضای ایجاد ارتباطه و وقتی دریافتش می کنه ، قبول می کنه و ارتباط شکل می گیره.
- از اون جایی که توی p2p ، peer ها هم سرویس دهنده و هم سرویس گیرنده هستن ، برنامه ها هم server هستن هم client .یعنی در نقش server به همسایه هاش که به اون peer متصل هستن سرویس میده و در نقش client از همسایه هاش تقاضای سرویس می کنه.

- transport-layer service

- پروتکل ها در لایه ی اپلیکیشن به دو دسته تقسیم میشن ، ۱- open source مثل HTTP و SMTP ۲- خصوصی (proprietary) مثل skype و zoom .
- پروتکل های لایه ی اپلیکیشن مبتنی بر اینکه قراره چه سرویسی رو ساپورت کنن ، یه سری نیازمندی هایی از شبکه دارن.

1 - ممکنه در اون اپلیکیشن ، نیاز به سرویس data integrity (صحت داده) از لایه ی پایینی که همون لایه ی transport هست داشته باشیم. یعنی داده هایی که لایه ی اپلیکیشن به لایه ی transport میده ،انتظار داره این داده ها بدون خطا ، بدون عوض شدن ، بدون گم شدن ، و کلا بدون اینکه چیزی صحت داده ها رو مخدوش کنه ، داده ها توسط لایه ی transport ، به پروتکل های لایه ی اپلیکیشن گیرنده منتقل بشن. این سرویس هم تحت عنوان data integrity و هم reliable data transfer شناخته میشه که بیشتر از عنوان دوم استفاده می کنیم تحت عنوان اینکه میخوایم connection مون reliable باشه.

مثلا در file transfer یا توی اپلیکیشن وب خیلی به این سرویس حساس هستن . اما یه سری از اپلیکیشن ها هم حساس نیستن، مثل اپلیکیشن صوت ، که اگه یه سری از داده هایی که می فرستیم خراب بشن یا گم بشن ، اینجوری نیست که ارتباط ما رو خیلی تحت تاثیر

قرار بدن.(اگه خرابی ها از یه سطحی بیشتر بشه ، توی کیفیت سرویس خودش رو نشون میده اما تا یه حدی قابل تحمله.)

2 - یه معیار و نیازمندی دیگه ، timing هست. مثلا یه سری برنامه ها مثل اپلیکیشن های real-time (مثل internet telephony یا interactive game)به تاخیر حساس هستن و اگه تاخیر از یه حدی بیشتر بشه ، توی استفاده ی کاربر از اون سرویس تاثیر میذاره.

3 - یه نیازمندی دیگه ، throughput هست . اپلیکیشن ها در قبال این نیازمندی دو دسته هستن: ۱- بعضی اپلیکیشن ها مثل multimedia ها که یه حداقل بیت ریتی نیاز دارن که بتونن سرویس شون رو به صورت مناسب ارائه کنن. مثلا توی سرویس voip(voice over IP) اگه کمتر از یه بیت ریتی بهش داده بشه ، عملکردش مختل میشه.

۲-اما یه سری از برنامه ها هستن مثلا elastic apps که در قبال throughput حساسیت دسته ی قبل رو ندارن. مثلا توی اپلیکیشن file transfer گاهی وقتا ممکنه به صورت مقطعی throughput نزدیک صفر هم بشه ، اما همچنان پروتکل هایی که فایل ها رو ارسال و دریافت می کنن منتظر می مونن که throughput شبکه افزایش پیدا کنه و دریافت و ارسال رو ادامه بدن.

4 - یه سرویس دیگه security هست که لایه ی transport در مقابل امنیت داده هایی که ما ارسال می کنیم به ما تضمین بده. مثلا داده های ما در مسیر مقصد توسط شخص ثالثی دریافت نشن و اون شخص نتونه از اطلاعات ما استفاده کنه. در واقع لایه ی transport باید اطلاعات ما رو رمز نگاری کنه که کسی دیگه ای متوجه محتویات بسته های ما نشه.(encryption)

همچنین کسی نیاد قسمتی از اطلاعات مارو دستکاری کنه و به نام ما به مقصد برسونه که به این سرویس هم data integrity نام داره (با data integrity قبلی فرق داره) . بحث امضاهای دیجیتال هم مربوط به همین قضیه هست.

- در اسلاید ۲ آورده شده که چه اپلیکیشن هایی چه نیازمندی هایی و با چه حساسیت هایی وجود دارن و در مقابل این معیار ها چه رفتاری از خودشون نشون میدن.
- اینکه اپلیکیشن ها چه نیازمندی هایی دارن یه طرف قضیه است ، اینکه با توجه به ساختار شبکه ، لایه ی transport میتونه چه سرویس هایی در اختیار پروتکل های لایه ی اپلیکیشن بذاره ، یه بحث دیگه ست .
- دو نوع سرویس لایه transport می تونه در اختیار لایه ی اپلیکیشن بذاره :۱-سرویس های پروتکل TCP ۲-سرویس های پروتکل UDP

- TCP services :

۱-توی سرویس هایی که این پروتکل به لایه ی اپلیکیشن میده، این ضمانتو می کنه که داده هایی که لایه اپلیکیشن ارسال می کنه به طور reliable در اختیار process لایه ی اپلیکیشن گیرنده قرار می گیره، بدون هیچ خطایی.(reliable transport)

۲-flow control : یعنی فرستنده ی پیام و گیرنده ی پیام اگر از لحاظ میزان بافرینگ همخوانی نداشته باشن(یعنی مثلا فرستنده با سرعت زیادی داده ارسال کنه ولی گیرنده محدودیت بافر داشته باشه) اینطور نشه که در اثر محدودیت بافر گیرنده و سرعت زیاد ارسال فرستنده، بافر گیرنده overflow کنه و یه سری بسته ها گم بشن. بلکه TCP تضمین می کنه که سرعت فرستنده به صورت adaptive بر اساس حجم بافر گیرنده که برای دریافت بسته هایی ارسالی تعبیه شده ، منطبق شده.

۳-congestion control (کنترل ازدحام) : ربطی به یه ارتباط خاص نداره و ربط به این داره که یه کل یه شبکه چجوری کار کنه. TCP ها یه مکانیزمی دارن که وقتی می بینن تاخیر شبکه از یه حدی بیشتر شده ، میان به صورت داوطلبانه rate ارسال بسته رو کم می کنن تا اوضاع شبکه به حالت مطلوب برگرده . اگر هم اوضاع شبکه خوب بود ، rate ارسال رو زیاد می کنن.

۴-connection-oriented (اتصال گرا) : ما احتیاج داریم برای استفاده از TCP ،قبل از اینکه بسته ای رو ارسال کنیم ، یه سری سیگنال بین TCP گیرنده و فرستنده رد و بدل کنیم ، یه سری پارامتر ها ست بشه ، یه سری توافقات صورت بگیره و یه resource هایی از سیستم عامل بگیرن ، تا بعد بتونن این انتقال داده رو به صورت reliable انجام بدن. به این کار میگن handshaking .

برای مثال وقتی می خوایم با TCP یه سرور ارتباط برقرار کنیم ، هم از سیستم عامل خودمون باید یه تایمر و یه بافر در اختیار بگیریم و هم سرور باید از سیستم عامل خودش، یه resource ای تحت عنوان تایمر و یه resource ای تحت عنوان بافر بگیره تا بعد بتونه اون مکانیزم مربوط به ارسال بسته و منتظر شدن برای اینکه ack بسته بیاد رو فراهم کنه. مثلا اگه بسته ای طبق اون زمانی که تایمر نشون میده، زمان اومدن ack اش بیش از حد طول کشید ، فرستنده بنا رو بر این میذاره که بسته گم شده و باید اونو re-transmit کنه . یا مثلا اگه بسته ها ترتیبشون به هم ریخت، یه بافری در گیرنده وجود داشته باشه که بعد از اینکه همه ی بسته ها ارسال شدن، توی این بافر اون ها رو دوباره مرتب کنه و به پروتکل های لایه ی اپلیکیشن بده.

- البته TCP همه ی سرویس ها رو نمیده! مثل timing ، minimum throughput guarantee ، security .

- UDP services :

- خیلی از کارایی که TCP انجام می داد UDP انجام نمیده . مثل reliable data transfer ، flow control ، congestion control . پس UDP به چه درد می خوره؟!
- یکی از کارکرد های لایه ی transport تحویل بسته به process ها هست که بهش می گفتیم multiplexing و de-multiplexing و توسط port number هم مشخص می کنن که هر بسته دقیقا باید به کدوم process تحویل داده بشه.
- پس چرا بعضی از اپلیکیشن ها ترجیح میدن که به جایTCP از UDP استفاده کنن؟ به خاطر اینکه سرویس هایی که TCP ارائه میده هزینه هایی داره و این هزینه ها دوتا چیز هستن :۱- تاخیر ۲- سربار زیاد TCP برای اینکه این سرویس ها رو فراهم کنه ، نسبت به UDP هدر بزرگتری داره . بنابراین از لحاظ صرفه جویی در پهنای باند UDP بهتره چون هدر کوچکتری داره.

TCP هم به خاطر سیگنالینگ اولیه ، هم به خاطر اینکه مجبوره یه سری از بسته ها رو دوباره بفرسته ، هم به خاطر congestion control ، و .... تاخیرش رو نسبت به UDP خیلی بیشتر می کنه و همین باعث میشه اپلیکیشن هایی که تاخیر براشون مهمه(مثل real-time ها) UDP رو ترجیح بدن.

- البته اپلیکیشن های real time هم هر چند به صورت اولیه دوست دارن UDP استفاده کنن، ولی بعضی از سازمان ها ، اون فایروالی که استفاده می کنن، اجازه ترافیک UDP رو نمیدن ، چون هیچ گونه ملاحظه ای نداره و وضعیت شبکه رو نمی بینه ، و اگه یه فایروالی ببینه که اپلیکیشنی داره از پروتکل UDP استفاده می کنه ، اونو drop می کنه و کانکشن UDP شکل نمی گیره.
برای همین ، از TCP هم ساپورت می کنن که اگه جایی UDP فیلتر شد، بتونن از طریق TCP سرویسشون رو ارائه بدن.
- ولی مشکلات دیگه مثل throughput و ... رو به شکل دیگه ای حل می کنن.معمولا در حال حاضر در اپلیکیشن هایی که ما داریم ، از تکنولوژی هایی در رابطه با adaptive coding استفاده میشه که ما rate ارسال بسته مون رو منطبق با شرایطی که شبکه داره و throughput ای که در اختیارمون میذاره ، تنظیم کنیم.(البته اگه throughput و تاخیر از یه حدی بیشتر بشن ، اون سرویس ها عملکرد خوبی نخواهند داشت)
- TCP و UDP سرویس امنیت رو ارائه نمی کنن. البته در نسخه های بهبود یافته ی TCP ، یه پروتکلی به نام Transport Layer Security (TLS) اضافه شده . این پروتکل توی لایه اپلیکیشن پیاده سازی میشه و توسعه دهنده باید لایبرری های مربوط به این پروتکل رو سمت سرور و کلاینت استفاده کنه و بعد وقتی میاد پیامی رو ارسال می

کنه ، اول توسط توابع مربوط به رمزنگاری ، عملیات ارتباط امن انجام می گیره و پیام در اختیار یه سوکت TCP قرار می گیره که اونو ارسال کنه.

- یه توسعه دهنده ی برنامه های تحت شبکه ، باید API هایی که توسط لایه ی transport در اختیارش قرار می گیره تو برنامه ی خودش استفاده کنه ، و توسط اون API ها سرویسی رو که میخواد از لایه ی transport بگیره. به این API ها ، socket API میگن.
- Socket : اون interface ای که بین لایه ی اپلیکیشن و لایه ی transport که توسط اون می تونیم پیام ها رو از لایه ی اپلیکیشن به لایه ی transport انتقال بدیم ،و یا برعکس.

مثلا در داخل یه آپارتمان تعدادی واحد وجود داره و هر واحد یه شماره ای داره ، در آپارتمان رو می تونیم به socket تشبیه کنیم .و اگه یه واحد بخواد پیام خودش رو به یه واحد دیگه در اون آپارتمان یا حتی آپارتمان های دیگه بفرسته ، میدونه که بیرون از در آپارتمان ،یه مکانیزمی وجود داره که این پیام رو منتقل کنه.(سرایدار میاد پیام و بر میداره میندازه تو صندوق پست، مامور پست برش میداره، آدرس مقصدو می بینه ، و به مقصد می رسونه :/)

در واقع اون سکنه ی آپارتمان ، مثل پروتکل ها و اپلیکیشن های لایه ی اپلیکیشن هستن ، اون دری که سکنه استفاده می کنن تا پیام رو منتقل کنن، نقش socket رو داره ، اون سرایدار ، نقش لایه ی transport

رو داره ، و اون مامور صندوق پست که از یه صندوق پست به صندوق پست دیگه نامه می بره، شبیه لایه ی network هست.

البته ما فقط آدرس آپارتمان برامون کفایت نمی کنه و باید شماره ی واحد رو هم داشته باشیم ، برای اینکه ی process توی کل شبکه به طور unique تعیین بشه ، باید علاوه بر آدرس host (همون آدرس IP ۳۲ بیتی) ، یه آدرس دیگه هم داشته باشه. این آدرس ، port number هست که میاد process رو داخل اون host مشخص می کنه.

- Port number ، ۲ بایت هست و ما میتونیم ۶۵ هزار و خورده ای عدد در این آدرس داشته باشیم(کلا رنجش از صفر تا تقریبا ۶۵ هزار هست).
- بعضی از این Port number ها ، به یه سرویس خاصی تخصیص داده شدن که سرور های اون سرویس های خاص، از این اعداد استفاده می کنن. مثلا توی اپلیکیشن وب ، که از پروتکل HTTP استفاده می کنیم ، Port number مون ، 80 هست. یا مثلا Port number برای mail server ، عدد 25 هست.

حالا چرا این عدد فیکسه ؟ چون اگه یه کلاینتی بخواد با سرور ارتباط برقرار کنه، نمیدونه Port number چه عددیه و هر عددی بین صفر تا ۶۵ هزار میتونه باشه .

پس سرور های اپلیکیشن های معروف ، بهشون یه Port number تخصیص داده شده که کلاینت ها علاوه بر آدرس IP، Port number شون رو هم بلد باشن تا بتونن اون پیام ها و درخواست هایی که برای یه سرویس دارن رو برای سرور ها ارسال کنن.

- یه host ای میتونه همزمان هم HTTP server برای اپلیکیشن وب داشته باشه و هم برای ایمیل ، mail server داشته باشه. چون Port number هاشون مختلفه ، شبیه اون واحد های مختلفی هستن که داخل یه آپارتمانن و لایه ی transport با توجه به این Port number میاد تمام بسته هایی که به پست میرسه رو به دست process متناظر اون بسته می رسونه.
- پس برای فرستادن یه HTTP message به یه آدرس وب سرور، یه آدرس IP داریم، و چون مربوط به اپلیکیشن وب هست ، مرورگر Port number رو برابر با 80 قرار میده و پیام رو می فرسته.

نتیجه :

- یه توسعه دهنده یا میتونه از TCP Socket یا UDP Socket استفاده کنه . توی TCP ، بسته ها به طور reliable و به شکل byte-stream مثل یه لوله منتقل میشن و در طرف مقابل دریافت میشن ، ولی توی UDP نباید انتظار یه کانال ارتباطی امن داشته باشیم که بسته ها خراب یا گم نشن یا ترتیبشون به هم نریزه. ولی خوبی ای که داره اینه که تاخیرش خیلی کمتر هست.

- ویدیوی ۶ نوشته نشده :/

یه برنامه ساده تحت شبکه با زبان پایتون نوشته میشه. شامل دوتا فایل ، یکی برای کلاینت و یکی برای سرور که کلاینت یه پیامی رو به سرور منتقل می کنه. این کارو با سوکت UDP انجام میدیم.

- توی ویدیوی هفت هم همین کار رو با سوکت TCP انجام میدیم.