UDP : User Datagram Protocol

- پروتکل UDP یکی از پروتکل های ساده ی لایه ی حمل و نقله. به عبارتی "no frills" یا "bare bones" عه ، یعنی بدون آرایش و هیچ افزونگی .
- همچنین "best effort" عه یعنی سگمنت ها ممکنه گم بشن یا ترتیبشون عوض بشه.
- UDP ، connectionless عه یعنی قبل از ارسال داده هیچ هماهنگی ای بین گیرنده و فرستنده صورت نمی گیره. همچنین سگمنت هایی که توسط UDP ارسال میشن مستقل از هم باهاشون برخورد میشه و هیچ state ای در UDP نگه داری نمیشه.
- چرا از UDP استفاده می کنیم؟

این سادگی در کنار اینکه کارایی رو کمتر می کنه ، یه سری خوبی هایی هم داره:

1 - چون نیازی نیست برای انتقال داده ها کانکشنی صورت بگیره، تاخیر اولیه ی ارتباط کم میشه.(تاخیر RTT رو نداریم)

2 - پیاده سازیش ساده ست. چون سمت فرستنده و گیرنده هیچ state ای حفظ نمیشه.

3 - چون کار زیادی انجام نمیده، هدرش سایز کمی داره.

4 - مکانیزم congestion control توی UDP فعال نیست یعنی مستقل از اینکه شرایط شبکه چجوری باشه ، هرموقعی که بسته ای از لایه ی شبکه به لایه ی حمل و نقل و به پروتکل UDP تحویل داده میشه ، UDP مبادرت به ارسال سگمنت می کنه. این قضیه باعث میشه حتی وقتی شرایط شبکه بد هست ، بتونیم توسط UDP بسته هامون رو ارسال کنیم و سرعت ارسالمون تابعی از شرایط شبکه نباشه.

- یه سری از پروتکل ها هستن که از UDP استفاده می کنن :

1 - Streaming multimedia apps : این اپلیکیشن ها loss tolerant هستن یعنی تا حدودی اگه بسته ها گم بشه می تونن تحمل کنن و در عین حال rate sensitive هم هستن یعنی به پهنای باند و سرعت ارسال حساس هستن. به این دلایل UDP یک گزینه ی مناسب برای استفاده ی این اپلیکیشن ها هست.

2 - DNS : از UDP استفاده می کنه که دلیلش سرعت بالاتر و هدر کوچکترش نسبت به TCP عه(overhead کمتری داره)

3 - SNMP : پروتکل مدیریت شبکه ست که برای مدیریت روتر ها و gateway ها استفاده میشه. علاوه بر اینکه سرعت UDP توی این پروتکل مهم هست، یه بحث دیگه هم وجود داره: باید مواقعی که شرایط شبکه مناسب نیست هم پیام های کنترلی برای روتر ها و

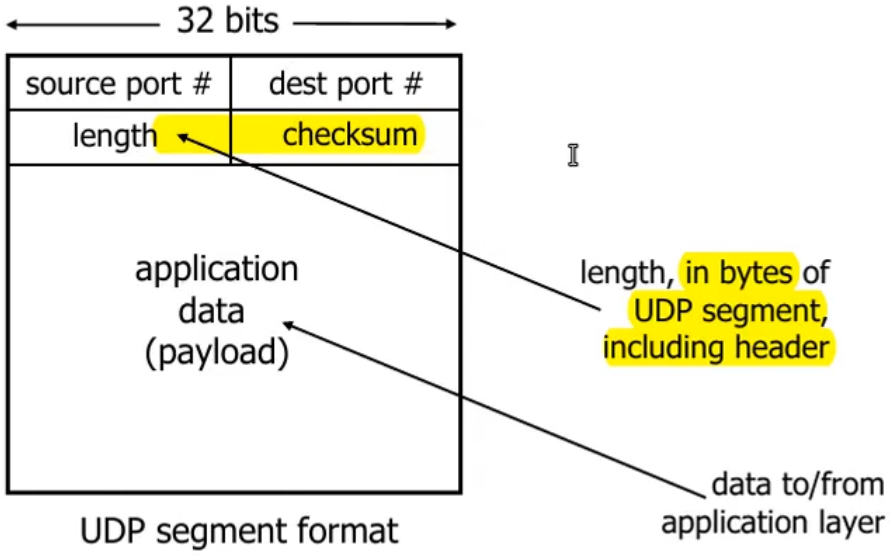
ادوات شبکه ارسال کنیم ؛ اگه بخوایم از TCP استفاده کنیم(که مکانیزم congestion control داره) در مواقعی که شرایط شبکه بد هست ما نمیتونیم با rate خوبی، برای ادواتمون بسته های کنترلی ارسال کنیم. پس نبود مکانیزم congestion control باعث شده که از UDP در SNMP استفاده کنن.

4 - HTTP/3 : نسخه ای از HTTP ایه که مبتنی بر UDP هست. اگه یه سری مکانیزم دیگه هم بخوایم(مثل reliable data transfer که وجودش توی HTTP که file transfer داریم ، ضروریه) ، باید اون ها رو از طریق لایه ی اپلیکیشن فراهم کنیم.

- RFC ِ متناظر با پروتکل UDP ، RFC 768 هست که خیلی مختصره.

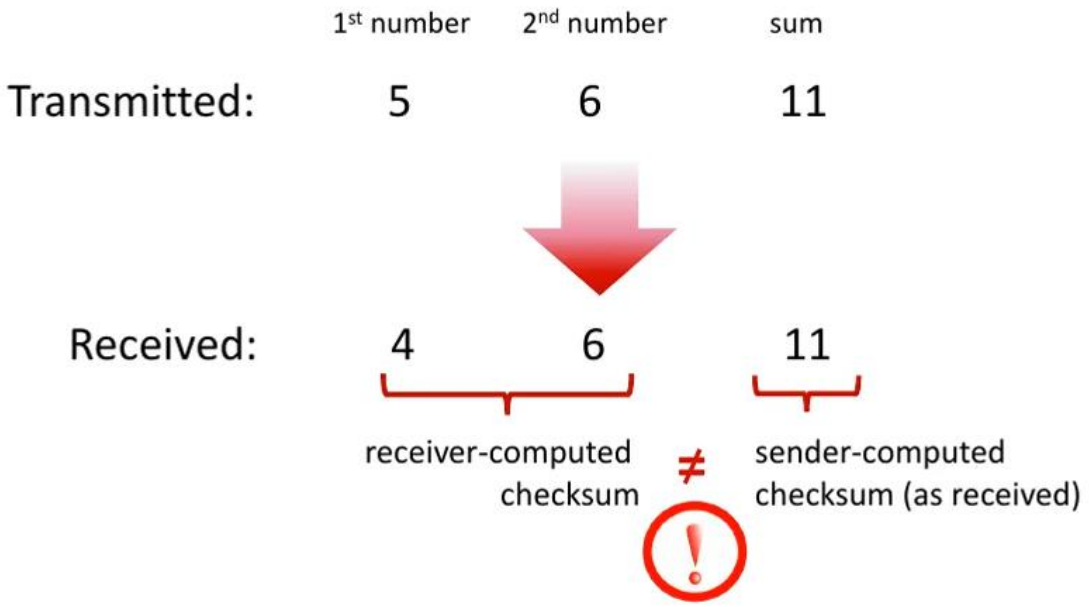
- UDP segment header

- فرمت سگمنت های UDP به این نحوه که یه بخش data داره که همون داده ایه که از لایه ی اپلیکیشن دریافت میشه. همچنین یه بخش header داره که خیلی متخصره . کلا 8 بایته، 4 بایت اول برای شماره ی پورت هاست ، بایت های 5 و 6 مربوط به فیلد length هستن که طول کل سگمنت UDP به همراه هدرش رو برحسب بایت نشون میده. فیلد آخر checksum عه که 2 بایت داره و برای آشکارسازی خطا به کار میره.



- UDP checksum

- هدف این فیلد توی سگمنت UDP ، آشکار سازی خطا(error) هست.
منظور از error اینه که سگمنتی که ارسال میشه ممکنه بعضی از بیت هاش flipped بشن(صفر ها به یک و یک ها به صفر تبدیل بشن) و تا حدی میخوایم این خطا ها رو detect کنیم.
- مثال (برای اعداد صحیح) : فرض کنیم میخوایم دوتا عدد 5 و 6 رو به گیرنده بفرستیم . روش checksum اینجوریه که علاوه بر این دو عدد، جمعشون رو هم میفرسته ، و بعد 3 تا عدد رو ارسال می کنه.
اگه موقع ارسال یکی از دو عدد عوض بشن، گیرنده میاد جمع دو عدد دریافت شده رو با اون عدد سومی که دریافت کرده(که جمع دوعدد قبل از ارسال بوده) مقایسه می کنه و اگه یکسان نباشن متوجه میشه یه خطایی رخ داده.



البته این مکانیزم آشکارسازی خطا قدرتمند نیست و ممکنه به جای یه عدد ، دو عدد به نحوی تغییر کنن که جمعشون همون مقداری بشه که سمت فرستنده بوده .

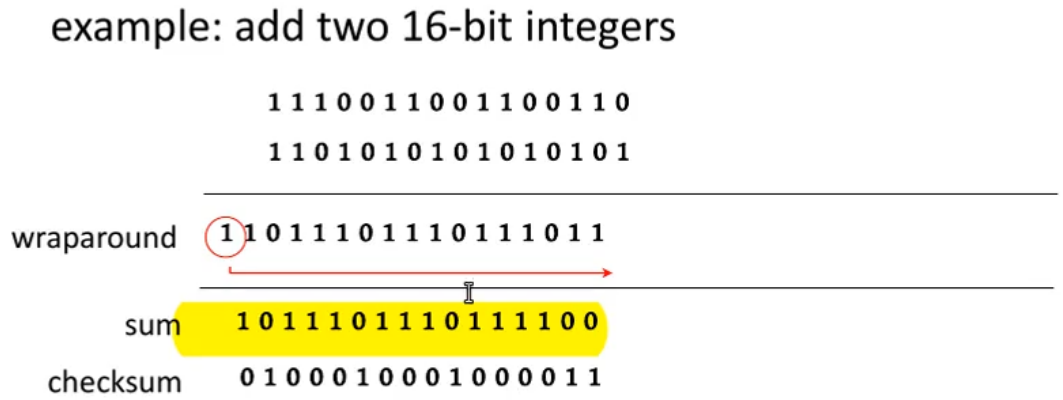
این اصل همون اصلیه که در UDP برای سادگی آشکار سازی خطا اعمال میشه. یعنی سمت فرستنده ، سگمنت ها رو 16 بیت 16 بیت در نظر می گیریم و با هم جمع می کنیم و بعد مکمل یک حاصل جمع رو در checksum قرار میدیم.

سمت گیرنده ، میاد checksum سگمنت دریافت شده رو محاسبه می کنه و با فیلد checksum مقایسه می کنه و اگه یکسان بودن فرض رو بر این میذاره که خطایی رخ نداده. اما اگه برابر نبودن، دیگه حتما می تونیم بگیم که خطایی رخ داده.

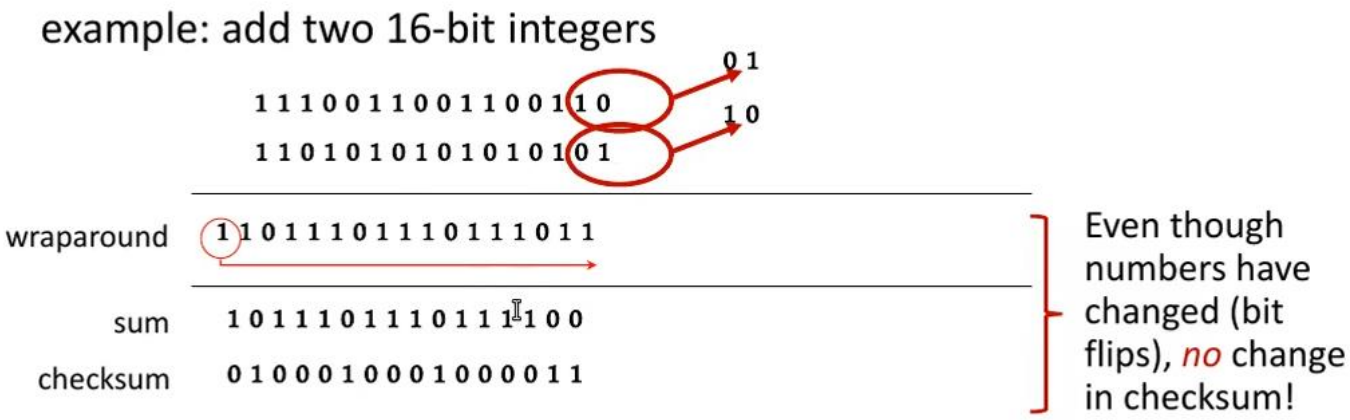
- مثال از checksum:

نکته ای که توی این مثال هست اینه که اگه دو عدد رو جمع کردیم و carry out ایجاد شد ، اون رو wraparound می کنیم یعنی برش

می گردونیم و با بیت کم ارزشمون دوباره جمع می کنیم. بعد مکمل یکش می کنیم و تو فیلد checksum قرارش میدیم.



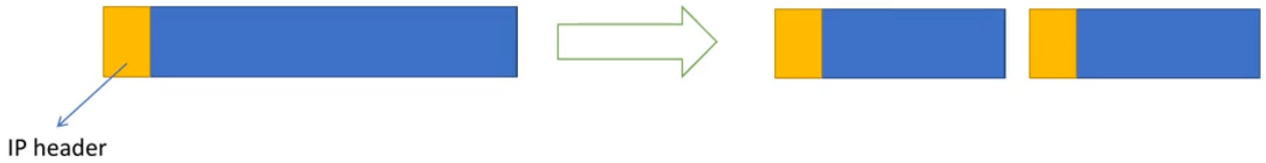
- مثال : (از اینکه بعضی وقتا checksum نمیتونه خطا رو تشخیص بده)



- UDP & Fragmentation

- Fragmentation یعنی قطعه قطعه شدن.
- توی لایه دوم (Link layer) اندازه ی frame از یه حد به خصوصی بیشتر نمیتونه باشه که بهش میگن MTU(Maximum Transmission Unit)

مثلا توی Ethernet طول frame بیشتر از 1500 بایت نمیتونه باشه. در این صورت اگه یه بسته ی IP (مربوط به لایه ی شبکه) طولش جوری باشه که نتونه کلش در داخل یه frame قرار بگیره ، اون بسته ی IP به بسته های کوچکتری شکسته میشه که این بسته های کوچکتر هرکدوم بتونن داخل یه frame جا بشن :



- Fragmentation تو لایه ی IP هرکدوم از تجهیزات شبکه (از end system ها تا روتر ها) میتونه انجام بشه. یعنی در طول مسیر اگه بسته مثلا به یک روتری برسه و لینکی که به اون روتر متصله ، MTU اش به نحویه که بسته کلا در داخل frame هاش جا نمیشه ، همون روتر در داخل لایه ی شبکه ش میاد این عمل fragmentation رو انجام میده. از اون به بعد reassembly انجام نمیشه و بسته های کوچکتر راه خودشون رو به صورت مجزا طی می کنن تا به گیرنده برسن(حالا در

ادامه هر کدوم از این ها ممکنه به گیرنده برسه یا گم بشه) و این لایه ی شبکه ی سمت گیرنده ست که عملیات reassembly رو انجام میده.

- نکته ای که هست اینه که تا جایی که میشه باید از fragmentation پرهیز کنیم، چون لایه ی شبکه هیچ گونه تلاشی برای ریکاوری انجام نمیده! یعنی اگه یکی از این fragment ها گم بشن ، لایه ی شبکه هیچ re-transmission ای انجام نمیده تا بتونیم از طریق این کار fragment ای که گم شده رو ریکاوری کنیم و کل بسته رو بسازیم.
اگه fragment ای گم شد ، بعد مدتی لایه ی شبکه IP ِ سایر fragment ها رو هم دور می ریزه و احیانا اگه اون بسته توسط TCP ارسال شده بوده، باید مجددا ارسال بشه ، و اگه توسط UDP ارسال شده بوده و مهم بوده ، باید تو لایه ی اپلیکیشن مجددا اون بسته ساخته بشه و به UDP سمت فرستنده داده بشه و مجددا ارسال بشه.
این قضیه باعث از بین رفتن مقدار زیادی از پهنای باند میشه ، چون بسته خودش به اندازه ی کافی بزرگ بوده ، و اگه یه fragment گم بشه کل بسته دور ریخته میشه یا باید مجدد ارسال بشه .
چون لایه ی شبکه از fragmentation ساپورت ِ ریکاوری نمی کنه ، میگیم fragmentation فرایند مطلوبی نیست.
- ما خیلی راحت می تونیم کاری کنیم که توی UDP ، fragmentation صورت بگیره. چرا ؟ چون UDP هر بسته با طولی که ما بهش بدیم رو ارسال می کنه ،(البته یه لیمیت 65536 داره به

خاطر فیلد length که حداکثر دوبایته) اما تا اون مقدار ماکزیمم که میتونیم در اختیار UDP قرار بدیم ، UDP دیگه توجه نمی کنه که MTU در مسیر به چه نحوه یا باید پیامو بشکونه و چیزیو مدیریت کنه که fragmentation رخ نده یا .... UDP فقط میاد بسته ای که دریافت کرده هدرش رو هم بهش اضافه می کنه و تحویل لایه ی شبکه میده . حالا یا این بسته توی لایه ی شبکه ی فرستنده fragment میشه ، یا توی مسیر ممکنه مشکل MTU پیدا کنیم و عمل fragmentation رخ بده.

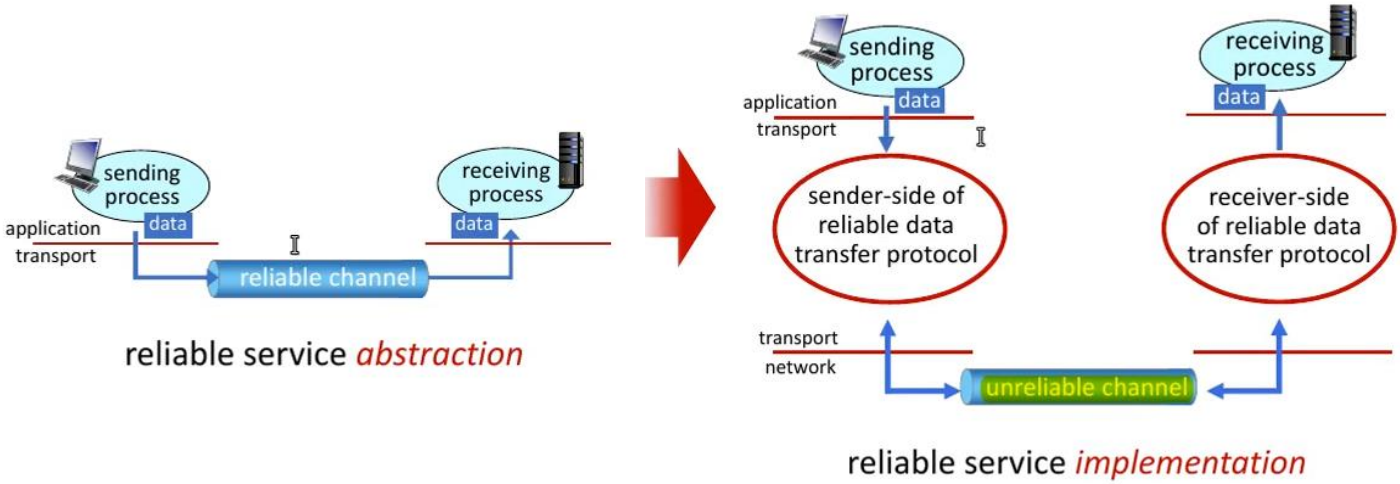
- TCP تا جایی که ممکنه تلاش می کنه که fragmentation رخ نده.

- **Reliable data transfer (RDT)**

- Reliable data transfer از موضوعات خیلی مهم توی شبکه های کامپیوتریه . بحث RDT فقط توی لایه ی حمل و نقل مطرح نیست ، بلکه توی لایه های اپلیکیشن و لینک هم مطرحه.
- وقتی یه پروتکل لایه ی حمل و نقل سرویس RDT رو ارائه می کنه ، یعنی اطلاعاتی که توسط process فرستنده در یک end system برای process گیرنده بر روی یه end system دیگه ارسال میشه ، و برای این کار ، این دوتا process دارن از پروتکل RDT استفاده می

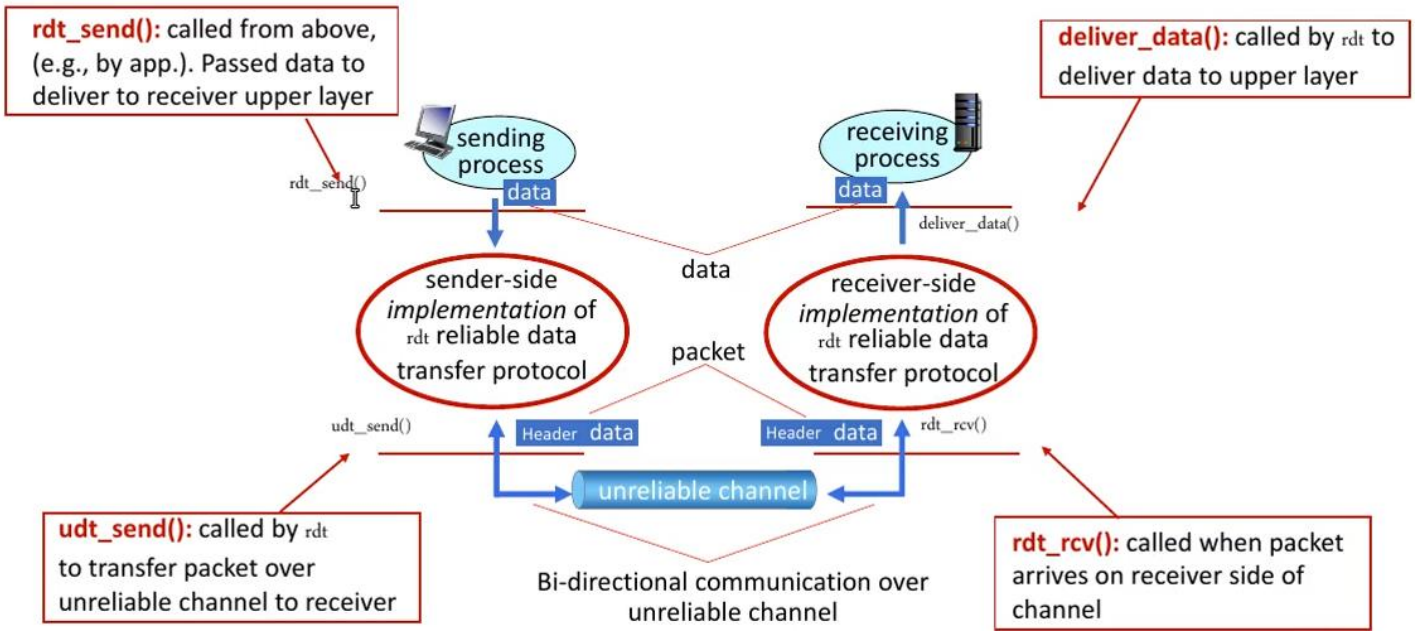
کنن، اطلاعات بدون هیچ خطایی منتقل میشه . به اصطلاح میگن یه کانال مطمئن مجازی بین فرستنده و گیرنده وجود داره.

- چالشی که لایه ی حمل و نقل باهاش مواجهه که سرویس RDT رو ارائه کنه ، اینه که اون کانال فیزیکی که بین فرستنده و گیرنده وجود داره ، یه کانال unreliable عه و بسته هایی که ما به لایه ی حمل و نقل میدیم که به دست end system گیرنده برسونه ، پروتکل لایه ی حمل و نقل باید این بسته ها رو از این کانال نامطمئنِ واقعی به دست گیرنده برسونه که ممکنه بسته ها توش گم بشن یا ترتیبشون بهم بریزه یا ... پس باید این کانال نامطمئن رو تبدیل به یه کانال مطمئن کنه. یعنی وقتی که process ها نگاه می کنن به پایین ، یه کانال مطمئن مجازی می بینن که هرچی دارن تحویل میدن بی هیچ خطایی در سمت دیگه دریافت میشه ، اما پروتکل RDT چیزی که می بینه یه کانال نامطمئن هست.



نکته ی مهمی که شکل های بالا دارن اینه که در قسمت سمت چپ ، کانال مطمئنمون یه کانال یه طرفه بین فرستنده و گیرنده ست و هر اطلاعات مهمی که می فرستیم بدون اینکه مشکلی براش پیش بیاد در سمت گیرنده دریافت میشه. برای پیاده سازی یه کانال مطمئن یه طرفه، باید هم فرستنده و هم گیرنده برای هم بسته ارسال کنن تا یه کانال مطمئن مجازی از سمت فرستنده به گیرنده ایجاد بشه.

- میزان پیچیدگی پروتکل های RDT چقدر هست؟ جواب این سوال بستگی به ویژگی های کانال نامطمئن داره. به اینکه آیا کانالمون بسته ها رو گم می کنه ، یا ترتیبشون رو به هم میزنه یا خطا توشون ایجاد می کنه یا ...
- پروتکل های RDT در سمت فرستنده و گیرنده ، state همدیگه رو نمیدونن، مثلا اگه یه بسته ای از فرستنده ارسال بشه و توسط گیرنده دریافت بشه فرستنده نمیفهمه که دریافت شده یا نه.(مگه این که توسط پیامی به فرستنده ارسال بشه)
- برای توصیف پروتکل های RDT احتیاج داریم که قراردادی رو راجع به API های یک پروتکل RDT فرضی با لایه های بالا و پایین (یعنی لایه ی اپلیکیشن و شبکه) در نظر بگیریم. توی شکل زیر ، API ها برای یک پروتکل RDT فرضی نمایش داده شده.



- به طور خاص ، یه API داریم تحت عنوان rdt_send() که لایه ی اپلیکیشن وقتی میخواد دیتایی رو به پروتکل RDT برای ارسال بده، از چنین API ای استفاده می کنه.
- یه API دیگه ، udt_send() هست که وقتی پروتکل RDT یه بسته ای رو ساخته و توسط کانال نامطمئن(سرویسی که لایه ی شبکه ارائه کرده) بفرسته، این تابعو کال می کنه(udt مخفف unreliable data transfer هست)
- وقتی یه بسته ای قراره از سمت لایه ی شبکه (گیرنده ی کانال) تحویل پروتکل RDT (در سمت گیرنده) داده بشه، فانکشن rdt_rcv() صدا زده میشه و پروتکل RDT سمت گیرنده بسته رو تحویل می گیره.

- یه فانکشن دیگه هم به اسم deliver_data() داریم که هر موقع پروتکل RDT گیرنده بخواد بسته رو تحویل process سمت گیرنده بده، این فانکشنو کال می کنه.

- Reliable data transfer : getting started

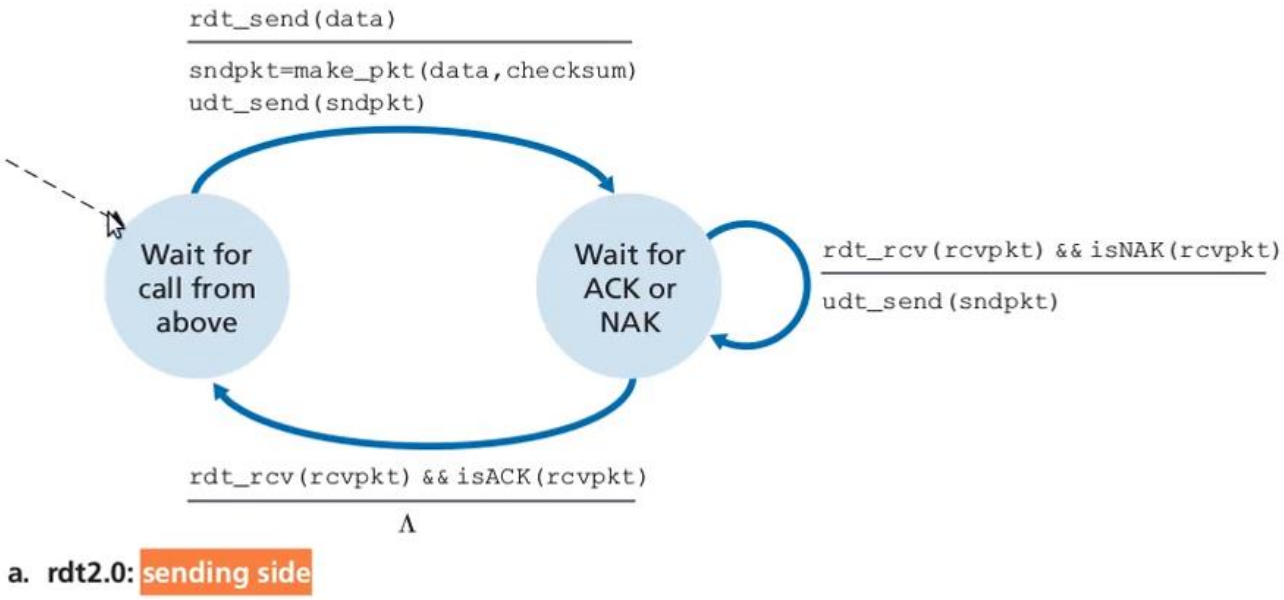
- از یک کانال ساده و ایده آل شروع می کنیم و به تدریج اون اثراتی که کانال ممکنه روی بسته هایی که بهش تحویل میدیم ایجاد کنه رو اضافه می کنیم.
- توی پروتکل های RDT که راجع بهشون صحبت می کنیم، مسیر ارسال اطلاعات رو یک طرفه در نظر می گیریم.(اما برای پیاده سازی در عمل هم فرستنده و هم گیرنده بسته ارسال می کنن)
- معمولا برای توصیف پروتکل های RDT ، از FSM(Finite State Machine) ها استفاده میشه. برای این بهش میگن finite (محدود) که تعداد state ها محدود هست.

توی FSM ها چرخه ی حیات پروتکل به صورت مدام بین چندتا حالت در چرخشه و به هر حالت یه state میگیم.

برای بیان شرایط ورود و خروج ، از یک state به state دیگه ، از یه سری بردار استفاده میشه و این بردار ها هرکدوم دوتا label دارن که توسط یه خط افقی از هم جدا میشن .label ای که بالا نوشته میشه مربوطه به اتفاقیه که افتاده (event) و باعث شده ما از یه state به یه

state دیگه منتقل بشیم. و اونی که پایین خط افقی نوشته میشه ، action یا عملیه که ضمن انتقال از یه state به state دیگه توسط پروتکل انجام میشه.

مثال :



a. rdt2.0: sending side

- اون فلشی که به صورت خط چینه ، نشون میده که اگه FSM بیش از یک state داره ، در ابتدا پروتکل در کدوم state قرار داره.
- یه نماد دیگه ای که استفاده می کنیم، Λ (لاندا) عه که هر موقع event یا action خالی هست یا یعنی به ازای یه اتفاقی از یه state به state دیگه می ریم، ولی کار خاصی انجام نمیدیم، یا کاری انجام می دیم ، اما به ازای اتفاق خاصی اون کارو انجام نداده باشیم، توی قسمت label پایین یا بالا Λ رو می ذاریم.