rdt 1.0 : Reliable transfer over a reliable channel

- توی این جلسه برای کانال ها با شرایط و ویژگی های مختلف بررسی می کنیم که ساختار پروتکل rdt باید به چه نحو باشه تا نهایتا بتونه سرویس کانال مطمئن مجازی رو برای لایه ی اپلیکیشن فراهم کنه.
- بحثمون رو با کانال ایده آل شروع می کنیم. اسم نسخه ای از پروتکل rdt که برای یک کانال مطمئن فیزیکی به کار میره ، rdt 1.0 هست.
- منظورمون از کانال ایده آل اینه که هیچ گونه خطای بیتی رخ نمیده، هیچ گونه گم شدگی بسته رخ نمیده ، و احیانا با یه تاخیری، بسته ای که فرستنده ارسال می کنه ، در سمت دیگه ی کانال دریافت میشه . اتفاقات ناخوشایند دیگه مثل به هم ریختن ترتیب بسته های ارسالی، duplicate شدن یک بسته(یه بسته بفرستیم ولی دو نسخه یا بیشتر به دست گیرنده برسه) هم رخ نمیده.
- در این حالت FSM فرستنده و گیرنده هرکدومشون تنها یک state دارن.در سمت فرستنده ما همیشه منتظر این هستیم که یه داده ای از لایه ی اپلیکیشن دریافت کنیم ، به محض اینکه توسط فانکشن rdt_send() که لایه ی اپلیکیشن اونو فراخوانی می کنه، یک داده به

ما تحویل داده شد ، میایم از روی اون داده یک بسته می سازیم ، مثلا یک سری هدر بهش اضافه می کنیم که مثلا مشخص بشه مقصد این بسته کجا هست ، و بعد میایم اون بسته رو با کال کردن فانکشن udt_send() در اختیار کانال سمت فرستنده قرار میدیم تا اون رو ارسال کنه.

سمت گیرنده هم ما فقط یک state داریم، همیشه منتظر هستیم که یه بسته از سمت دیگه ی کانال که سمت گیرنده هست دریافت بشه، و وقتی که یک بسته دریافت شد و توسط فانکشن rdt_rcv() اون بسته در اختیار rdt سمت گیرنده قرار گرفت، میاد قسمت داده رو از توی بسته می کشه بیرون(با فانکشن extract()) و بعد این داده رو توسط فانکشن deliver_data() در اختیار لایه ی اپلیکیشن قرار میده .

در این حالت rdt ساختار ساده ای داره چون اون کانال فیزیکی ای که بین فرستنده و گیرنده هست کانال مطمئنیه و اتفاق ناخوشایندی توش رخ نمیده و به خاطر همین ایجاد یه کانال مجازی مطمئن از روی یک کانال فیزیکی مطمئن ، احتیاج به کار خاصی نداره. پروتکل rdt هم توی این حالت عمدتا نقش رابط رو بین لایه ی اپلیکیشن و لایه ی شبکه ایفا می کنه.

- rdt 2.0 : channel with bit errors

- این کانال یک مرتبه به کانال واقعی نزدیک تره. توی کانال ممکنه خطای بیتی رخ بده یعنی بسته هایی که به کانال داده میشه تا به گیرنده برسونه سمت گیرنده بعضی بیت ها ممکنه flipped شده باشن. اما بقیه ی اتفاقات ناخوشایند رو فعلا ازشون صرف نظر می کنیم.
- عدد 0 توی اسم این پروتکل rdt ، یعنی تلاش اول ما برای طراحی نسخه ی دوم پروتکل ما. تلاش بعدی میشه rdt 2.1 و ....
- برای مقابله کردن با خطا توی rdt از چه روشی می تونیم استفاده کنیم؟

مثل مادری که برای فرزندش دیکته میگه و اگه فرزند درست بشنوه ، میگه «خب» و منتظر کلمات بعدی می مونه، و اگه درست نشنوه از مادر میخواد دوباره براش تکرار کنه.

- استراتژی اصلی که برای مقابله با خطا توی rdt وجود داره، ARQ(Automatic Repeat Request) هست که متشکل از این قسمت هاست:

1 - ما باید یه checksum داخل پیام هایی که میفرستیم داشته باشیم تا سمت گیرنده متوجه بشه که آیا بیتی توش خطا رخ داده یا نه.

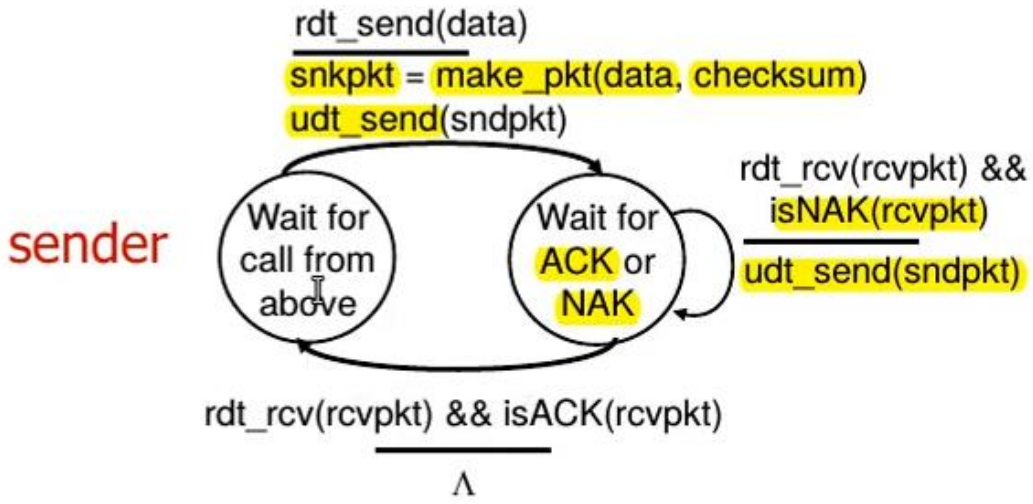
2 - ارسال Acknowledgments(ACKs) : هر موقع گیرنده یک بسته ای رو دریافت کرد و متوجه شد که خطایی در داخلش رخ نداده ، یه سیگنال ACK برای فرستنده ارسال می کنه .

3 - ارسال Negative Acknowledgments(NAKs) : هر موقع گیرنده یک بسته ای رو دریافت کرد و متوجه شد که خطایی در داخلش رخ داده ، یه سیگنال NAK برای فرستنده ارسال می کنه. از اون سمت فرستنده هم اگه این سیگنالو دریافت کنه بسته رو مجدد برای گیرنده ارسال می کنه.

به کل این مجموعه میگیم Automatic Repeat Request .

- یه مکانیزم دیگه ای که توی rdt استفاده می کنیم stop and wait هست.به این مفهوم که فرستنده فقط یک بسته ارسال می کنه و منتظر پاسخ گیرنده میشه (ACK یا NAK) اگه ACK دریافت کرد از حالت انتظار برای دریافت پاسخ خارج میشه.(میتونه بره ببینه اگه بسته ی دیگه ای هست اونو ارسال کنه) و در صورتی که NAK دریافت کرد ، مجددا باید بسته ی قبلی رو ارسال کنه و همچنان منتظر پاسخ گیرنده می مونه تا نهایتا یک ACK دریافت کنه.
- با این اوصاف FSM سمت فرستنده ی rdt 2.0 به این شکله که دوتا state داریم ، state اولیه به این نحوه که ما منتظر دریافت داده ای از لایه ی بالا هستیم ، (wait for call from above) و وقتی یه داده

ای توسط فانکشن rdt_send() به rdt سمت فرستنده تحویل داده شد، یه بسته از اون داده می سازیم که حاوی فیلد checksum عه. اسم این بسته رو sndpkt می ذاریم و توسط فانکشن udt_send() در اختیار کانال قرار میدیم (که میدونیم ممکنه برخی از بیت های بسته رو خراب کنه). بعد state مون عوض میشه منتظر دریافت ACK یا NAK از گیرنده میشیم. در صورتی که بسته ای دریافت بشه و اون بسته NAK باشه، تو همون state می مونیم و اگه ACK باشه، کار خاصی انجام نمیدیم و فقط state عوض میشه. الان مجاز هستیم که اگه داده ی جدیدی وجود داره شروع به ارسالش کنیم.



- سمت گیرنده توی این حالت، FSM فقط یه state داره، و اون هم اینه که همیشه منتظره یه بسته ای از کانال دریافت کنه.اگه بسته دریافت شد ولی خراب بود، توسط udt_send() یک NAK ارسال می کنه، و

اگه بسته خراب نبود، میاد قسمت داده رو از بسته خارج می کنه ، بسته رو در اختیار لایه ی بالا قرار میده، (چرا می تونه اینکارو انجام بده؟ چون فرض کردیم re-order توی کانال صورت نگرفته و بسته ها ترتیبشون به هم نریخته) نهایتا یک ACK هم توسط udt_send() برای فرستنده ارسال می کنه.

- وقتی ما در state فرستنده هستیم از state گیرنده خبر نداریم مگه اینکه از طریق ارتباطی که با گیرنده داریم بفهمیم state گیرنده چی هست. دلیل ارسال این ACK ها و NAK ها در سمت گیرنده همینه.

- برای اینکه با نحوه ی خوندن FSM و دنبال کردن اتفاقاتی که میتونه در یه پروتکل rdt صورت بگیره آشنا بشیم، دوتا مثال می زنیم.
  - مثال اول : هیچ گونه خطایی در کانال رخ نمیده (rdt 2.0)

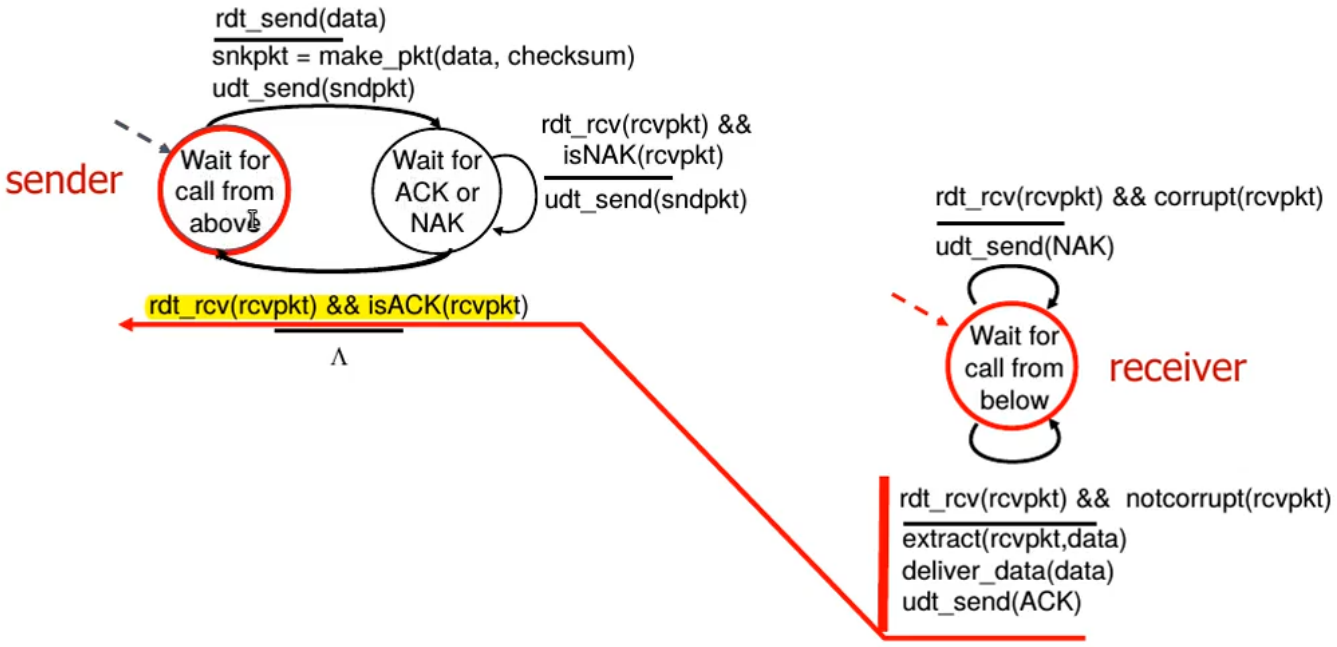
در این صورت با فرض اینکه فرستنده و گیرنده در state های اولیه ی خودشون هستن، سمت فرستنده منتظر هستیم که از لایه ی اپلیکیشن داده برای ارسال دریافت کنیم و سمت گیرنده هم منتظر هستیم که یک بسته از لایه ی پایین دریافت کنیم.

اگه از لایه ی اپلیکیشن ، یه داده ای سمت rdt فرستنده قرار بگیره، متناظر با این اتفاق یه سری کارها باید انجام بشن تا نهایتا state ما از

منتظر بودن برای لایه ی اپلیکیشن(که داده بفرسته) تغییر به کنه به حالتی که منتظر دریافت ACK یا NAK هستیم.اما کارایی که باید انجام بشن، اینه که یه بسته باید از روی داده ساخته بشه(که checksum هم داخل اون بسته وجود داره) و بعد توسط udt_send() ارسال میشه.

حالا اینجا چون با یه الگوریتم توزیع شده روبرو هستیم، برای ادامه ی روند کار مهمه که در سمت مقابل چه اتفاقی میفته .

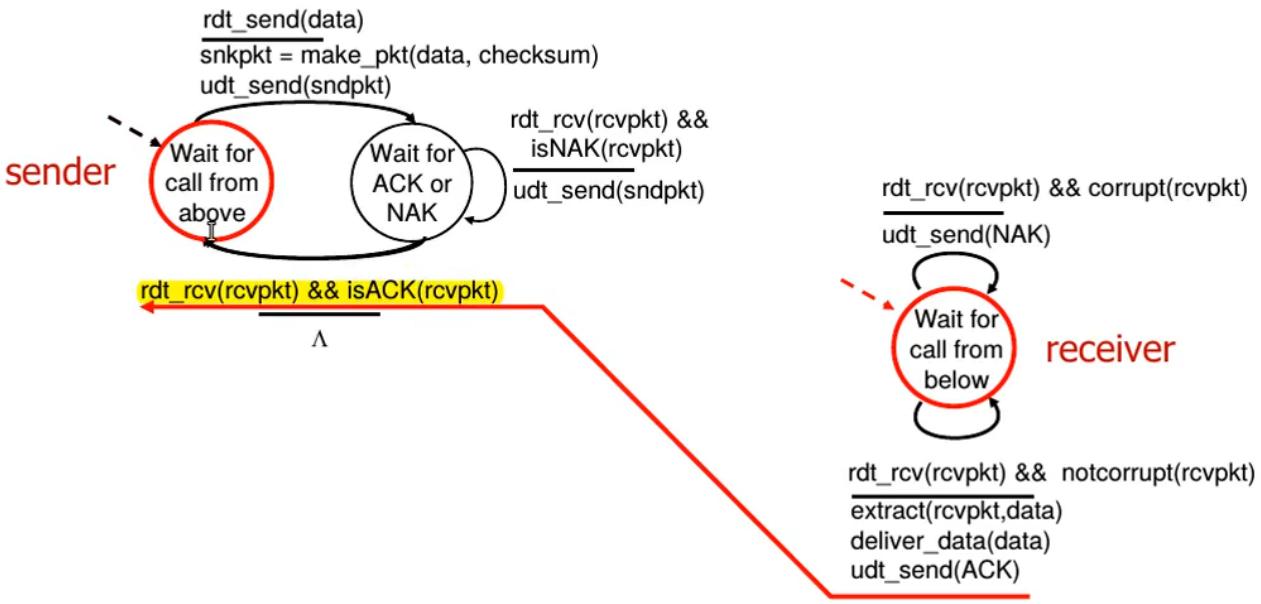
بسته ی ارسالی ما به گیرنده می رسه و چون ما فرض کردیم خطایی داخل کانال رخ نمیده، داده از بسته خارج میشه ، تحویل لایه ی اپلیکیشن داده میشه و یه ACK برای فرستنده هم ارسال میشه ، بنابراین سمت فرستنده state مون از wait for ACK or NAK به wait for call from above تغییر می کنه.



- مثال دوم : rdt 2.0 : corrupted packet scenario

- توی این مثال کانال در بسته ی اولی که فرستاده میشه خطا ایجاد می کنه اما در بسته های بعدی خطا ایجاد نمی کنه.
- ابتدا در سمت فرستنده ، منتظریم که داده ای از لایه ی اپلیکیشن دریافت کنیم ، بعد بسته از روی داده ساخته میشه ، و بعد بسته ارسال میشه(تا اینجا مشابه مثال قبل)

اما وقتی که بسته به گیرنده می رسه ، در عین حال که state مون سمت فرستنده عوض شده و منتظر دریافت ACK یا NAK هستیم، در سمت گیرنده چون فرض کردیم بسته ی اول دچار خطا شده، گیرنده میاد NAK ارسال می کنه . این NAK سمت فرستنده دریافت میشه و فرستنده میاد بسته ی قبلی که ساخته بود رو، مجددا ارسال می کنه. و بسته ی دومی که ارسال میشه سمت گیرنده فرض کرده بودیم که کانال توش خطایی ایجاد نمی کنه پس سمت گیرنده داده رو از بسته خارج می کنه و تحویل لایه ی اپلیکیشن میده و یه ACK برای فرستنده میده. و این بار سمت فرستنده چون ACK دریافت کردیم، لازم نیست کار خاصی انجام بدیم و فقط state مون عوض میشه .



- rdt 2.0 یه مشکل اساسی داره اون هم اینه که به طور ضمنی فرض شده که برای ACK و NAK خطایی رخ نمیده. در صورتی که ما میدونیم کانالی که می تونه در بسته ها خطا ایجاد کنه ، می تونه اون خطا ها رو در بسته های ACK و NAK هم ایجاد کنه. حالا اگه در بسته های ACK و NAK خطا رخ بده، فرستنده دیگه نمیدونه در سمت گیرنده چه اتفاقاتی افتاده، آیا بسته ای که به گیرنده رسیده خطایی درش رخ داده یا بدون خطا رسیده.

ممکنه در این شرایط بگیم که فرستنده می تونه بسته رو retransmit کنه ، اما این کار ممکنه باعث duplicate شدن بسته ها بشه. چرا؟ چون ممکنه بسته ی قبلی که ارسال کرده بودیم خطایی توش رخ نداده باشه و گیرنده اونو تحویل لایه ی اپلیکیشن داده باشه، ولی الان که به

جای ACK یه ACK مخدوش شده سمت فرستنده دریافت کردیم، اگه بیایم retransmit کنیم، دوباره گیرنده بسته ای که دریافت کرده رو تحویل لایه ی اپلیکیشن میده و این بسته ، یه بسته ی تکراریه.

- چطوری مشکل بسته های تکراری رو برطرف کنیم؟

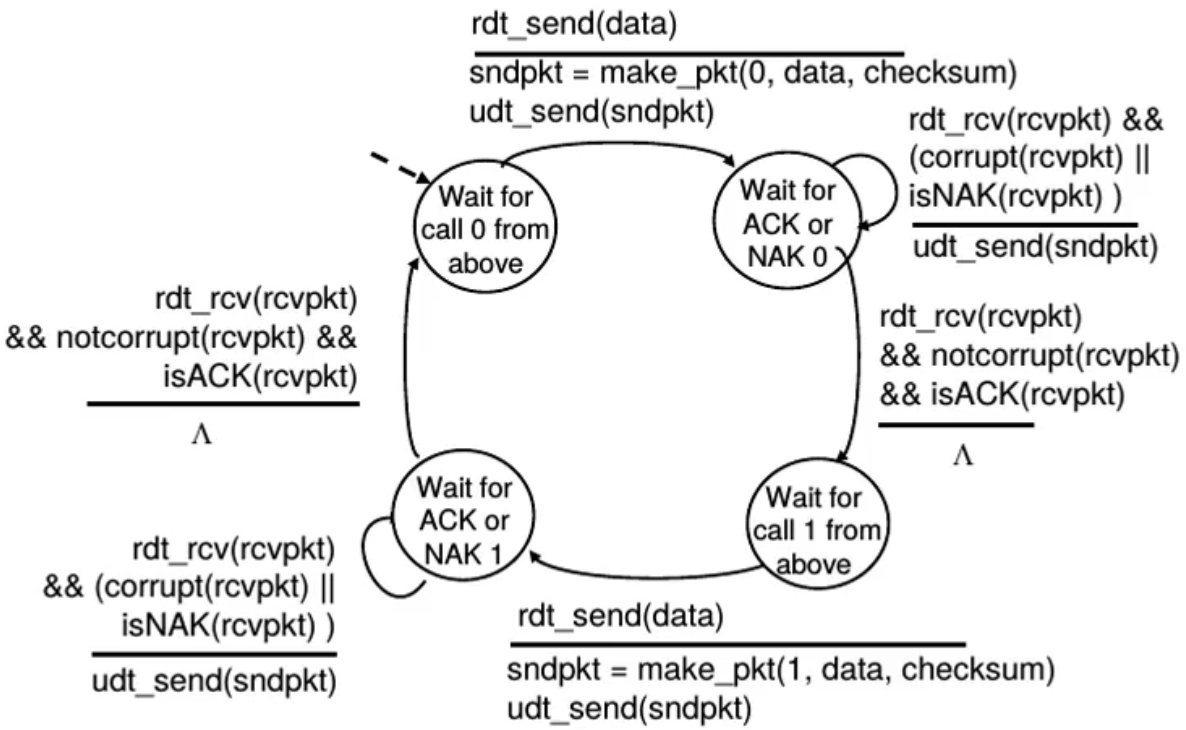
راهش اینه که در سمت فرستنده ما از sequence number استفاده کنیم، در این صورت سمت گیرنده می تونه شماره ی بسته های دریافتی رو چک کنه و اگه بسته ای تکراریه ، اونو دور بریزه.

- شماره گذاری بسته ها باید چجوری انجام بشه؟

از مکانیزم stop and wait استفاده می کنیم، به طوری که بسته ها رو به طور باینری شماره گذاری می کنیم(0, 1, 10, 11, ...) . چرا ؟ چون توی این مکانیزم فقط یک بسته ارسال می کنیم، اگه NAK یا یه بسته ی مخدوش دریافت کردیم، دوباره اون بسته رو ارسال می کنیم، و اگه ACK دریافت کنیم، میریم یه بسته ی جدید ارسال می کنیم.

پس به عبارتی یا بسته ی جدید ارسال می کنیم یا یه بسته ی قدیمی ، که اینو با 0 یا 1 نشون میدیم.

- FSM سمت فرستنده در rdt 2.1 (که این اصلاح ACK و NAK توش اعمال شده) به این شکله :

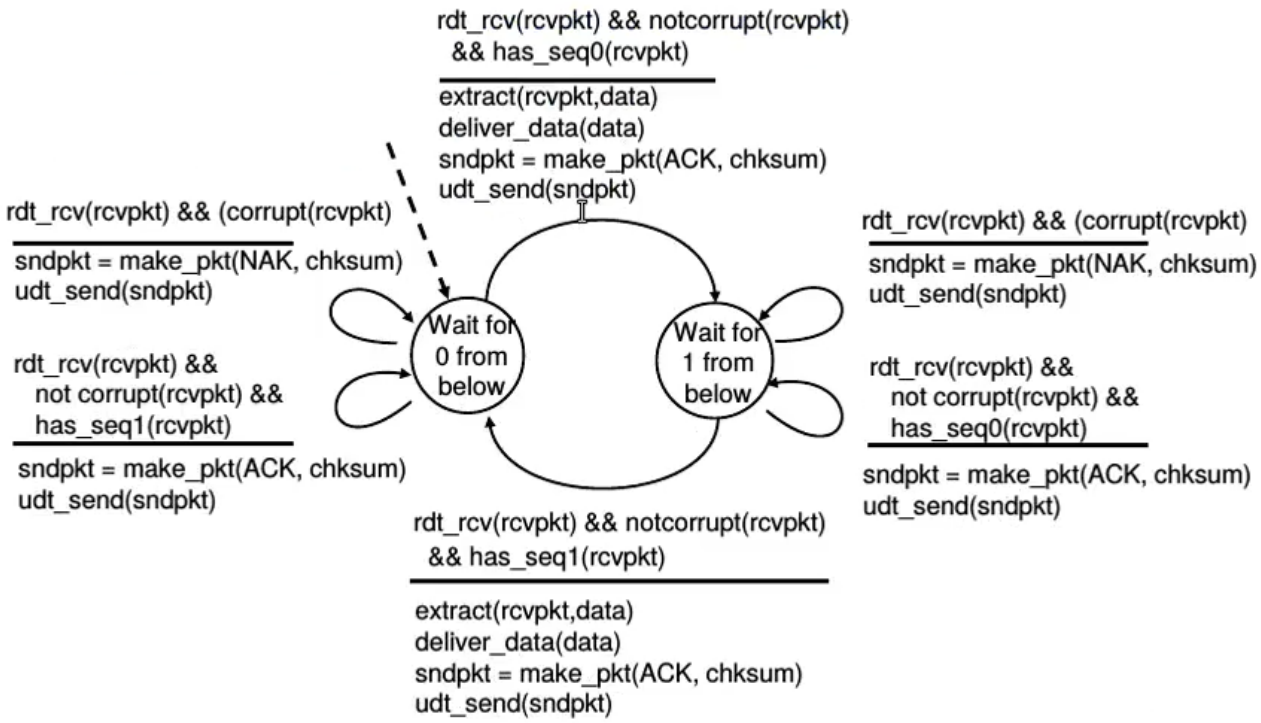


- تعداد state ها توی این FSM نسبت به rdt 2.0 دو برابر شده . چون ما اومدیم حالت هایی که متناظر با ارسال بسته های صفر و یک هستن رو مجزا در نظر گرفتیم ، و یک تقارن داخل این دیاگرام وجود داره.
- اگه از state اولیه شروع کنیم ، منتظر دریافت داده ای هستیم و وقتی دریافتش می کنیم بسته ای که ازش می سازیم رو با شماره ی 0 شماره گذاریش می کنیم.(توی تابع make_pkt() یه آرگومان دیگه برای sequence number هم اضافه شده) بعد بسته رو ارسال می کنیم و به state بعدی می ریم و منتظر دریافت ACK یا NAK می مونیم .
- توی state بعدی، یه بسته دریافت میشه ، و این بسته یا مخدوشه، یا اینکه NAK عه ، در هردوی این حالت ها مجددا به همین state بر می گردیم و بسته رو retransmit می کنیم. یه اتفاق دیگه هم ممکنه توی

این state بیفته . این که بسته رو دریافت می کنیم، این که بسته مخدوش نشده باشه و ACK هم باشه (هر دو شرط با هم) در این صورت می فهمیم بدون خطا سمت گیرنده دریافت شده ، پس می ریم به state بعدی که منتظر دریافت داده از لایه ی اپلیکیشن باشیم.

- توی این state ، می خوایم به بسته ی متناظر با داده، شماره ی 1 رو اختصاص بدیم.(بقیه ی روند کار هم مشابه دوتا state بالاییه که 0 اختصاص می دادیم)

- FSM سمت گیرنده :



- مجددا نسبت rdt 2.0 تعداد state ها دو برابر شده.(متناظر با بسته هایی با شماره ی یک و بسته هایی با شماره ی صفر .
- State اولیه متناظر با بسته هایی با شماره ی صفره ، و منتظریم که از لایه ی پایینی بسته با شماره ی صفر دریافت کنیم. توی این state سه تا اتفاق ممکنه بیفته :

1 - بسته رو دریافت می کنیم ، بسته مخدوش نیست اما sequence number اش 1 عه به جای اینکه 0 باشه. به عبارتی این بسته بسته ی تکراریه. توی این حالت بسته رو دور می ریزیم و بعد یه بسته ی ACK می فرستیم که فرستنده متوجه بشه بسته ای که فرستاده تکراریه و فرستنده state اش عوض میشه.

2 - بسته رو دریافت می کنیم و بسته مخدوشه ، پس مشابه حالت قبل چیزی به لایه ی اپلیکیشن تحویل نمیدیم و میایم یه بسته ی NAK برای فرستنده ارسال می کنیم.

3 - بسته رو دریافت می کنیم ، بسته مخدوش نشده و sequence number اش هم 0 عه. این میشه همون چیزی که توی این state منتظرش بودیم. بعد میایم داده رو از بسته خارج می کنیم و تحویل لایه ی اپلیکیشن میدیم و یه بسته ی ACK هم می سازیم و به فرستنده ارسال می کنیم تا state فرستنده عوض بشه.

- دقیقا همین اتفاقات برای دریافت بسته ها با شماره ی یک هم میفته.

- پس rdt 2.1 پروتکلیه که می تونه مشکل خطا در کانال رو برطرف کنه.

جمع بندی :

- سمت فرستنده :

1 - از sequence number استفاده کردیم.

2 - شماره گذاری بسته ها به صورت باینری بود.

3 - سمت فرستنده منتظر دریافت ACK یا NAK یا بسته ی مخدوش هستیم و متناظر با هرکدوم کاری انجام میدیم.

4 - نسبت به rdt 2.0 ، state ها دو برابر بود.

- سمت گیرنده :

1 - با چک کردن شماره ی بسته ها از تکراری شدن بسته ها جلوگیری می کردیم.State ها هم متناظر با این بودن که بسته با شماره ی یک دریافت می کردیم یا صفر.

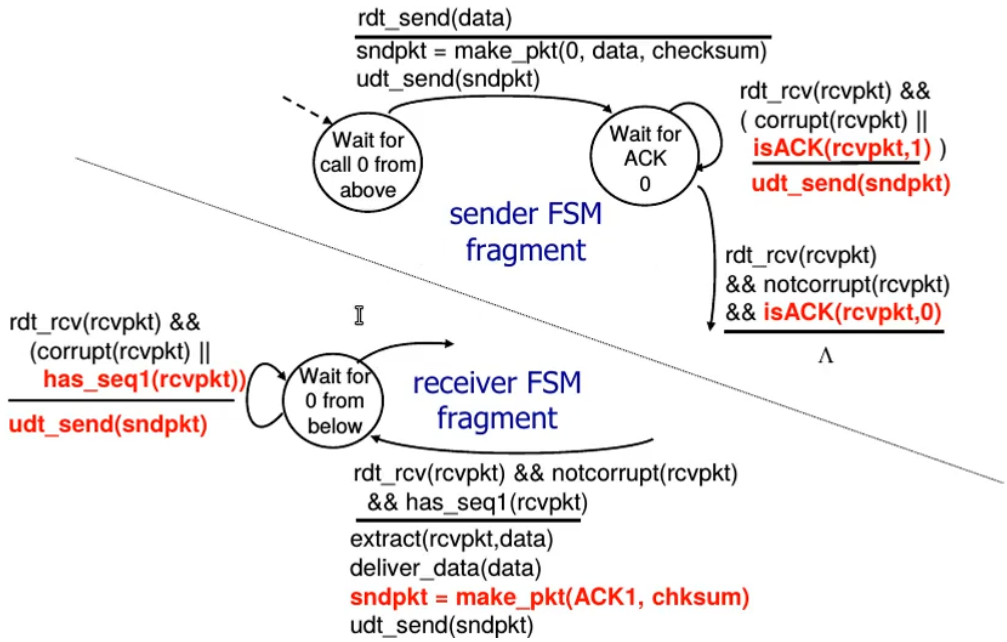
2 - گیرنده خبر نداره بسته ی ACK یا NAK ای که برای فرستنده فرستاده بدون مشکل به دستش رسیده یا نه.

- rdt 2.2 : a NAK-free protocol

- از لحاظ عملکرد شبیه rdt 2.1 عه با این تفاوت که فقط از ACK استفاده می کنه.
- کاری که NAK انجام می داد رو چطور توی این پروتکل انجام میدیم؟ ACK ها باید شماره ی بسته ها رو هم با خودشون ببرن ؛ به عبارتی گیرنده باید sequence number بسته هایی که داره متناظر باهاشون ACK می فرسته رو داخل بسته های ACK قرار بده.

بنابراین به جای ارسال NAK ، می تونیم ACK آخرین بسته ای که با موفقیت دریافت کرده رو بفرسته.به این کار میگن duplicate ACK . پس اگه سمت فرستنده یه ACK تکراری دریافت کردیم اونو به عنوان یه NAK در نظر می گیریم.

- TCP هم از نسخه ی NAK-free پروتکل rdt استفاده می کنه.
- FSM به روز شده ی فرستنده و گیرنده در rdt 2.2 (اگه فقط یک قسمت از دو قسمت متقارن رو در نظر بگیریم) به این شکله :



- اون قسمتایی که با rdt 2.1 تفاوت پیدا کردن با رنگ قرمز نمایش داده شده.
- توی قسمت فرستنده ، توی state ای که منتظر دریافت ACK بسته های صفر هستیم، اگه ACK ای دریافت کنیم که متناظر با بسته های یک هست ، اینو به عنوان یک NAK تلقی می کنیم و دوباره به همین state بر می گردیم و بسته ی قبلی با sequence number = 0 رو دوباره ارسال می کنیم.(duplicate ACK)

  وقتی هم ACK بسته ی صفر رو دریافت کنیم، دیگه ازین state می تونیم خارج بشیم.(متناظر با ACK عادی توی rdt 2.1)
- در سمت گیرنده باید مشخص کنیم که اگه یه ACK می فرستیم، مربوط به چه بسته ایه و اگه NAK می فرستیم، باید ACK متناظر با آخرین بسته ی موفقیت آمیز رو ارسال کنیم.

  وقتی توی state ای هستیم که منتظریم بسته ای با sequence number = 0 دریافت کنیم، اگه بسته مخدوش بود، یا بسته ایه که sequence number = 1 باشه ، باید بسته ی قبلی رو ارسال کنیم،(duplicate ACK)

  اگه بسته ای که میخواستیم رو دریافت کنیم، و مشکلی هم نداشته باشه، ACK بسته با شماره ی یک رو ارسال می کنیم. (state ای که نمایش داده نشده توی شکل، منتظر دریافت ACK بسته یک هستیم توش)

- rdt 3.0 : channels with errors and loss

- توی این کانال علاوه بر خطای بیت ها ، امکان گم شدن بسته ها هم وجود داره.
- گم شدن بسته ها هم می تونه شامل حال بسته هایی بشه که داده حمل می کنن، و هم بسته هایی در حال انتقال ACK هستن.
- به جز مکانیزم های checksum ، sequence number ، ACKs ، retransmission که تا الان استفاده کردیم باید از مکانیزم های دیگه ای هم استفاده بکنیم که با این پروتکل یه کانال امن بسازیم.
- توی همچین شرایطی چه کاری باید انجام بدیم؟
  توی همون مثال دیکته گفتن، اگه مادر به فرزند جمله ای رو گفت و هیچ پاسخی رو دریافت نکرد این احتمال رو میده که اصلا فرزند جمله رو نشنیده باشه، پس یه بار دیگه تکرار می کنه.
- توی این پروتکل ، ابتدا به مقدار معقولی (reasonable) منتظر دریافت عکس العمل (ACK) از گیرنده می مونیم، اگه چیزی دریافت نکردیم مجددا بسته رو ارسال می کنیم.
- حالا ممکنه این بسته ای که دریافت نکردیم، به خاطر تاخیر بوده باشه، نه به خاطر این که گیرنده بسته رو دریافت نکرده و ACK نفرستاده.
  توی این مواقع retransmission باعث ایجاد بسته های تکراری در گیرنده میشه که قبلا گفتیم مکانیزم sequence number این مشکل

رو حل می کرد و مثل قبل، گیرنده باید sequence number بسته های ACK رو با هم ارسال کنه.

- برای پیاده سازی rdt 3.0 ، چون نیاز داریم که بعد از ارسال هر بسته ، یه مقدار منتظر بمونیم، احتیاج به یه تایمر داریم که در بدو ارسال بسته اون تایمر رو هم باید راه اندازی کنیم.

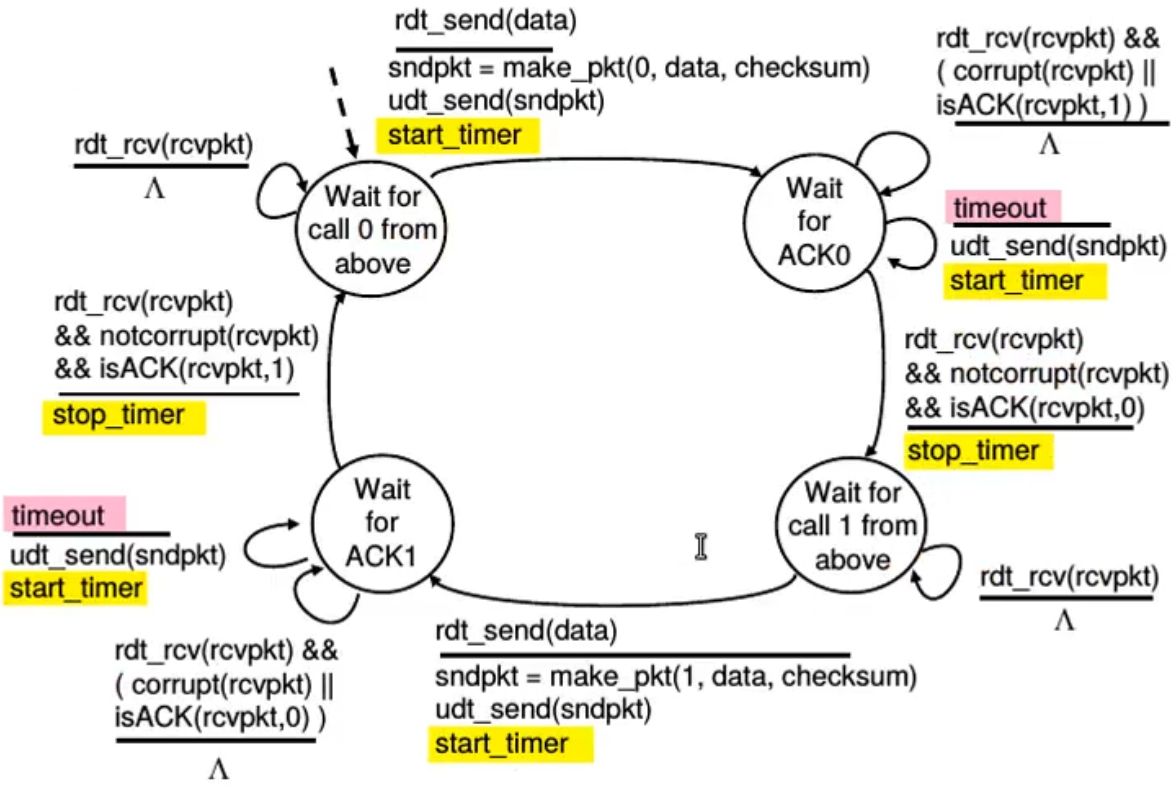
از اونجایی که این تایمر به صورت countdown عه و مقدار اولیه رو هی کم می کنه، اگه به صفر رسید و به عبارتی timeout شد ، مجدد بسته رو ارسال می کنیم. اگه قبل از timeout شدن، ACK دریافت کردیم، باید تایمر رو متوقف کنیم که retransmission انجام نشه.

- حالا اون مقدار اولیه ای که باید تایمر رو روش تنظیم کنیم چه مقداری باشه؟

معمولا اگه RTT رو بدونیم، و یه مدت زمانی هم برای پردازش بسته توسط گیرنده در نظر بگیریم، مقدار اولیه ی ایده آل توی تایمر ، processing time + RTT میشه.

ولی توی عمل هم مقدار RTT متغیره هم شرایط گیرنده ممکنه مشخص نباشه ، که بعدا راجع به مقداری که باید برای تایمر درنظر بگیریم صحبت می کنیم.

- شکل FSM مربوط به rdt 3.0 سمت فرستنده :



- خیلی از بخش ها مشابه rdt 2.2 هست با این تفاوت که بحث تایمر و timeout شدن تایمر و ارسال مجدد به دلیل timeout شدن، به FSM اضافه شده.
- مثلا توی state اول منتظر دریافت داده از لایه ی اپلیکیشن هستیم و به بسته ای که ازش ساخته میشه مقدار صفر رو تخصیص میدیم، اگه این اتفاق بیفته ، همزمان با اینکه بسته رو می سازیم و بعد ارسالش می کنیم، تایمر رو هم راه اندازی می کنیم. بعد به state بعدی میریم و منتظر دریافت ACK همین بسته می مونیم .
- توی state بعد که منتظر دریافت ACK بسته ی صفر هستیم ، سه تا اتفاق ممکنه رخ بده :

1 - بسته دریافت شده اما مخدوشه.یا اینکه ACK اش اون ACK ای که میخواستیم نیست، پس کار خاصی انجام نمیدیم و توی همین state باقی می مونیم.

2 - قبل از اینکه بسته ای دریافت کنیم، timeout رخ بده . در این صورت باید retransmission انجام بدیم و مجدد تایمر رو راه اندازی کنیم.

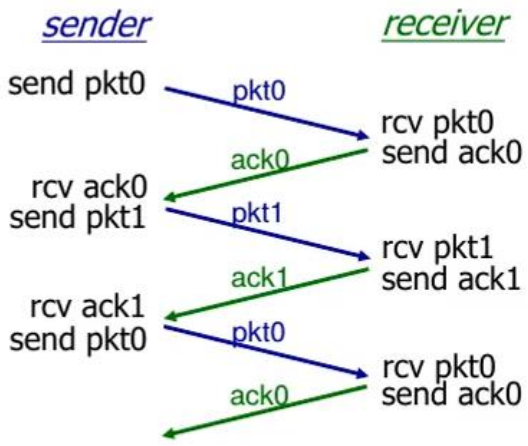
3 - بسته دریافت بشه و مخدوش نباشه و ACK اش هم صفر باشه، در این صورت تایمر رو متوقف می کنیم ، و به state بعدی میریم.

- توی state بعدی ، منتظر دریافت داده از لایه ی اپلیکیشن می مونیم. وقتی داده رو دریافت می کنیم از sequence number = 1 برای شماره گذاری بسته ها استفاده می کنیم .

ادامه ی روند کار مشابه وقتیه که sequence number برابر با 0 بود.

- مثال از rdt 3.0 :

- مثال اول : فرض می کنیم که کانال مشکلی ایجاد نمی کنه ، در این صورت فرستنده بسته ها رو یکی یکی ارسال می کنه و هر بسته رو که ارسال می کنه ACK

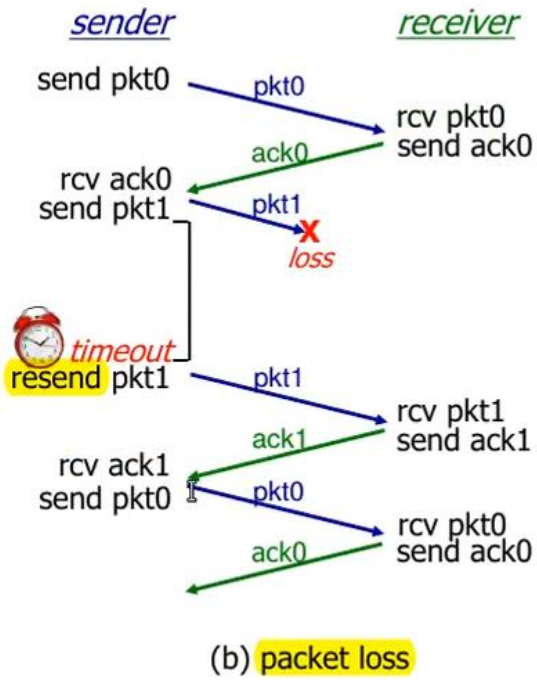


(a) no loss

اش رو دریافت می کنه و متعاقبا میره بسته ی بعدی رو ارسال می کنه.

- مثال دوم : فرض می کنیم کانال یک بسته رو گم می کنه . چون بسته گم شده طبیعتا سمت فرستنده هیچ ACK ای دریافت نمیشه و بعد از مدتی timeout رخ میده و بسته دوباره ارسال میشه .

بار دوم دیگه کانال بسته رو گم نمی کنه و بسته به دست گیرنده می رسه و گیرنده هم ACK رو ارسال می کنه و به state بعدی می ریم و منتظر دریافت بسته با sequence number = 0 می مونیم.(تایمر رو هم متوقف می کنیم که retransmission الکی رخ نده)

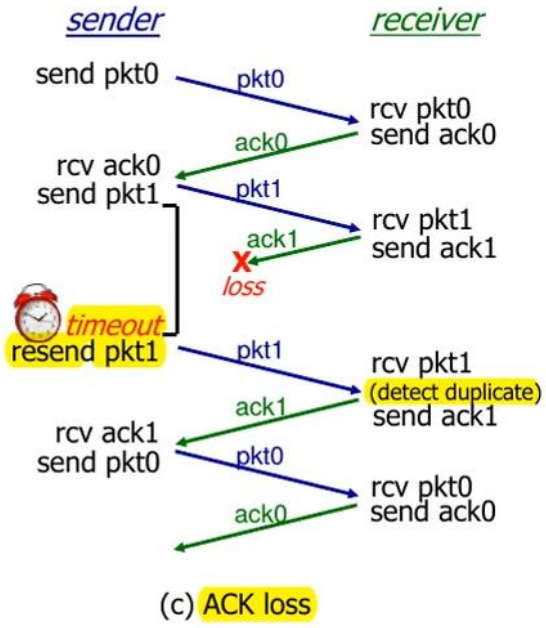


(b) packet loss

- مثال سوم : بسته ی ACK گم میشه.

- بسته ی صفر رو ارسال می کنیم بعد ACK رو دریافت می کنیم ، بعد بسته یک رو ارسال می کنیم ، گیرنده دریافتش می کنه ، ACK هم می فرسته اما گم میشه و به دست فرستنده نمی رسه. بعد یه مدت timeout رخ میده و بسته ی یک رو مجدد ارسال می کنیم . الان این

بسته یک ، duplicated عه ، چون قبلا گیرنده این بسته رو دریافت کرده بود.پس با توجه به sequence number تشخیص میده که بسته تکراریه و اونو دور می ریزه . بعد هم ACK 1 رو ارسال می کنه.فرستنده ACK 1 رو دریافت می کنه و میره به state بعدی و بسته ی صفر رو ارسال می کنه و .... ادامه ی کار مثل حالت عادی دنبال میشه.

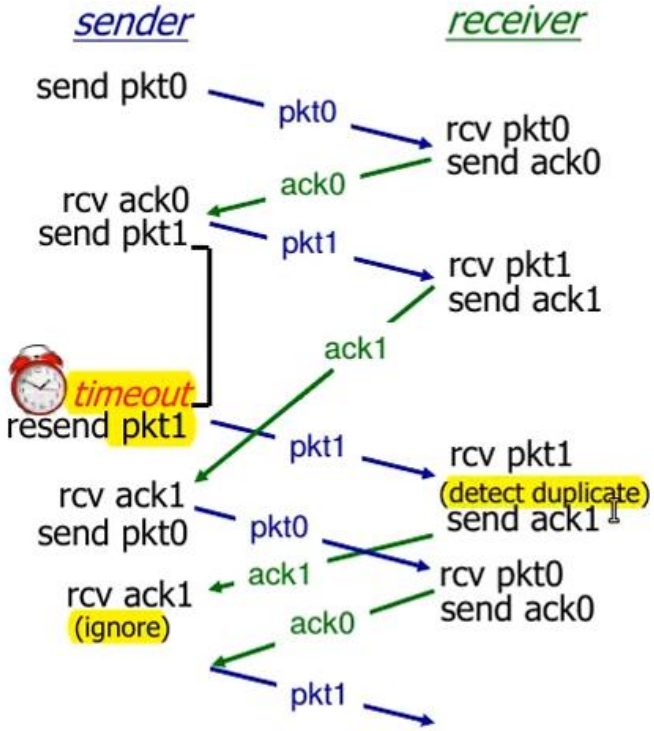


- مثال چهارم : مقدار اولیه ی تایمر مقدار کمی بوده و قبل از اینکه ACK رو دریافت کنیم timeout رخ میده.(بسته ای گم نشده) پس ACK ای که با delay همراه شده ، باعث میشه فرایند retransmission رخ بده.

- اول بسته ی صفر ارسال میشه ، ACK اش هم دریافت میشه ، بسته ی یک که ارسال میشه، مشکلی براش پیش نیومده و دریافت میشه و گیرنده هم ACK اش رو می فرسته، اما این ACK با تاخیر بیشتر از حالت معمول به فرستنده می رسه و قبل از اینکه برسه تایمر timeout شده و فرستنده دوباره بسته ی یک رو برای گیرنده می فرسته، بعد از

اینکه دوباره بسته فرستاد، تازه ACK بسته ی قبلی بهش می رسه. متناظر با ACK ای که با تاخیر رسیده، بسته ی صفر رو ارسال می کنیم، و بعد ACK بسته ای که retransmission شده بود به دست فرستنده می رسه! ولی فرستنده دیگه اینو ignore می کنه. و بعد ACK بسته ی صفر به دست فرستنده می رسه و از اینجا به بعد روال عادی رو داریم.

- نکته ای که هست اینه که به دلیل تاخیری که ایجاد میشه ، و بسته ی یک دوباره ارسال میشه، گیرنده دریافتش می کنه اما با توجه به sequence number اش بسته رو دور می ریزه و به لایه ی اپلیکیشن تحویل نمیده.



(d) premature timeout/ delayed ACK