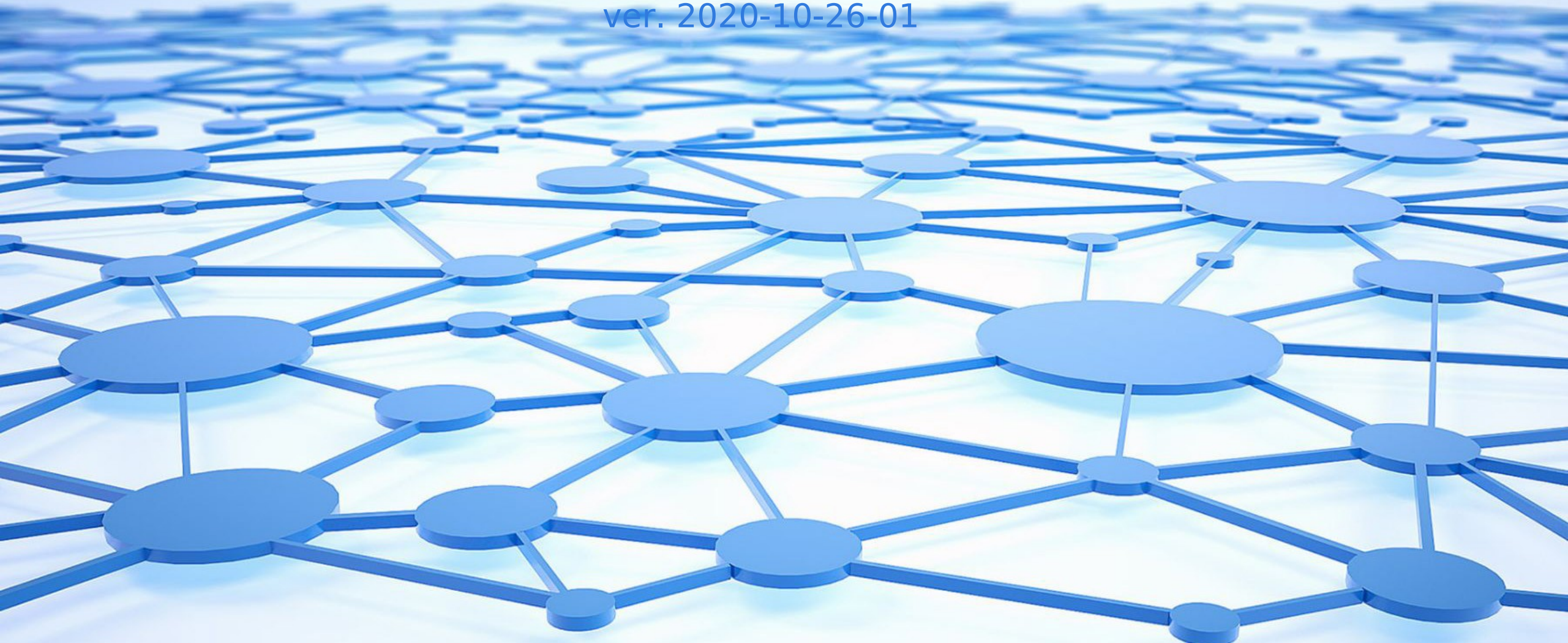
Úvod do počítačových sítí

Přednáška 6

(2021/2022)

ver. 2020-10-26-01



Řešení chyb v přenosech

- Příchozí rámec je kontrolován a kontrola selže, co dále ?
  - Kontrola pomocí parity, checksumy, crc …
- Na chybu je třeba reagovat primárně tím, že poškozený rámec nedám k dalšímu zpracování vyšší vrstvě
  - Proč bych to také dělal když už vím, že je spatně a tedy že výsledek správný nebude, ale může vzniknout více chyb
- Možností řešení je více a záleží na protokolu a použití
  - Nemusíme dělat nic
  - Můžeme zkusit chybu opravit (FEC - Forward Error Correction)
  - Můžeme se pokusit přenos opakovat (ARQ - Automatic Request Query)

Zahazování poškozených rámců

- Pokud zjistíme, že rámec není v pořádku nemusíme v některých případech „dělat nic“
- „Dělat nic“ v reálu znamená vynechat dvě akce
  - Předat data z rámce L3 vrstvě – NEDĚLÁM
  - Dávat odesilateli zprávu, že je něco spatně
- Pochopitelně nejjednodušší řešení, ale jde použít jen někde
- Typicky v multimediálních streamech
  - Výpadek jednoho či několika málo rámců nevadí
    - uživatel výpadek nezaznamená a nebo jen nepatrně – nepoškozuje to úplně funkčnost vysílání
    - Řešení by bylo „časově drahé“ a tedy viditelnější než když neudělím nic

Oprava poškozených rámů

- FEC - Forward Error Correction
- Oprava může být, po nic nedělání, nejrychlejší cesta jak se chybou v přenosu vypořádat
  - Nemusím posílat info odesilateli, že se něco nepovedlo a pak čekat na nová data
- Oprava dat není možné vždy / u všech typů přenosu
- Pro opravu potřebuji mít:
  - Omezenou množinu dat vhodně kombinovanou se zabezpečením
  - Přenášet jakákoliv data, ale vhodně upravená - „kódovaná“
    - Pozor zde se nejedná o kódování RZ, Manchester atd – to bylo na L1
    - Název je stejný, ale smysl je posílat dat tak, aby se v nich dala nalezená chyba upravit
  - Výskyt malého počtu chyb
    - Pokud je chyb hodně většinou není oprava vůbec možná
    - Teoreticky by možná být mohla, ale počet násobných zabezpečení by zabral většinu kapacity přenosového kanálu a to už pak nedává smysl

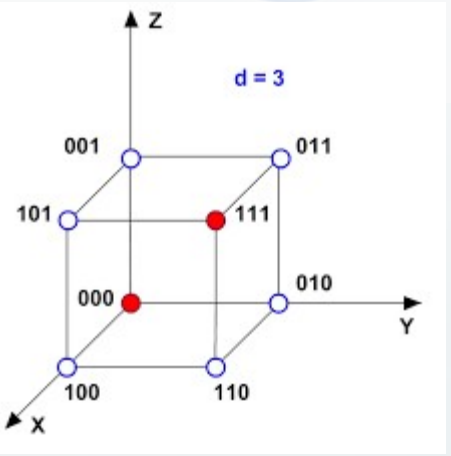
Hamingova vzdálenost

- Nemůžou se přenášet úplně libovolná data, ale jen vybraná
  - Např jen 010 a 101
    - Pro úplně libovolný přenos to nedává smysl, ale pro speciální - například řídící operace ano
  - Pokud bubu mít pevně danou množinu přenášených „slov“ mohu snáze určit chybu a zároveň ji mohu opravit
    - Samozřejmě za předpokladu, že se nejedná o násobné chyby
- Možnost najít v datech chybu a případně jí opravit je závislá na „rozdílnosti!“ dat
  - Pokud přenáším 1111 a 0000 tak 1110 je zjevně chyba „pravděpodobně“ se mělo jednat o 1111
  - Pokud přenáším 1111 a 1110 tak 1110 může být ok, ale také to může být chyba v prvním slově
- Z přenášených dat můžeme určit Hammingovu vzdálenost
  - Jedná se o nejmenší počet bitů, ve kterých se všechna možná přenášená slova liší
- Označuje se d_H

Hamingova vzdálenost: Metody určení

- „Ručně“ - odpočtením bitů
  - 00 a 01 => $d_H = 1$
  - Problém u delších slov, problém pro strojové zpracování
- Pomocí XOR pro každou dvojici
  - 000 a 001 => 000
001

0+0+1 = $d_H = 1$
- „Přenosem kódu na krychli“
  - Jednotlivé vrcholy definují jednotlivá slova
  - Vzdálenost vrcholů určuje vzdálenost dvou slov
    - Kolika hranami musím projít, abych se dostal z jednoho vrcholu na druhý



zdroj: http://myop.martinandco.eu/pdf/vypocetnisystemy.pdf

Hamingova vzdálenost příklad

- Příklad 1: Přenášíme data: 1111, 0000, 1010, určete Hammingovu vzdálenost
  - Nejprve určíme vzdálenost všech kombinací slov:
    - 1111 a 0000 => 4 (4 bity jsou různé)
    - 1111 a 1010 => 2 (2 bity jsou různé)
    - 0000 a 1010 => 2 (2 bity jsou různé)
  - Hammingova vzdálenost (tedy nejmenší nalezená vzdálenost) je 2, $d_H = 2$
    - Nejméně ve dvou bitech se všech slova liší
- Příklad 2: Určete Hammingovu vzdálenost pro kód, který přenáší libovolné kombinace bitů o délce znaku 2 (00,11,01,10)

00	00	00	11	11	01
11	01	10	01	10	10
----	----	---	---	---	---
1+1=2	0+1=1	1+0=1	1+0=1	0+1=1	1+1=2

  - $d_H(00,11,10,01) = 1$
    - (je to min z 2,1,1,1,2)

Hamingova vzdálenost a detekce a oprava chyb

- Tím, že Hammingova vzdálenost vypovídá o vlastnostech kódu, může vypovídat o jeho možnostech detekce a případně korekce chyb
  - Logicky, čím větší rozdíly v povolených slovech, tím snadnější poznání, že je něco špatně a „snad“ i jak by to mělo být dobře
    - „Snad“ proto, že při násobné chybě nemusí být oprava možná
- Detekce chyb
  - $d_H => n+1$ => $n <= d_H - 1$
  - Jsem schopen detekovat nejvíce chyb jako je rovno $d_H - 1$
- Oprava chyb
  - $d_H => 2n+1$ => $n <= (d_H - 1)/2$
  - Jsem schopen opravit maximálně tolik chyb, kolik odpovídá $(d_H - 1)/2$

Hamingova vzdálenost a detekce a oprava chyb: příklad

- Příklad 1: Kolik chyb jsem schopen detekovat, pokud přenáším jen slova 0001, 1111, 1010
  - Detekce chyb d_H=>n+1 => n<=d_H-1
  - D_H= 0000 0000 1111

    1100 1111 1010 1010

    ---------- ---------- ----------

    1+1+1+1=4 1+0+1+0=2 0+1+1+1=2 => d_H=2
    - n<=d_H-1, n<= 2-1; **n<=1** => Jsem schopen detekovat 1 chybu

- Příklad 2: Kolik chyb jsem schopen opravit, pokud přenáším jen slova 0001, 1100, 1001
- Oprava chyb
  - d_H=>2n+1 => n<= (d_H-1)/2
  - d_H = 2
  - n<= (d_H-1)/2; n<=(2-1)/2; **n<=0,5** => nejsem schopen opravit žádnou chybu
    - Logicky, protože pracuji na úrovni celých bitů a vyšlo mi „půl bitu“

Korekce chyb: Samoopravné kódy

- Vhodným způsobem upravím přenášená data, aby obsahovala násobnou možnost detekce chyb a tím i umožnila opravu
  - Upravím zde znamená „doplním o zabezpečovací bity“
  - Úprava jde samozřejmě na úkor využití kapacity přenosového kanálu
    - Logicky, čím více bitů na zabezpečení, tím méně prostoru pro bity datové
  - Typicky se jedná o násobné zabezpečení jednoho bitu
    - Nejjednodušším příkladem může být kombinace podélné a příčné parity, protože vím, že chyba je v konkrétním řádku a zároveň sloupci a tím pádem i vím jaka správní hodnota tam má být
    - Problém násobných chyb řeší jen částečně – násobné chyby se mohou „vymaskovat“
- Příkladem mohou být
  - Hammingovo kódování
  - BCH kódy
- V běžných přenosových systémech se masivně nepoužívají
  - Důvodem je složitost a „cena“ z pohledu zabrané kapacity

Korekce chyb: Hammingovo kódování

- Jedná se speciální příklad příklad lineárních dvojkových kodů (n,k)
- Doplňuje násobné zabezpečovací bity
- Kódy (3,1), (7,4), (15,11), (31,26), …
  - Často použitá varianta je (7,4)
    - $d_H = 3$
    - Dvě chyby detekuje
    - Jednu chybu opravuje
- 2^r=>r+k+1; n=r+k
- n – délka slova r- paritní bity k – informační bity
- Princip tvorby
  - Paritní bit jsou na pozicích druhých mocnin (1, 2, 4, 8, 16, …..)
  - Informační bity jsou na ostatních pozicích (3, 5, 6, 7, 9, 10, 11, …)
  - Paritní bit se vypočítá z **některých** bitů informačního slova. Pozice informačních bitů, které se vynechávají udává pozice paritního bitu
  - $P_1\ P_2\ I_1\ P_3\ I_2\ I_3\ I_4$
  - $P_1 = I_1 + I_2 + I_4$ (ve zbytku slova 1 bit přeskočím, jeden zkontroluji, jeden přeskočím, jeden zkontroluji …)
  - $P_2 = I_1 + I_3 + I_4$ (první bit přeskočím, dva zkontroluji, dva přeskočím, dva zkontroluji ….)
  - $P_3 = I_2 + I_3 + I_4$ (tři přeskočím, čtyři zkontroluji, čtyři přeskočím, čtyři zkontroluji, ….)
- Tvořím s – syndrom, který slouží ke kontrole po přenosu a který musí být nulový
  - $S_1 = P_1 + I1 + I2 + I4 = 0$
  - $S_2 = P_2 + I1 + I3 + I4 = 0$
  - $S_3 = P_3 + I2 + I3 + I4 = 0$

Korekce chyb: Další kódování

- Rozšířené Hammingovo kódování
  - Rozšiřuje základní Hammingův kód o přidání paritního sloupce
  - Např z (7,4) vznikne (8,4)
    - 4 informační a 4 zabezpečovací
    - $d_H = 4$
      - Tři chyby detekuje
      - Jedno chybu opraví
- BCH kódy
  - Využívají CRC
  - Používají se např při satelitních přenosech
    - Přidávám sice hodně informace navíc, ale zpoždění v přenosu (RTT) je tak veliké, že se to vyplatí

Korekce chyb: Opakování přenosu

- ARQ - Automatic Request Query
- Pokud detekuji chybu a
  - Nejsem schopen ji opravit (Hammingovy kódy, BCH, …)
  - Data potřebuji celá, protože se nejedná např o multimediální přenos

  musím řešit opakovaný přenos
- Pro opakování přenosu je třeba dát odesilateli info, že je třeba data poslat znovu
- Zavedeme potvrzování a timeout
- Potvrzování
  - Potvrzení má typicky formu informačního rámce
    - Jsou i výjimky, kdy se potvrzení přibalují k datům, v případě duplexních kontinuálních přenosu při vysoké míře aktivity v komunikaci
  - Kladné - potvrzuji správně přijetí každého rámce (ACK)
  - Záporné - posílám info v případě, že jsem detekoval chybu (NACK)
  - Kladné i záporné - kombinace obou případů
- Timeout
  - Nejen přenos, ale i doručení potvrzení může selhat
  - Řeší problém nekonečného čekání
    - Čekám X dlouho a následně přenos prohlásím za neúspěšný a mohu jej zopakovat

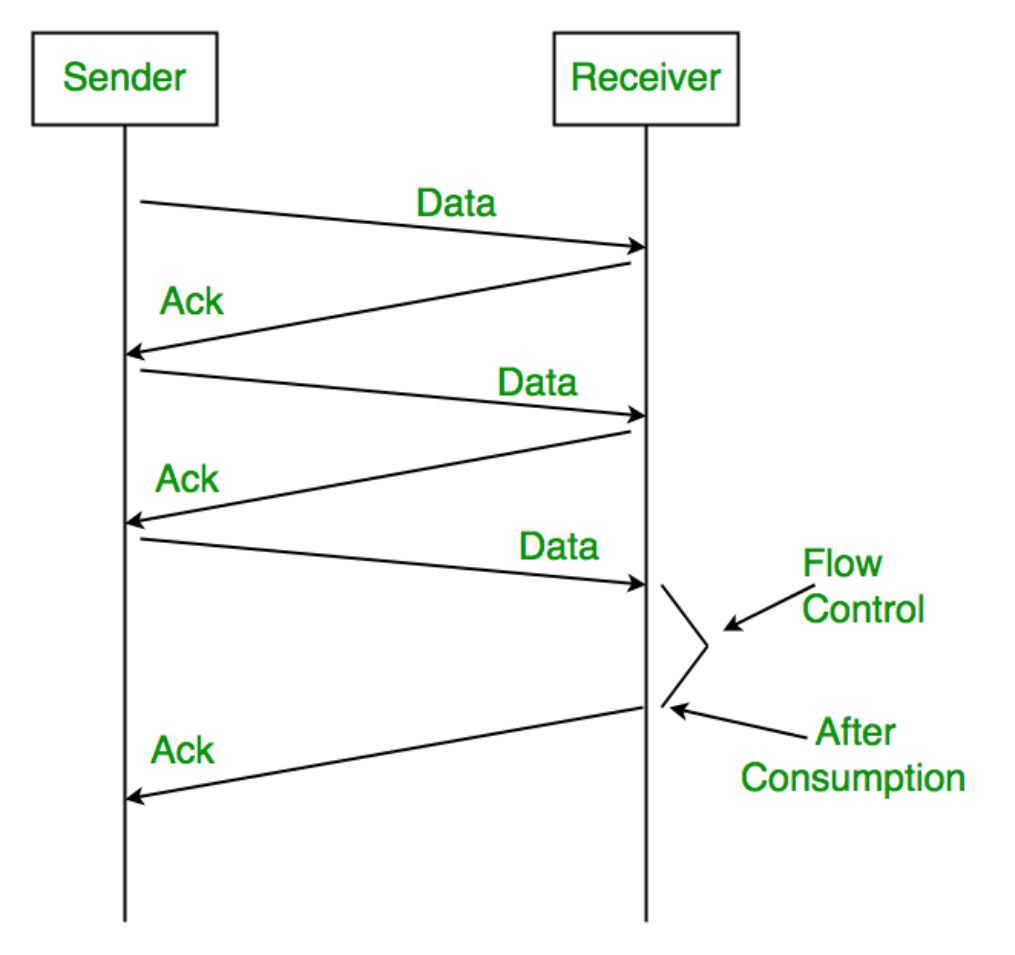
Simplexní protokol bez omezení

- Simplexní pro jednoduchost
  - Data(myšleno co chci posílat) posílám jen jedním směrem
- Bez omezení – posílám data včetně zabezpečení, ale nestarám se o to jak to dopadne
  - Nepoužívám potvrzování
  - Nepoužívám samoopravné kódy
- V běžných provozech na L2 jen omezené využití
  - Pro představu se může jednat o „alternativu“ UDP(L4), ale na L2
- Není možné použít pro všechny typy přenosů, ale jen
  - Tam kde výpadek části dat nevadí
  - O konzistenci dat se starají až vyšší vrstvy
    - Ale to obecně nechceme, protože je to časově náročnější

Simplexní protokol Stop & Wait

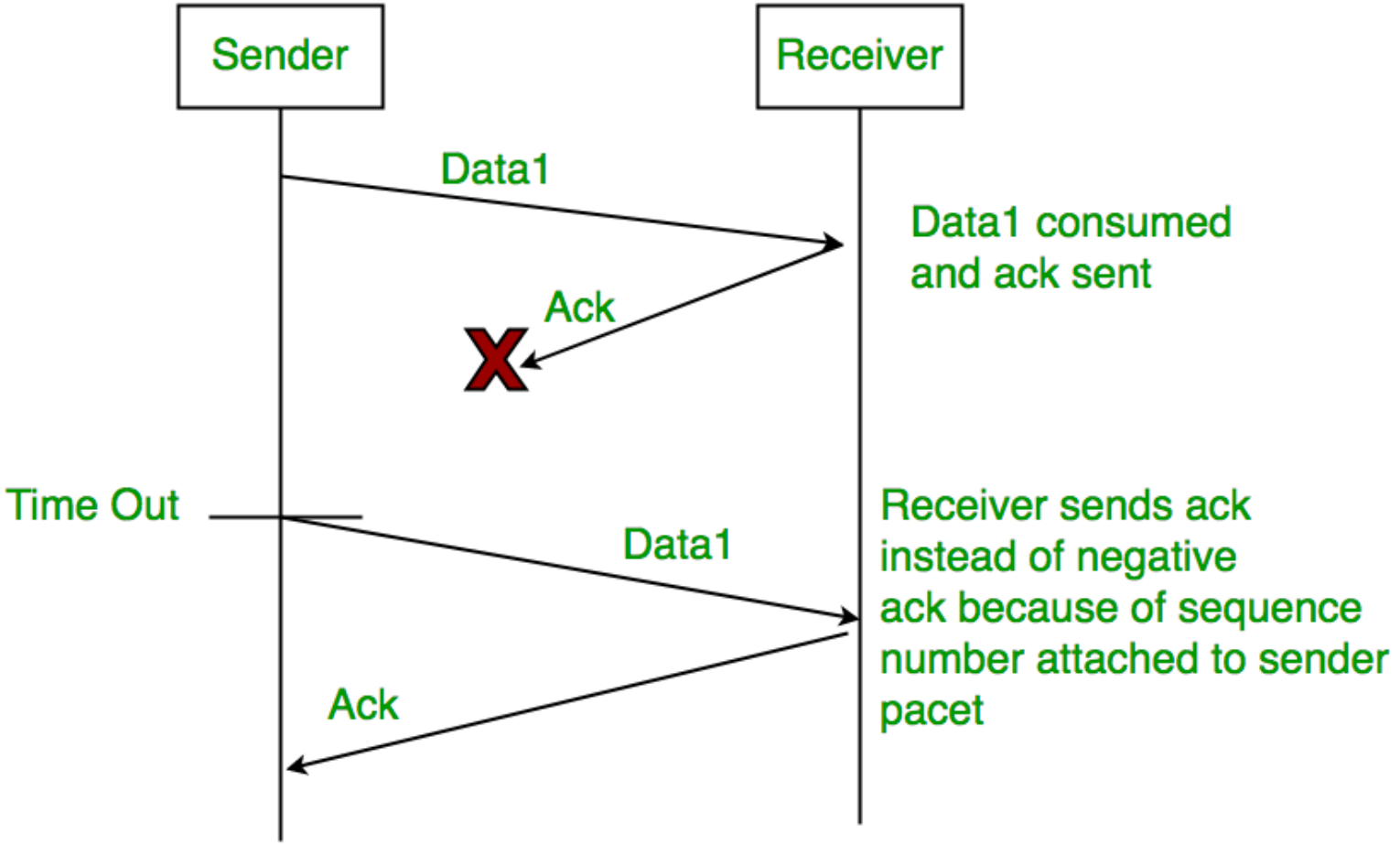
- Využívá kladné potvrzování jednotlivých rámců
- Postup přenosu
  - Odesílatel odešle data VČETNĚ zabezpečení
    - To je nutné, abych věděl jak mám rozhodnout o správnosti / chybovosti přenosu
  - Příjemce přijme data a zkontroluje zabezpečení
    - Pokud je zabezpečení v pořádku, posílá se kladné potvrzení – ACK
    - Pokud zabezpečení není v pořádku neudělá se nic
      - Prostě data zahodíme a čekáme až přijdou znovu a správné
  - Odesílatel čeká na příjem kladného potvrzení
    - Pokud jej přijme a je v pořádku začne posílat další data
    - Pokud žádné potvrzení nedorazí do určitého času – timeoutu – přenos zopakuje
    - Pokud potvrzení nedorazilo v pořádku – bylo poškozené např – odešlou se data znovu a opět se čeká na potvrzení
      - Ale to je vlastně problém … řekneme si dále

Simplexní protokol Stop & Wait: příklad bezchybného přenosu



zdroj: https://www.geeksforgeeks.org/stop-and-wait-arq/

Simplexní protokol Stop & Wait: příklad chybného přenosu

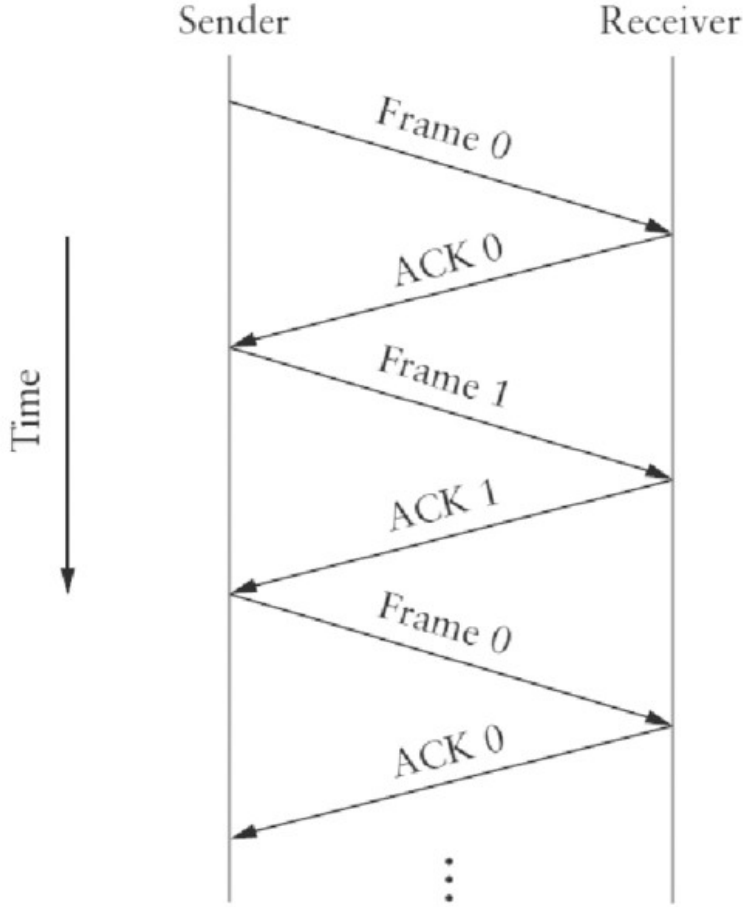


zdroj: https://www.geeksforgeeks.org/stop-and-wait-arq/

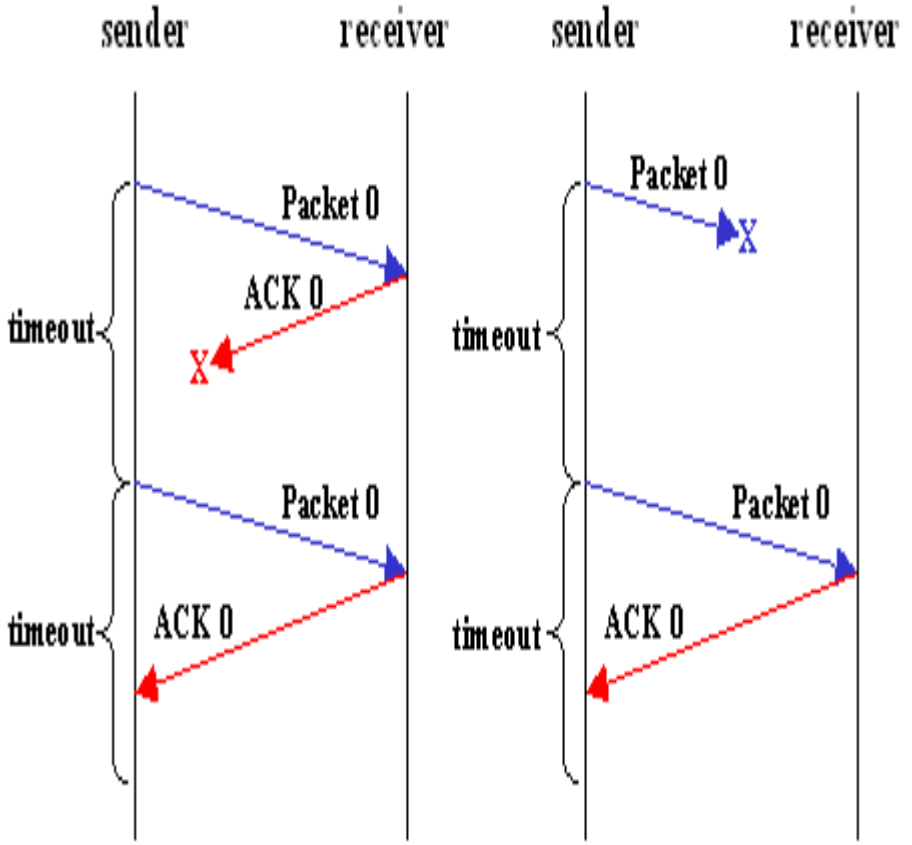
Simplexní protokol Stop & Wait: číslování rámců

- U Stop & Wait protokolu může nastat problém v případě, že
  - ACK nepřijde správně
    - Je to informační rámec a ten při přenosu poškodil
  - ACK bylo odesláno, ale nedorazí
    - Data vůbec nedorazí
- V obou případech odesílatel NEDOSTAL ACK v pořádku a bude přenos opakovat a odešle podruhé stejná data
  - To je ok a to od něj i čekáme
- Příjemce data obdrží, ale neobdrží další data jak očekával, ale dostane znovu stejná data
  - Na straně příjemce nastává duplicita, kterou příjemce není schopen odhalit
    - Zas by to musela řešit vyšší vrstva – opět časově drahé
- Vyřešíme zavedením číslování rámců a potvrzení
  - Pro Frame0 přijde potvrzení ACK0, pro Frame1 bude ACK1
  - U simplexního Stop & Wait protokolu nám stačí rozlišit dva rámce Frame0 a Frame1
    - Tedy zda už posílám nový a nebo znova předchozí rámec

Simplexní protokol Stop & Wait: číslování rámců - příklad



zdroj: https://www.researchgate.net/figure/5-Timeline-for-Stop-and-Wait-Protocol-with-1-bit-Sequence-Number-31_fig18_236259629



zdroj: https://www.isi.edu/nsnam/DIRECTED_RESEARCH/DR_HYUNAH/D-Research/stop-n-wait.html

Prokotol Ask and Go

- Zatím jsme vždy pracovali s předpokladem, že příjemce je stále připraven přijímat data
- To ale nemusí být pravda a může být zdrojem problémů
  - Například proto,že aktuálně příjemce nemá ke zpracování dostatek paměti / dostává dat moc
- To řeší protokol Ask and Go a nebo také někdy Task and GO
- Princip je jednoduchý, napřed se zeptám zda příjemce data může přijmout a až řekne, že ano, začínám přenos
- Evidentně budu posílat nějaká data navíc
  - Dotaz a potvrzení dotazu
- Ale data „navíc“ budou výrazně menší než požadovaná data a tedy snižuji nutnost opakování přenosu
  - Což může být velice výhodné při pomalém přenosu a dlouhých blocích dat
  - Z důvodu úspory energie například v bezdrátových sítích

Prokotol Ask and Go příklad

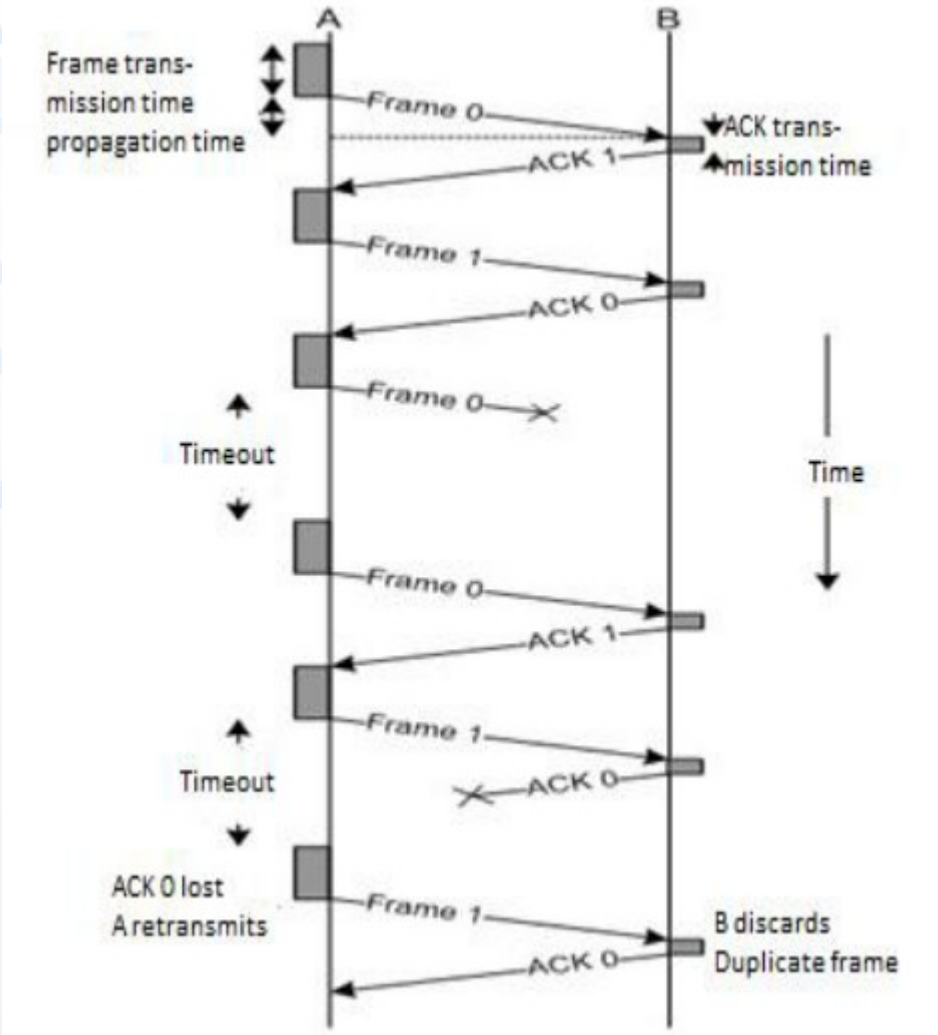
Vysílač		Přijímač
ptám, se zda mohu vysílat	**ASK1 →**	
ptám, se zda mohu vysílat	**← ACK1**	Odpovídám, že ano a očekávám data
Odesílám data	**DATA1 →**	
	← ACK1	Data jsou v pořádku a posílám kladné hodnocení

Data jsou přijata v pořádku

mohu pokračovat s dalším rámcem

Jednotlivé a kontinuální potvrzování

- Protokol Stop & Wait používá jednotlivé potvrzování
  - Každý jeden rámec je odeslán, pak je čekáno na jeho potvrzení a AŽ po kladném potvrzení se pokračuje dále
  - Jednoduché na implementaci, na zdroje – málo paměti
  - Používán byl například v IPX/SPX společnosti Novell
  - Jedná se o polo-duplexní komunikaci
  - Vede k velice špatnému využití přenosového kanálu, zvlášť při vyšším RTT
    - $n=t_{TRANS}/(t_{PROP}+t_{TRANS}+t_{PROP}+t_{ACK})$ [%]
      - Pro LAN (ethernet) kde je latence linky 25.6 µs je efektivita ~ 92%
        - » Což je ok
      - Pro WAN kde je latence linky např 50ms je efektivita ~ 2.3%
        - » Což je je velice špatně
- Řešením je „kontinuální“ potvrzování
  - Nečekám po každém odeslání na potvrzení, ale posílám ihned další data
  - Potvrzení přichází se zpožděním

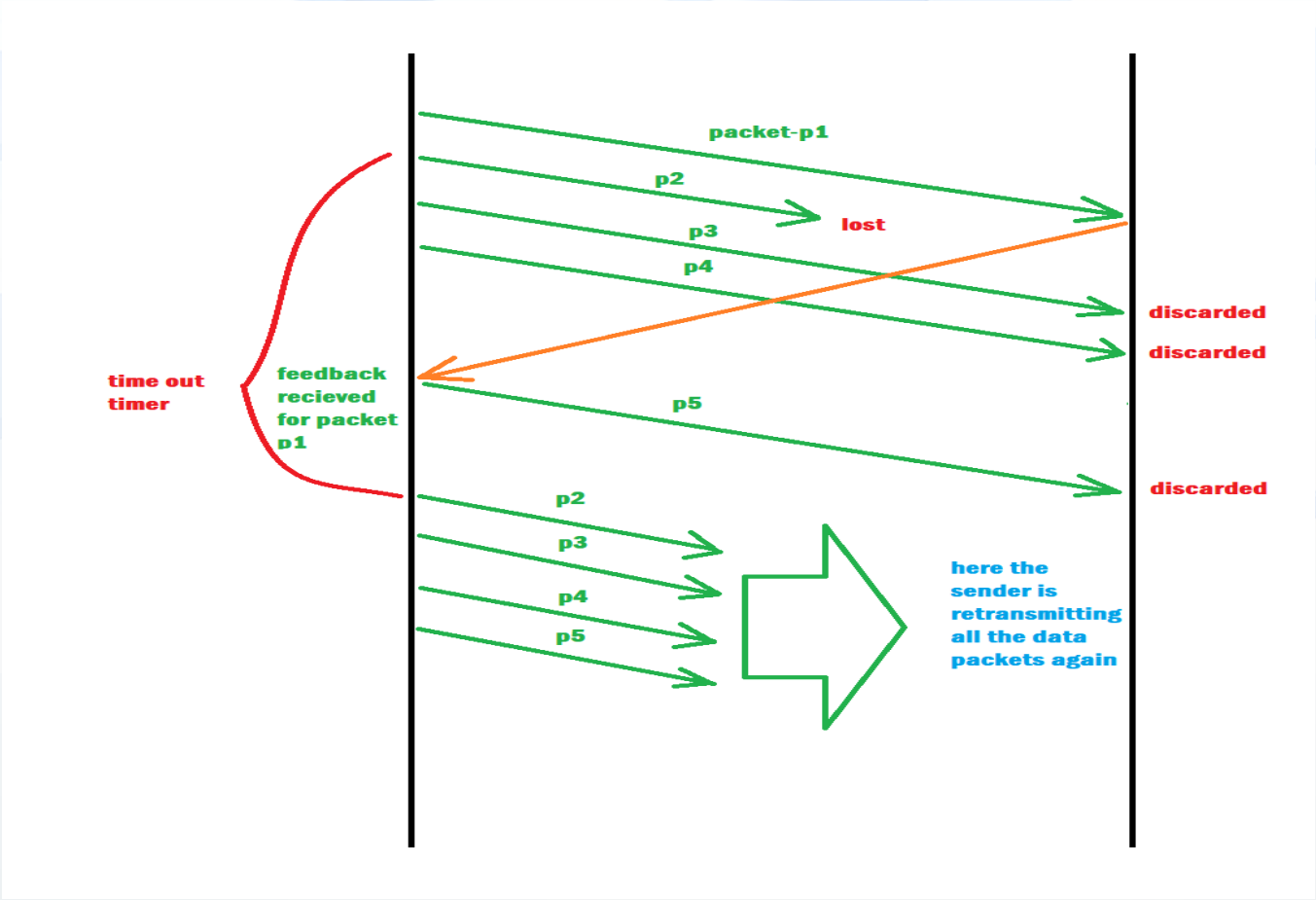


zdroj: https://www.researchgate.net/figure/Link-utilization-and-working-process-of-stop-and-wait-ARQ_fig1_275043653

Kontinuální potvrzování: Go Back N

- Nečekání na přijetí ACK1 pro Frame1, ale posílá data ihned po odeslání Frame1
- Lepší využití linky, protože nedochází k čekání bez přenosu
- Tím, že data i potvrzení mohou chodit na „přeskáčku“ umožňuje duplexní přenos
- Vyžaduje číslování rámců a potvrzení
  - Rámce mohou být posílány v různém pořadí – v případě opakovaní přenosu – a tedy je potřeba je identifikovat
- Princip fungování
  - Pokud se přijmou data ve správném pořadí a mají v pořádku zabezpečení, pošle se kladné potvrzení a data jsou postoupena vyšší vrstvě na další zpracování
  - Pokud se přijmou data se správným zabezpečením, ale MIMO pořadí (čekám Frame2, ale přišel Frame3), tato a následující data jsou zahozena
  - Jelikož příjemce neobdržel Frame2 a tedy neposlal ACK2, odesilatel pošle Frame2 po vypršení timeoutu znovu a zároveň i všechny následující rámce
- Velmi často se v komunikaci nevyskytuje jediná chyba osamoceně, ale bývá poškozeno více rámců zároveň – na což Go Back N reaguje
- Špatné využití přenosového kanálu, protože při chybě 1 rámce se jich odesílá znovu N, přestože jejich přenos už mohl být úspěšný

Kontinuální potvrzování: Go Back N – příklad

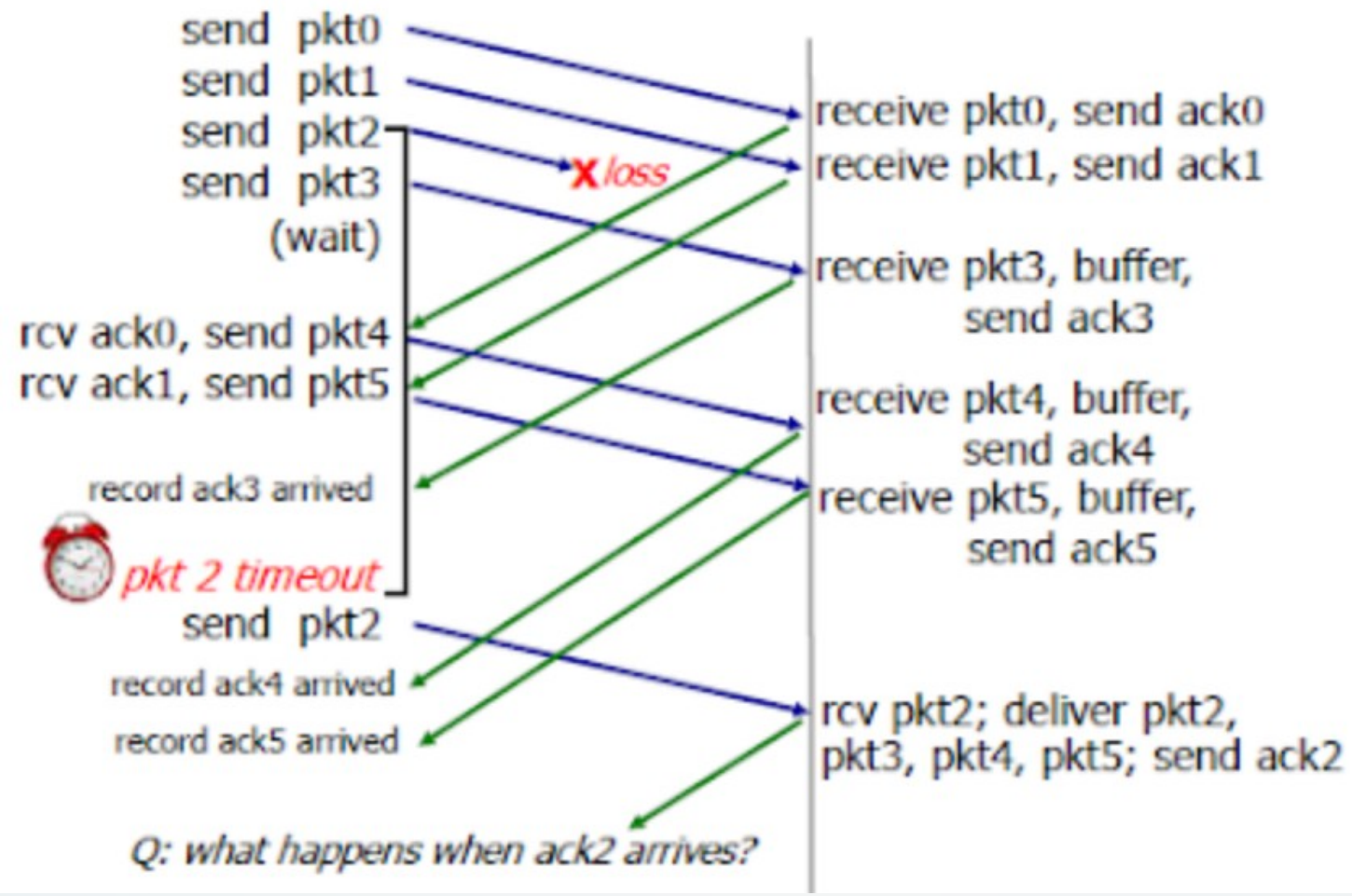


zdroj: https://www.studytonight.com/post/flow-control-gobackn-arq-protocol

Kontinuální potvrzování: Selective repeat

- Jak bylo uvedeno Go Back N lépe využívá kapacitu přenosového kanálu než Stop&Wait, ale v případě chyby přenáší veškerá data od chyby znovu – což může být neefektivní
  - Ale nemusí, záleží na charakteru linky, tedy jaký typ chyb převažuje, zda samostatné a nebo sdružené
- Řešením je kontinuální selektivní kladné potvrzování, které stejně jako Go Bank N nečeká na potvrzení pro práce odeslaný rámec, ale posílá ihned další data
- Na rozdíl od Go Back N, ale při chybě posílá znovu JEN chybný rámec
  - Nedochází k duplicitním přenosům a tedy se lépe využívá kapacita přenosového kanálu
  - Na straně příjemce musí existovat buffery, kde se uchovávají rámce došlé mimo pořadí, protože vyšší vrstvě L3 se musí odesílat ve správním pořadí

Kontinuální potvrzování: Selective repeat - příklad

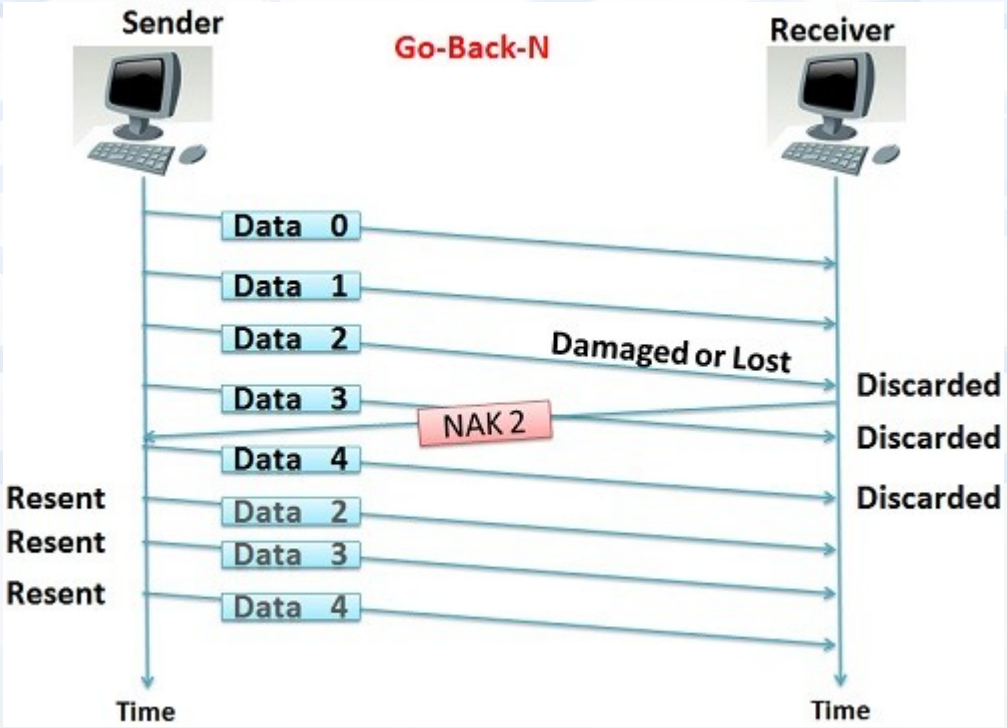


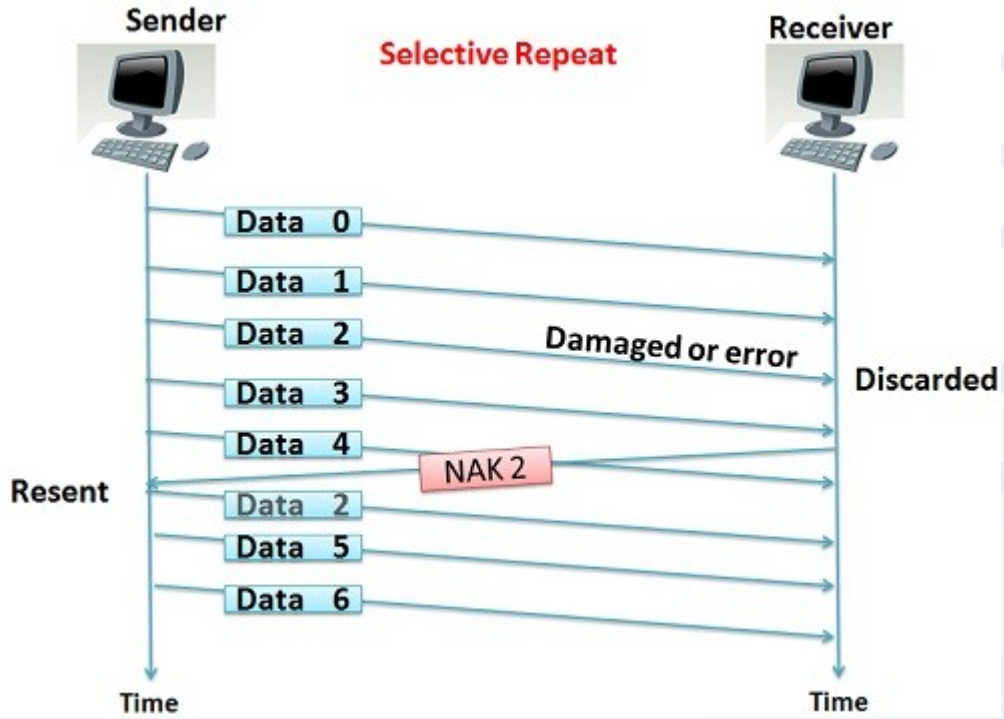
zdroj: https://www.geeksforgeeks.org/sliding-window-protocol-set-3-selective-repeat/

Kladné a záporné potvrzování

- Jak jsme v předchozích případech uváděli, tak ACK – kladné potvrzení slouží k potvrzení správného přijetí dat
- Pokud kladné potvrzení nepřijde včas – tedy do uplynutí timeoutu – je přenos považován za chybný a dle protokolu je opakován
  - Někde jen jeden rámec(Stop&Wait) někde i více rámců(Go Back N)
- Pokud data nedorazila správně, neděje se při kladném potvrzení nic a čeká se na timeout
- To ale může být problém, protože timeout musí být vyšší než je doba nutné k odeslání dat a přijetí potvrzení
  - Jako příjemce musím dlouho čekat než odesílateli dojde, že nastala chyba o které já už vím
- Řešením je kombinace kladného a záporného potvrzování, protože pak se i informace o chybě přenáší ihned jak se existence chyby zjistí
- Typicky se záporné potvrzování označuje jako NACK nebo někdy NAK

Kladné a záporné potvrzování - příklad





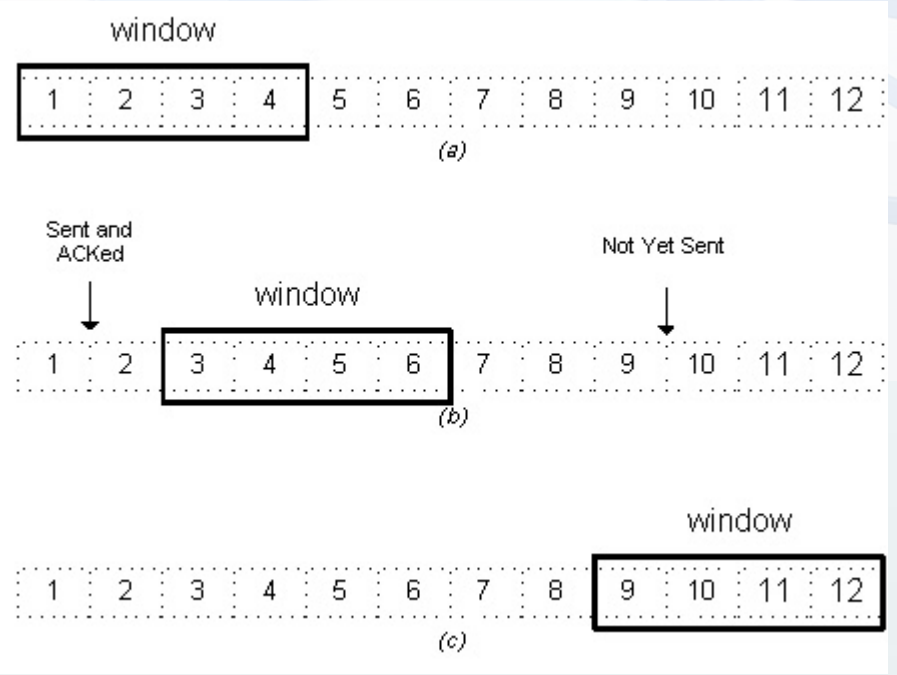
zdroj: https://techdifferences.com/difference-between-go-back-n-and-selective-repeat-protocol.html

Protokoly s klouzajícím okénkem

- Při použití kontinuálního potvrzování může odesílatel využít maximálně kapacitu přenosového kanálu – což ale ne vždy musí být žádoucí
  - Příjemce nemusí stačit data zpracovávat
    - Z důvodu výkonu
    - Z důvodu nedostatečné paměti pro uchování rámců ve frontě na zpracování
- Pokud příjemce data nestačí zpracovávat nemá (podle toho co zatím víme) jinou možnost než nadbytečná data zahodit
  - Například proto, že je nemá kam uložit
- Do hry opět vstupuje timeout
  - Kterého už jsme se snažili kombinací kladného a záporného potvrzování zbavit a který nám komunikaci výrazně zpomaluje
- Základní myšlenka je, že poslat data znovu je často nákladnější, než je poslat jednou i když pomaleji než je kapacita linky, ale tak, že je příjemce stačí zpracovat

Protokoly s klouzajícím okénkem II.

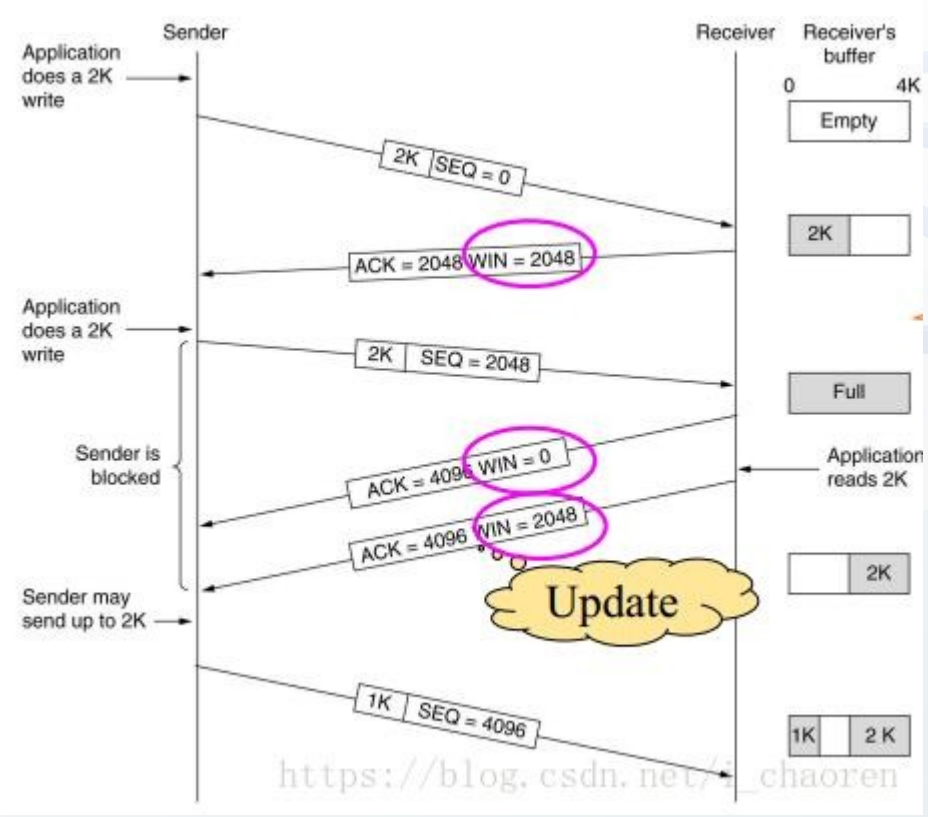
- Nebudeme odesílat libovolné množství dat – počet rámců – ale jen „pevně“ definovaný počet
- Tento počet – blok – nám definuje „okénko“ tedy „kolik dat najednou vidíme“
- Odesílám rámce ihned za sebou jen do velikosti okénka
  - Tím, že nám limituje pohled na data tak jich více „nevidíme“
- Po přijetí ACK1 posuneme okénko o 1 pozici vpravo
- Rychlost posunu okénka se mění podle příchod ACK
- Pokud nepřijde ACK1 nemůžeme okénko posunout, protože může dojít k opakovanému přenosu
- Příjemce nemusí mít neomezeně buffery, protože dle velikosti okénka ví kolik dat MAX najednou může dostat



zdroj: https://www.thecrazyprogrammer.com/2017/05/sliding-window-protocol-program-c.html

Protokoly s klouzajícím okénkem: řízení rychlosti II.

- Příjemce se spolupodílí na stanovení velikosti
  - Odesílatel jej nemusí respektovat / může navrhnout vlastní hodnotu
- Je možné realizovat jako samostatné rámce a nebo jako součást datových rámců
- Řízení rychlosti se může řešit na více vrstvách
  - Například L4 - TCP
- Řízení rychlosti řeší dva problémy
  - Zahlcení příjemce
    - Kapacita sítě je dostatečné, ale příjemce nestíhá
      - Odesílatel a příjemce mohou být HW výrazně rozdílní
      - Příjemce může být využíván více klienty
  - Zahlcení sítě
    - Cesta mezi odesílatelem a příjemce je saturovaná nebo chybuje
    - Data nedorazí vůbec nebo dorazí chybně



zdroj: https://www.programmersought.com/article/58372052298/

Protokoly s klouzajícím okénkem: řízení rychlosti III – reakce na zahlcení

- Zahlcení příjemce
  - Pokud je úplné nemůže příjemce dělat nic a data zahodí
    - Odesílatel nedostane ACK a ví, že musí snížit rychlost
  - Pokud má příjemce ještě prostor reagovat, pošle WIN s nižší hodnotou než je aktuální
    - Tím dokáže rychlost snížit
    - Může poslat i WIN0 a tím dočasně přenos pozastavit
    - K obnovení přenosu dojde na základě zprávy od příjemce s WIN > 0
- Zahlcení sítě
  - Příjemce odeslal NACK
    - Data dorazila, ale jsou chybná
    - Odesílatel pošle data znovu A ZÁROVEŇ sníží WIN, protože očekává problém v síti
    - Toto se může opakovat
  - Odesílateli nedorazilo ani ACK ani NACK nebo došla poškozená
    - Data se ztrácejí nebo přenos chybuje
    - Stejně jako v předchozím případě odesílatel reaguje snížením rychlosti, protože očekává že při stejné rychlosti se bude s větší pravděpodobností chyba opakovat