# Transportlaget

## Transportlaget: services of protokoller

Transportlaget står for:

END2END transport: Forbindelse mellem processer.

Den står også for at opdele data i mindre portioner(segmenter). Segmentering og re-assembly

Skal fremsende data til korrekt proces. Multiplexing og demultiplexing.

Fejl-korrektion.

**TCP (transmission control protocol)**

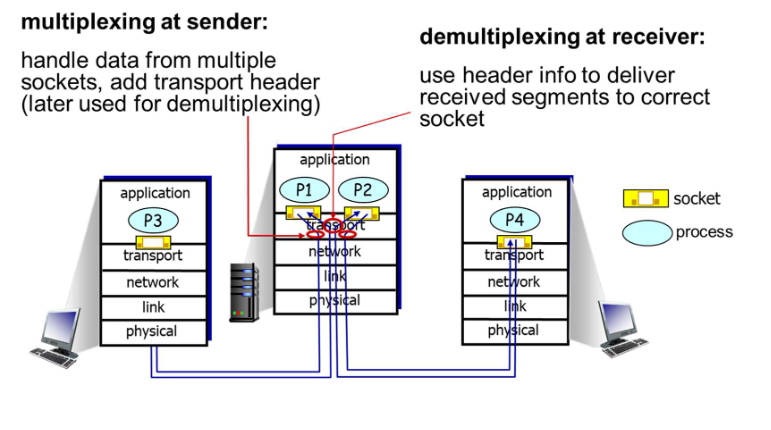
TCP er forbindelses-orienteret og det er en pålidelig byte-stream service. Bruges når pakkerne SKAL frem uden corruption og med den rigtige rækkefølge. Den er to-vejs.

**UDP (user datagram protocol)**

UDP er forbindelsesløs og er derfor ikke en pålidelig byte-stream service. Pakkerne bliver sendt afsted uden at tjekke om de bliver modtaget. Kan derfor tabe pakker.

UDP har ingen ventetid på forbindelsen. Den har også en meget simpel og lille header. Altså lavt overhead.  
Headeren indeholder dog en length, som er størrelsen af payload i bytes, inkl. Headeren. Der er også en checksum, der bruges til at verificere dataen når den kommer frem.

### Multiplexing og demultiplexing



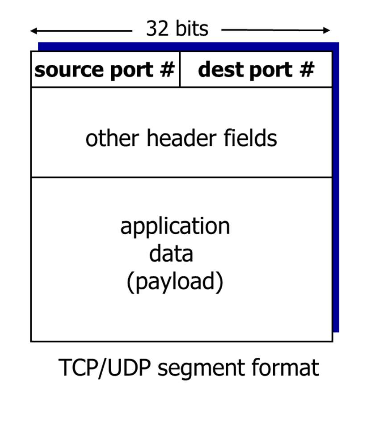
Processer adresseres/navngives af deres værts ip-nummer + port.   
Bindeledet mellem transportlaget og applikationslaget er en **Socket.** Så en proces kan have en eller flere sockets, så den kan kommunikere med en eller flere andre processer.

**Multiplexing:**

Senderen skal kunne håndtere data fra flere forskellige sockets.   
Når transportlaget modtager data, laver den en header med en IP og port, med hvor dataen kom fra og hvor det skal hen.

**Demultiplexing:**

Den kigger på headeren og ser hvor at dataen skal afleveres, altså hvilken socket.

🡨 transportlags segment. Hver datagram har en source IP og destination IP. Desuden har det et transportlags segment. Transportlags segmentet har en source og destination port.

### De-multiplexing UDP.

De-multiplexing for UDP virker meget intuitivt. Der er kun dest og source port nummer. Den tjekker først om der findes en aktiv socket på dest-porten. Hvis der gør, videresendes dataen.   
I UDP kan en socket godt have 2 sendere.

### De-multiplexing TCP.

De-multiplexing for TCP har source IP og port og dest IP og port. Den bruger så alle 4 værdier til at sende data til den rette socket.   
I TCP kan der være mange sockets, en til hver forbindelse. Det er vist nedenunder.

## 

## 

## Principper for pålidelig data-kommunikation

For at sende data fra en proces til en anden, skal applikations-laget bruge transport laget, som så bruger netværkslaget til at sende det videre.

### Tilstandsmaskiner

Tilstandsmaskiner bruges til at lave en graf over handlinger og tilstande. Det følgende er en kaffemaskine:

#### 

#### Transmission protocol

Sender-tilstandsmaskinen(transport laget)(STM) får data fra applikationslaget og laver en transportlags pakke, med dataen som payload og sætte en header på. Derefter sender den det underlæggende transmissionsmedie til at sende det over netværket.

Modtager-tilstandsmaskien(transport laget)(MTS) afventer data fra det underlæggende lag. Når den får data, pakker den dataen ud og afleverer det der hvor headeren siger det skal hen.

Men der kan være corruption på transportmediet. For at klare det problem bruges ACK(acknowledgement) og NAK(Negative acknowledgement). Hvis pakken er modtaget korrekt, sendes ACK tilbage og hvis ikke en NAK. Så sendes dataen igen. Måden dataen checkes på er ved en Checksum.

Dog kan ACK og NAK også corruptes. For at fixe det, kan man sætte et sekvensnummer på pakken, så ACK og NAK kan sende det med tilbage. Derved kan STM vide hvilken pakke den får svar på. ACK kan corruptes mens sekvensnummeret bliver. Så sender STM dataen to gange. MTS ignorerer bare dataen anden gang og sender et nyt ACK tilbage.

Man kan også gøre det med kun ACK. Så hvis dataen corruptes, sendes ACK tilbage, men med sekvensnummer fra pakken før. Det er altså det seneste data der er korrekt modtaget.

Hvad gør man hvis man TABER pakkerne og de ikke kun corruptes? Man kan tilføje en timer på STM, som venter på at få et ACK. Hvis den ikke får et ACK efter ”rimelig tid” sendes dataen igen.   
Det er dog et problem at bestemme denne ”rimelige tid”.

For lang: Unødvendig langsom.  
For kort: Unødvendig retransmission af data og ACK.

Hvad man gør i praksis, er at holde øje med round trip tiden(RTT) og lave timeout tiden ud fra den.   
Den metode vi er kommet frem til hedder ”Den alternerende bit-protokol”. Den er håbløs langsom. Typen hedder Stop&Wait da den afventer kvittering før den afsender ny data.

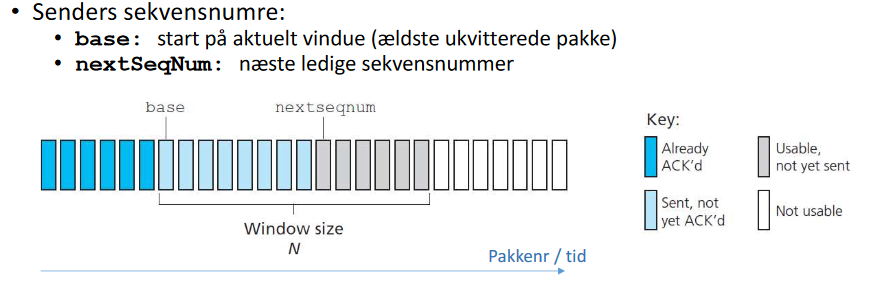
## Pipelinede protokoller

Ideen I pipelinede protokoller er at sende flere pakker afsted ad gangen. Skal dog stadig ikke oversvømme modtageren eller overbelaste netværket.

Pakkerne bliver gemt i bufferen på senderen og modtageren indtil de er 100% leveret.

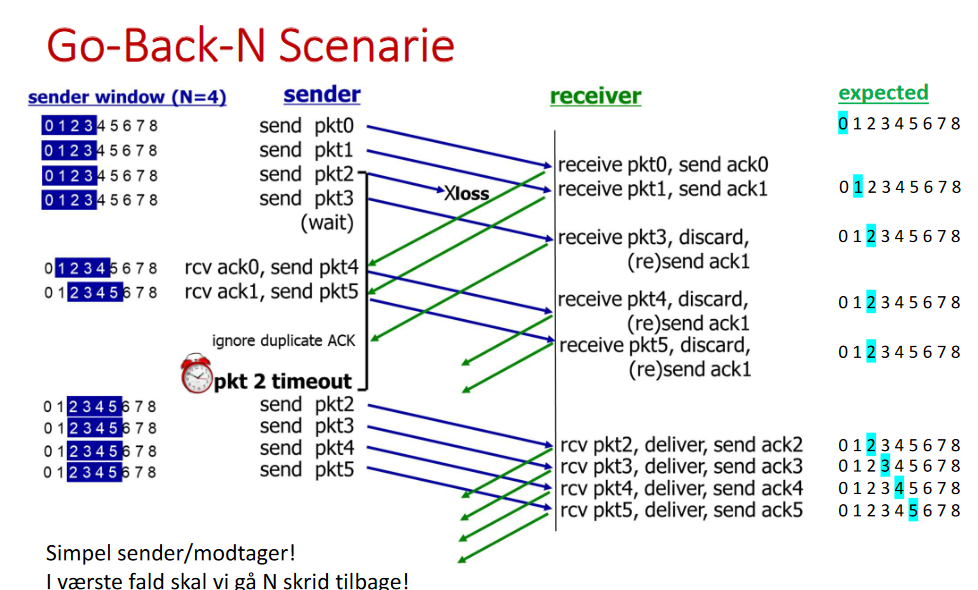
### Go-back-N

Go-back-N er en strategi, hvor vi har højst N antal ukvitterede pakker undervejs i vores pipeline(window size). Der er stadig et sekvensnummer i headeren. Sekvensnummeret har K bits i størrelse. Derfor er der [0,1…,2k-1] tal. Når det sidste tal er nået, wrapper den rundt og starter forfra.



Når senderen modtager ACK(N) betyder det at alle pakker til og med N er korrekt modtaget.   
Der sættes en timder for ældste ukvitterede pakke. Hvis der er timeout gensendes alle pakkerne i vinduet der er sendt efter N.   
Hver gang der modtaget et ACK glider vinduet en tand frem.

Modtageren checker om der er huller i nummerrækken den modtager pakkerne i. Sender kun ACK hvis det er i rigtig rækkefølge. Hvis der kommer pakker out of order, sendes ACK for sidste korrekt modtaget pakke.

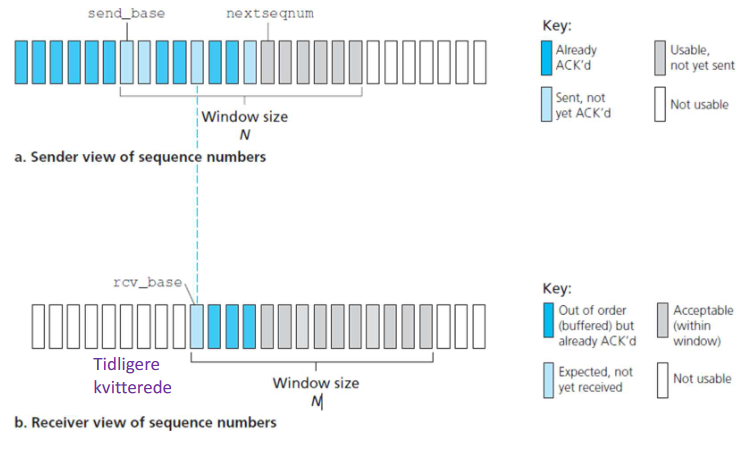


I ovenstående eksempel, går pakke 2 tabt.

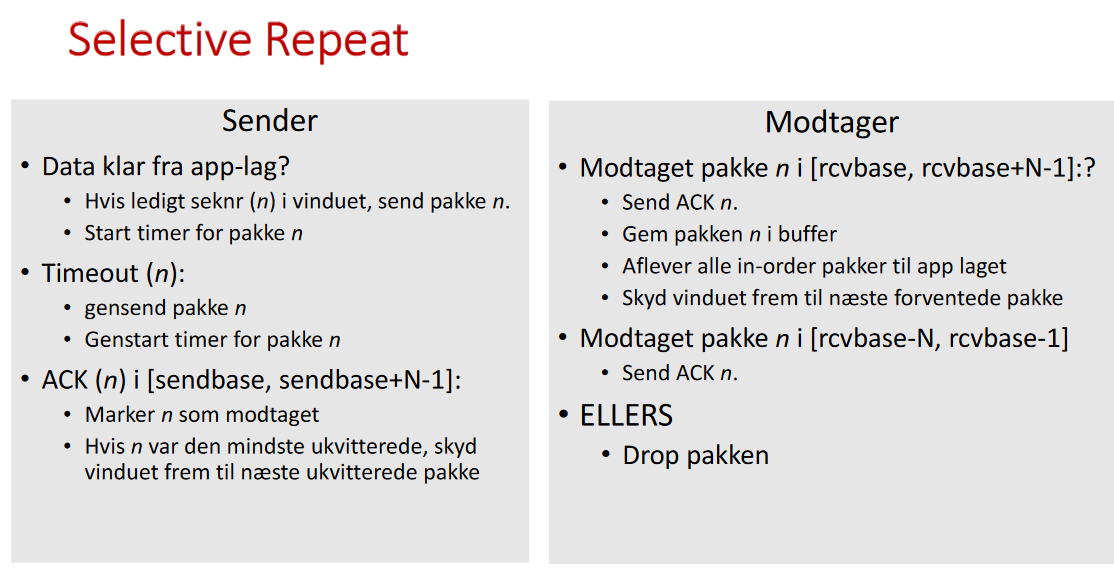
## Selective repeat

Har stadig max N ukvitterede pakker.

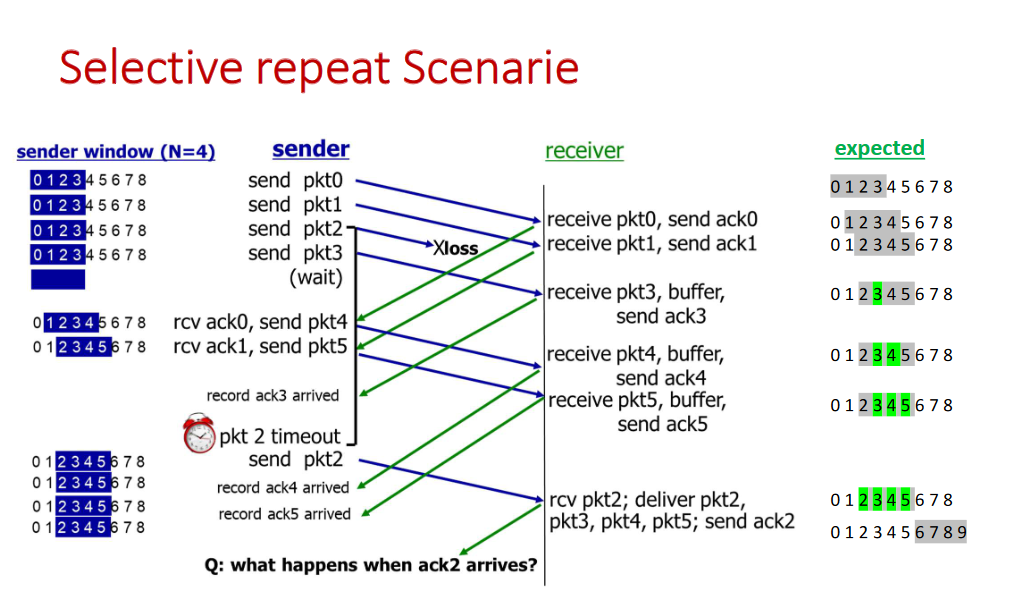
I Go-Back-N blev pakker out of order bortkastet, men I selective repeat bliver de gemt i bufferen. Den kvitterer så individuelt for de modtagne pakker så senderen kun skal gensende dem der mangler kvittering. Senderen sætter også en timer for hvert eneste pakke i pipelinen.



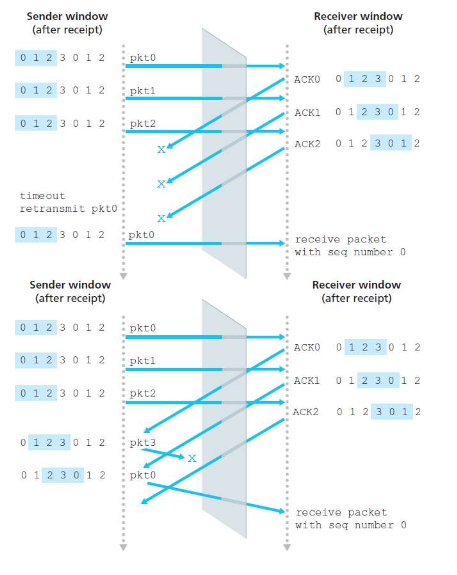
Recieveren accepterer nun pakker inden for Window size.



Billedet ovenover forklarer det godt.



Pakke 2 er igen tabt. Husk at der tændes en timer for eneste pakke. Der sendes altid et ACK tilbage med sekvensnummer, når en pakke er modtaget.

I nederste eksempel godtager modtageren en forkert pakke med sekvensnummer 0. Her bliver dataen leveret igen, som om at det var ny data.

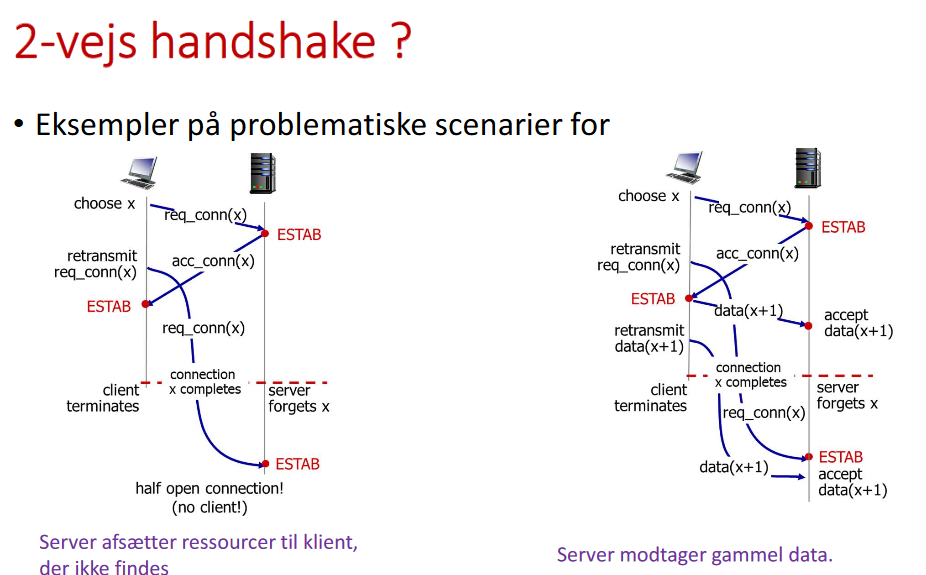
Man kan ikke se forskel på ny og gammel data, når man genbruger sekvensnumrene.

Man vil gerne have 2N <k2-1 antal sekvensnumre. Altså det dobbelte antal.

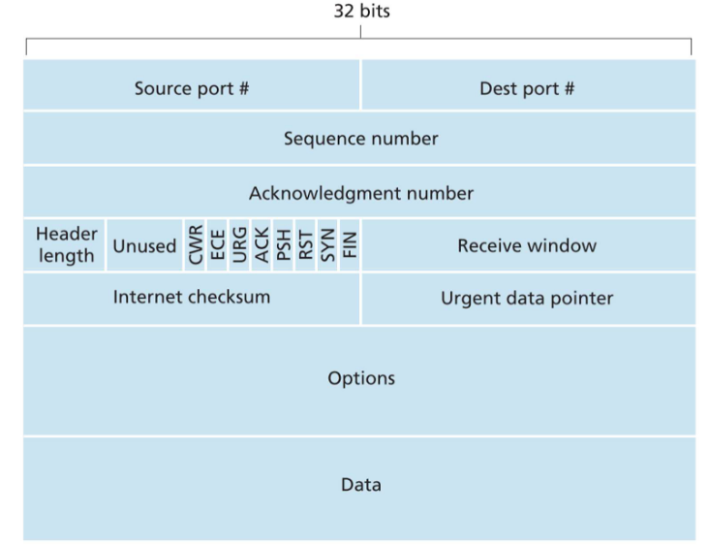
Dog kan en pakke komme meget ”for sent” og have et sekvensnummer som er blevet wrappet og brugt igen.   
En løsning er, at man antager at pakker ikke lever mere end 3min.   
Man kan også sætte et tidsstempel på, men alle computeres ure er ikke synkroniserede.

## Etablering og nedlukning af TCP-forbindelser

For at blive enige om at oprette en forbindelse skal der afsættes buffer-plads, bestemmes sekvensnummer og vidues størrelse. Dette skal også gøres i begge retninger.

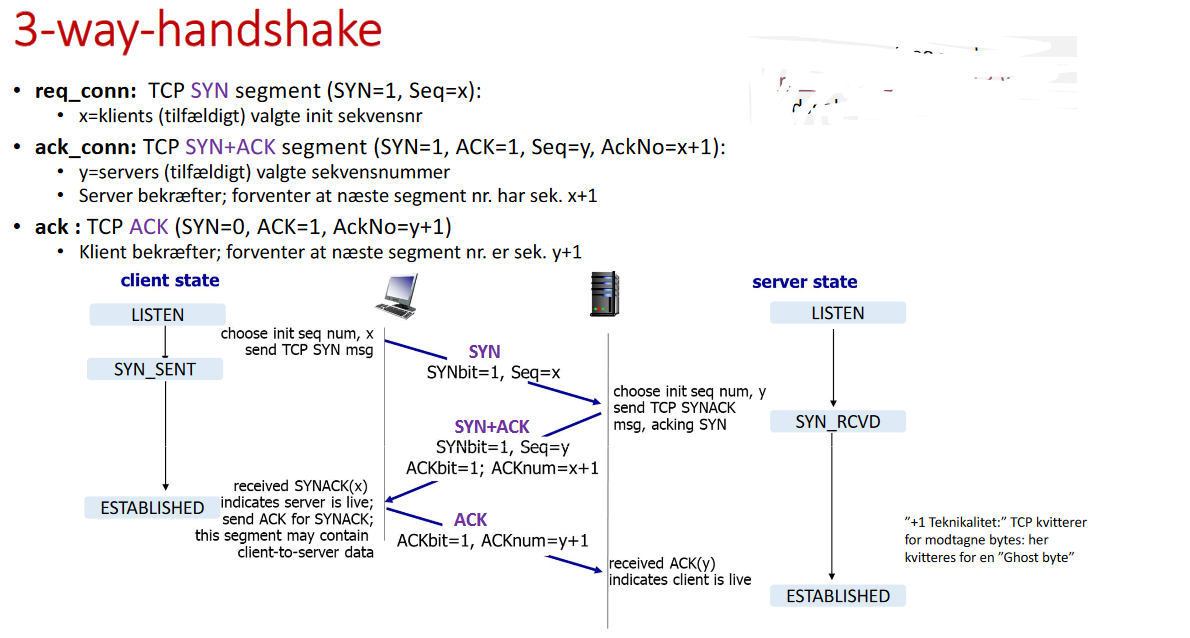
Gamle pakker må ikke medgå i ny forbindelse. Ved nedlukning skal alt data først leveres. Pakkerne med oprettelse og nedlukning kan også gå tabt. 

For at fixe problemerne på billedet, bruger TCP en 3-vejs handshake. Klienten og serveren laver et gensidigt ACK. Altså der kommer et ACK begge veje.

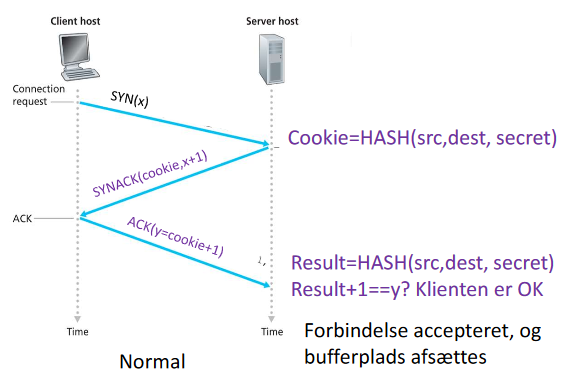
🡨 TCP segment  
  
Kontrolbits:  
SYN = SYNchronize

FIN = FINish

ACK = ACKnowledge. Hvis ACK = 1 er der gyldig data i acknowledgment feltet.



Forbindelsen kan bruges til at angribe servere. En klient kan spoofe og konstruere falske SYN pakker. Når serveren modtagere en SYN pakke, bliver der oprettet en buffer. Hvis nok gør det, bliver serveren overvældet.   
SYN Cookies forsvar:

****  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
**Kontrolleret nedlægning**Klienten sender en FIN segment og serveren sender et ACK hvor de derefter lukker. Afsenderen giver tid til at sende det sidste ACK. Kan også gøres den anden vej.   
Dog er det teoretisk umuligt at sikre 100% enighed om lukning (two-army problem).