**Chapter 15: Transmission Control Protocol**

Table of Contents

[15.1 TCP Services 1](#_Toc86834119)

[15.2 TCP Features 2](#_Toc86834120)

[Numbering System 2](#_Toc86834121)

[15.3 Segment 3](#_Toc86834122)

[15.4 A TCP Connection 6](#_Toc86834123)

[Connection Establishment 6](#_Toc86834124)

[Connection Termination 8](#_Toc86834125)

[SYN Flooding Attacks 10](#_Toc86834126)

[15.5 State Transition Diagram 11](#_Toc86834127)

[15.6 Windows in TCP 14](#_Toc86834128)

[15.7 Flow Control 16](#_Toc86834129)

[Silly Window Syndrome 17](#_Toc86834130)

[15.8 Error Control 19](#_Toc86834131)

[Fast Retransmission 20](#_Toc86834132)

[Deadlocks 21](#_Toc86834133)

[15.9 Congestion Control 22](#_Toc86834134)

[Exponential Increase 22](#_Toc86834135)

[Additive Increase 23](#_Toc86834136)

[Multiplicative Decrease 24](#_Toc86834137)

[15.10 TCP Timers 26](#_Toc86834138)

[Retransmission 26](#_Toc86834139)

[Persistence Timer 28](#_Toc86834140)

[Keepalive 29](#_Toc86834141)

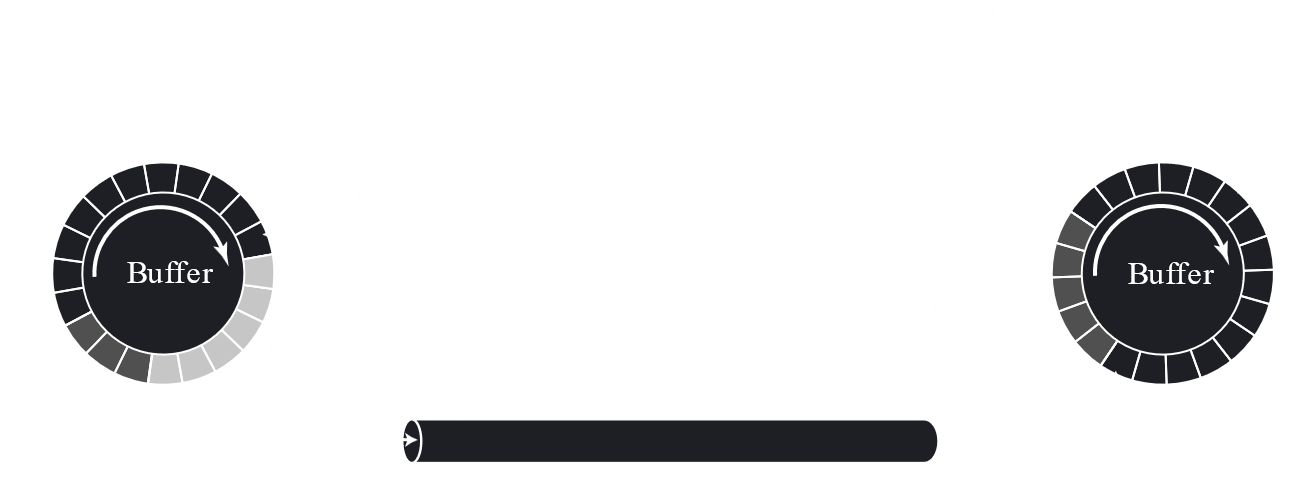
[Time-Wait 29](#_Toc86834142)

## 15.1 TCP Services

TCP provides several services, some of which have been discussed earlier.

* Process-to-Process Communication
* Stream Delivery Service
* Full-Duplex Communication
* Multiplexing and Demultiplexing
* Connection-Oriented Service
* Reliable Service

The Stream Delivery Service merits some extra discussion. TCP can transport a **stream of bytes**. Even though the IP protocol that transports the data breaks up the data into smaller packets, the TCP layer can maintain the stream so that no data is **out of order**. This makes use of a **buffer** on the sender’s side and another on the receiver’s side, which maintains the order using a **sliding window** mechanism.



TCP creates **segments** of data, with some header, which it sends to the IP layer.

## 15.2 TCP Features

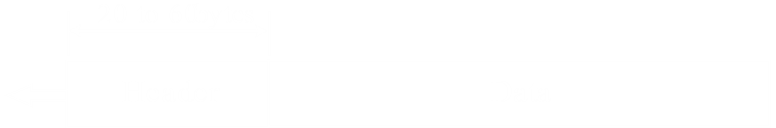
### Numbering System

TCP is called a **byte-oriented** protocol. This it because it **numbers every byte** of data. The number for bytes starts at an **arbitrarily generated** number, not at .

Note that, even though the bytes are the things being numbered, a **segment** does not consist of just one byte. It can have **multiple bytes**. Each segment is assigned a **sequence number**, which is the number of the **first byte** in the segment. The segments themselves do not have their own number.

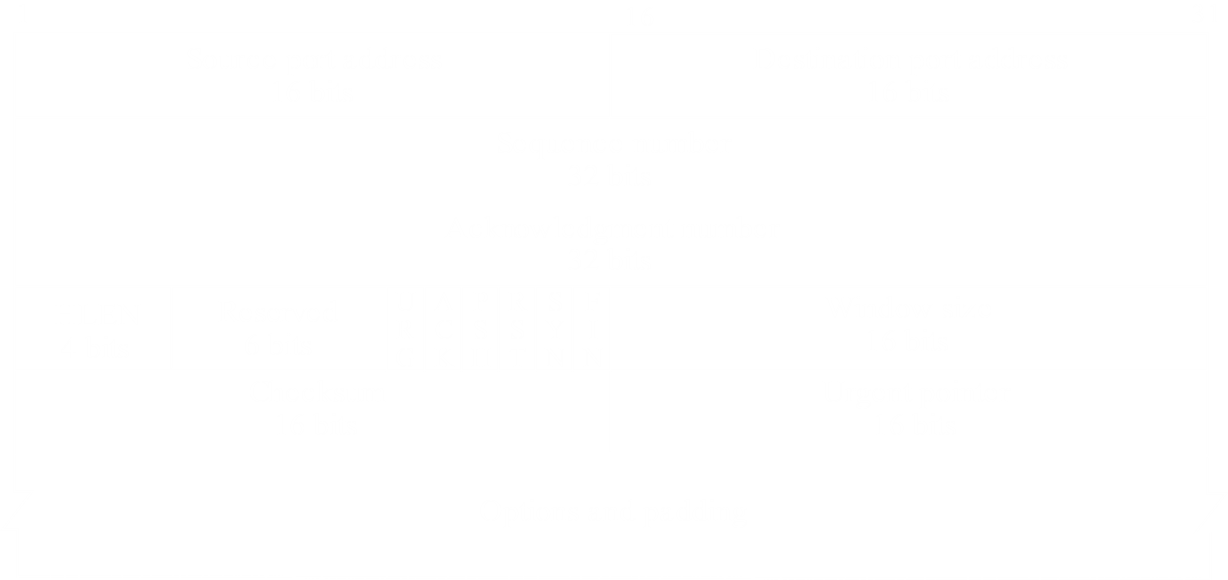
Since TCP is **reliable**, it also needs to send **acknowledgements** for every segment.

## 15.3 Segment



A TCP **segment** consists of a **header** of **20-60 bytes** and some data. Just like the IPv4 header, the 20 bytes are the minimum requirement and the extra 40 bytes are for **options**.

The **header** itself consists of several fields.

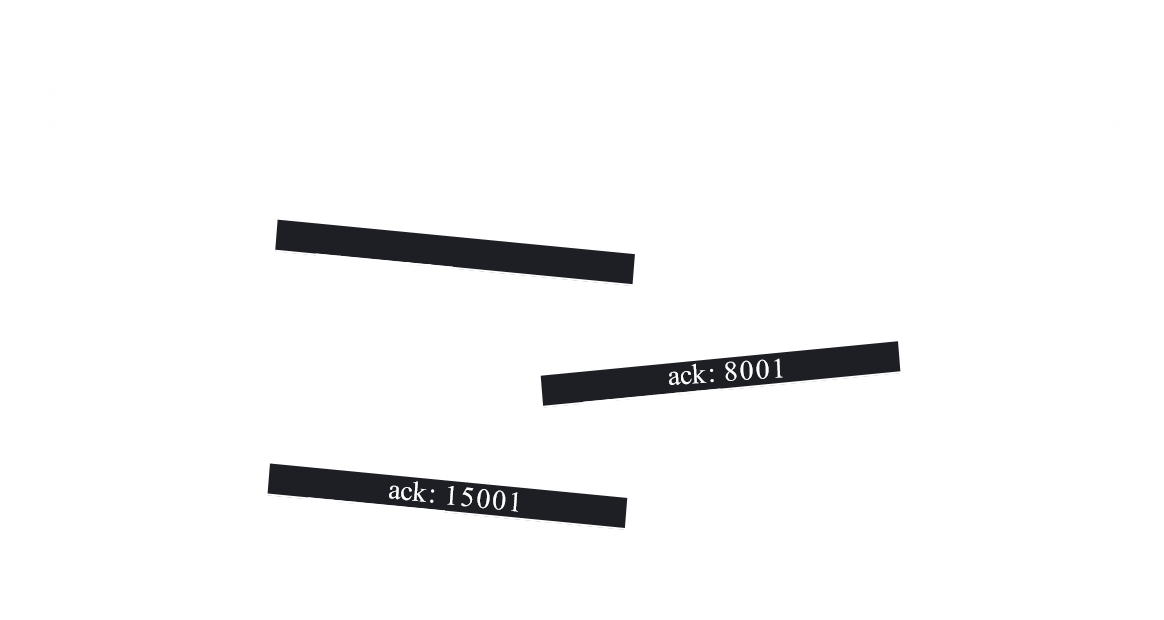


* **Source Port Address** (16 bits)
* **Destination Port Address** (16 bits)
* **Sequence Number** (32 bits) – This is required for flow control. Since there are 32 bits, a single segment can have a maximum of bytes.
* **Acknowledgement Number** (32 bits) – This is the number of the **next byte** which the receiver expects to receive.
* **HLEN** (4 bits) – This is the header length.
* **Reserved** (6 bits)
* **Control** (6 bits) – These are 6 **flags** used by TCP for control purposes. These include:
  + **URG** - The **Urgent Flag** is set to indicate that there is some urgent data that must be sent out immediately, without following the typical order in which it would have been sent out. The actual data can be found using the **Urgent Pointer** field.
  + **ACK** – The **Acknowledgement Flag** is related to connection establishment and teardown.
  + **PSH** – The **Push Flag** is used to push the data to the upper layer. Data is normally sent if the receiving TCP’s buffer is full. This can be overridden to send the data immediately by setting this flag.
  + **RST** – The **Reset Flag** is related to connection establishment and teardown.
  + **SYN** – The **Synchronization Flag** is related to connection establishment and teardown.
  + **FIN** – The **Finished Flag** is related to connection establishment and teardown.
* **Window Size** (16 bits) – This is the remaining space in the receiver’s window so that the sender knows how much more data it can send.
* **Checksum** (16 bits)
* **Urgent Pointer** (16 bits) – This is related to the **URG flag**. This points to the data that must be sent urgently.
* **Options and Padding**

## 15.4 A TCP Connection

The main thing that sets TCP apart is the fact that it is **connection oriented**.

### Connection Establishment



**Connection establishment** involves just **three messages** and the process is called a **three-way handshake**.

1. **SYN** - A message from the Client to the Server, informing the server that the client wants to establish a connection
2. **SYN + ACK** - A reply from the server
3. **ACK** - An acknowledgement from the client

The first thing to happen is **Passive Open**. This is the Server’s process opening up a **socket** and declaring that it is prepared to receive connections. Next, the **client’s process** requests a connection to be made from its own device. This is called **Active Open**. This is when the message exchanges start.

The initial **SYN message** includes a **sequence number**, which is used to synchronize the starting point of the communication from the client. This is required since the starting sequence number is actually a **randomly selected value** of 32-bits.

If the SYN message has a value of for the sequence number, the **SYN + ACK message** will have the value in the **ACK field**, since the server expects that sequence next. However, the server is also sending a **sequence number**, its own starting sequence number. This is so that the client can synchronize with communication from the server. Once the client receives this message, the connection is open from its side.

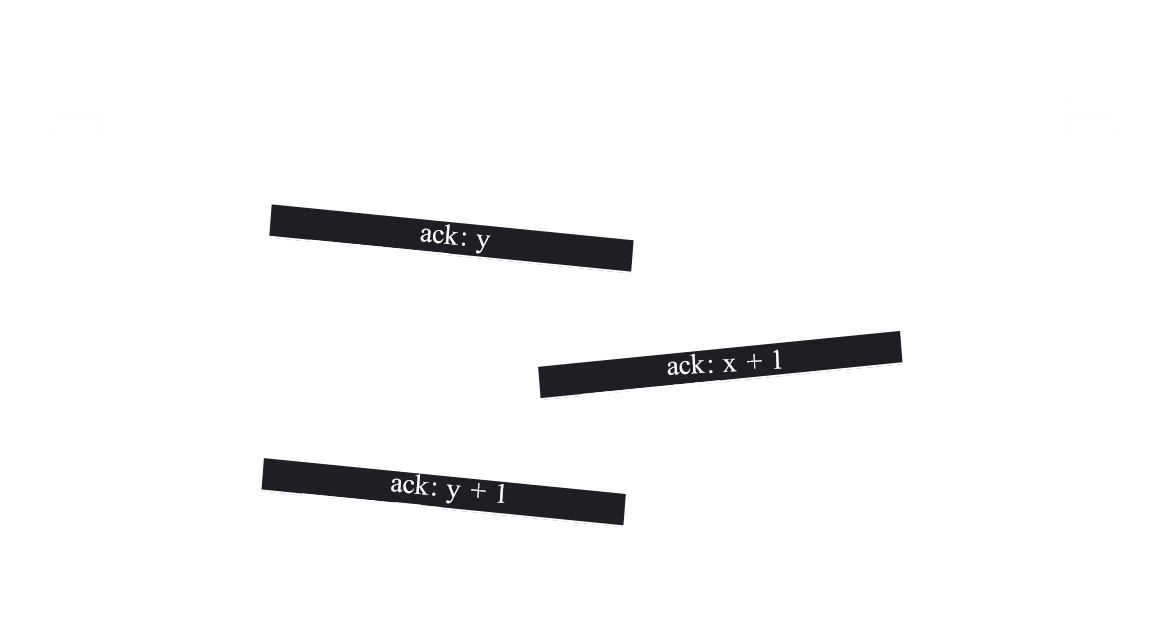
Finally, the client sends an **ACK message**. If the initial sequence number sent by the server was , the **ACK field** for this message will have the value . Once the server receives this message, the connection is open from its side.

Notice that the last **ACK message** sent by the client still has the initial sequence number. This tells us that the ACK message **does not consume** a sequence number. However, note that a **SYN message**, either alone or combined with an ACK message, **does consume** a sequence number.

In both the SYN + ACK and the ACK messages, there is a field labelled **rwnd**, which stands for **Receiving Window**. This value indicates the **number of bytes** the receiver (the one sending the ACK message) has free in its **buffer**, so that the sender knows how much more data it can send.

### Connection Termination

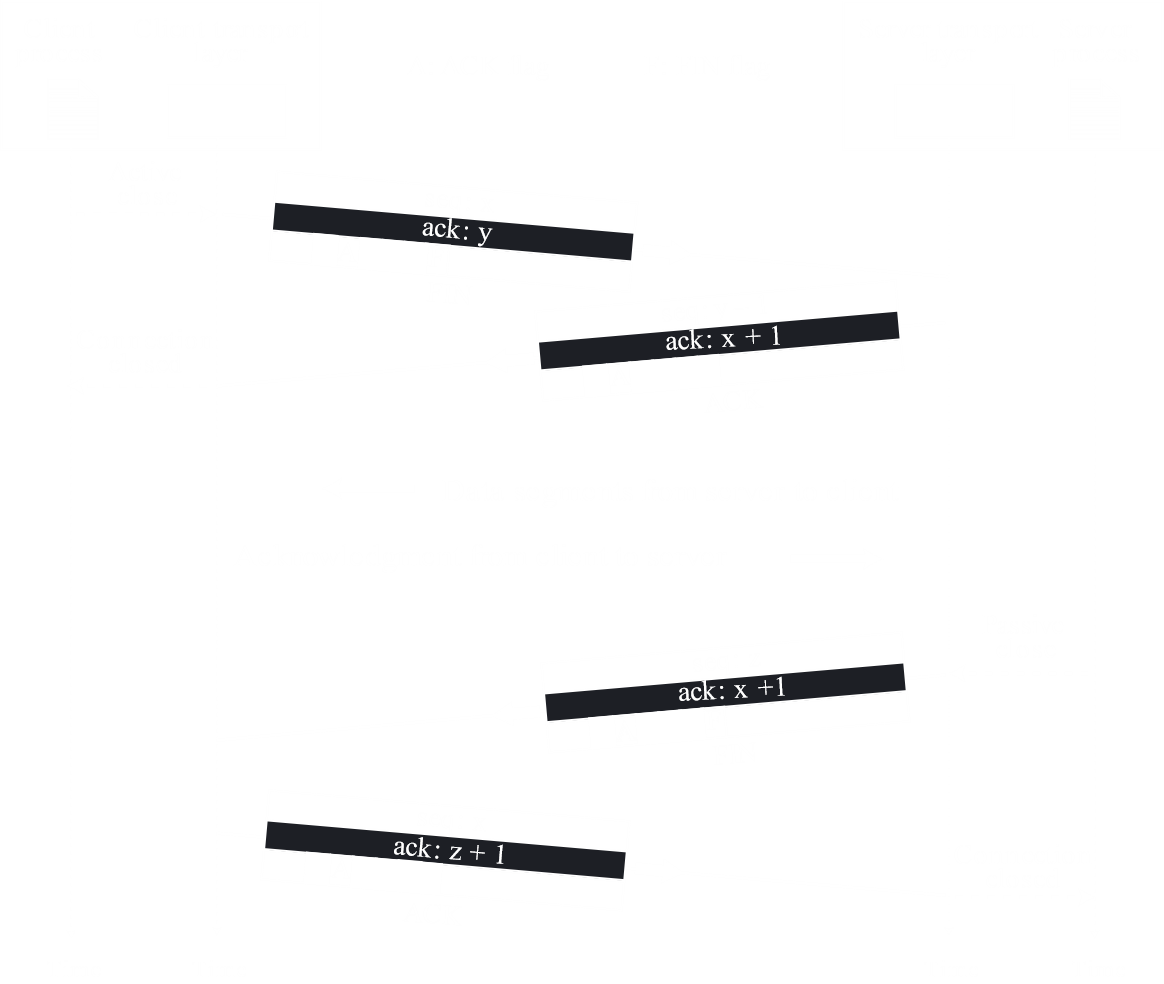
Once a connection is established, **bi-directional data transfer** can take place. Once all the data is transferred, we enter the **Connection Termination** stage.



Connection termination also involves **three messages**:

1. **FIN** – A FIN segment has the FIN flag set. This is sent from the client to the server, requesting that the connection be closed. It is sent when the client process requests its own machine to do so, a process called **Active Close**. Like the SYN message, the FIN message also **consumes a sequence number**.
2. **FIN + ACK**- The server acknowledges the FIN request from the client and also sends its own FIN request. This involves the server’s process agreeing to close its own connection, called a **Passive Close**. When the client receives this message, it closes the connection from its side.
3. **ACK** – The client sends a final acknowledgement to the server, confirming that the connection on the server’s side can also be closed.

There are however, situations where a **three-way handshake** is not appropriate for the connection termination stage. Perhaps the client is requesting that the connection be terminated, but the server is not ready to do so, i.e. it still has data to send. In this case, a **four-way handshake with half-close** is used. The only difference is that the server sends an **ACK message**, allowing the connection on the client end to be closed, but not a **FIN message**. Instead, it sends the FIN message when it is ready to close the connection on the server side, at which point the client responds with its own ACK message. Notice that the **Passive Close** occurs when the server sends its FIN message, not when it acknowledges the client’s FIN message.



Once the connection on the client-side is closed, communication from the client can no longer occur, but communication from the server can.

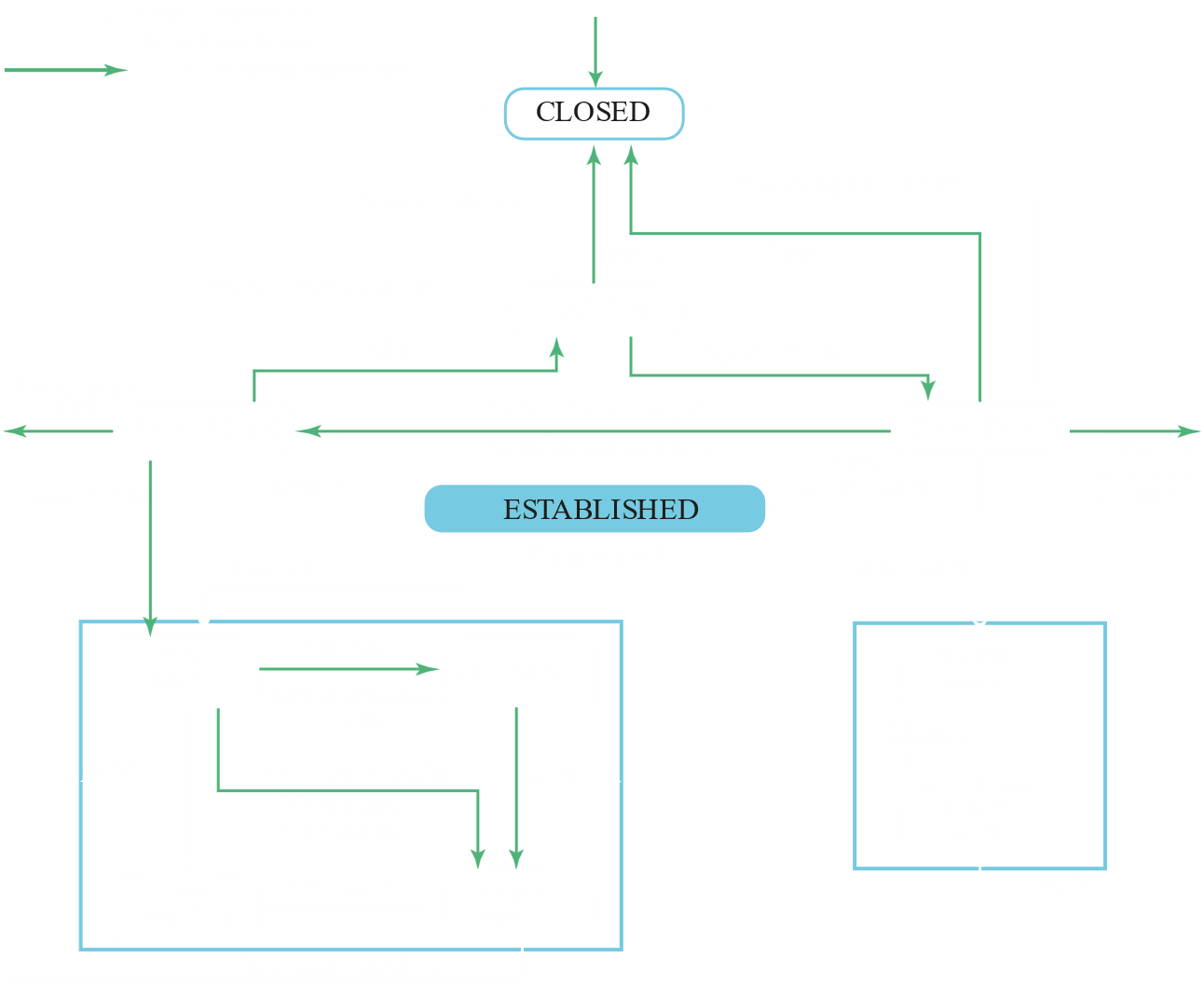
### SYN Flooding Attacks

A **SYN Flooding Attack** involves a **large number** of **SYN messages** being sent to the same server from **invalid clients**. This causes the server to be unable to serve actual clients.

This issue was solved by the **SCTP Protocol**. We will look into how this was done when studying SCTP.

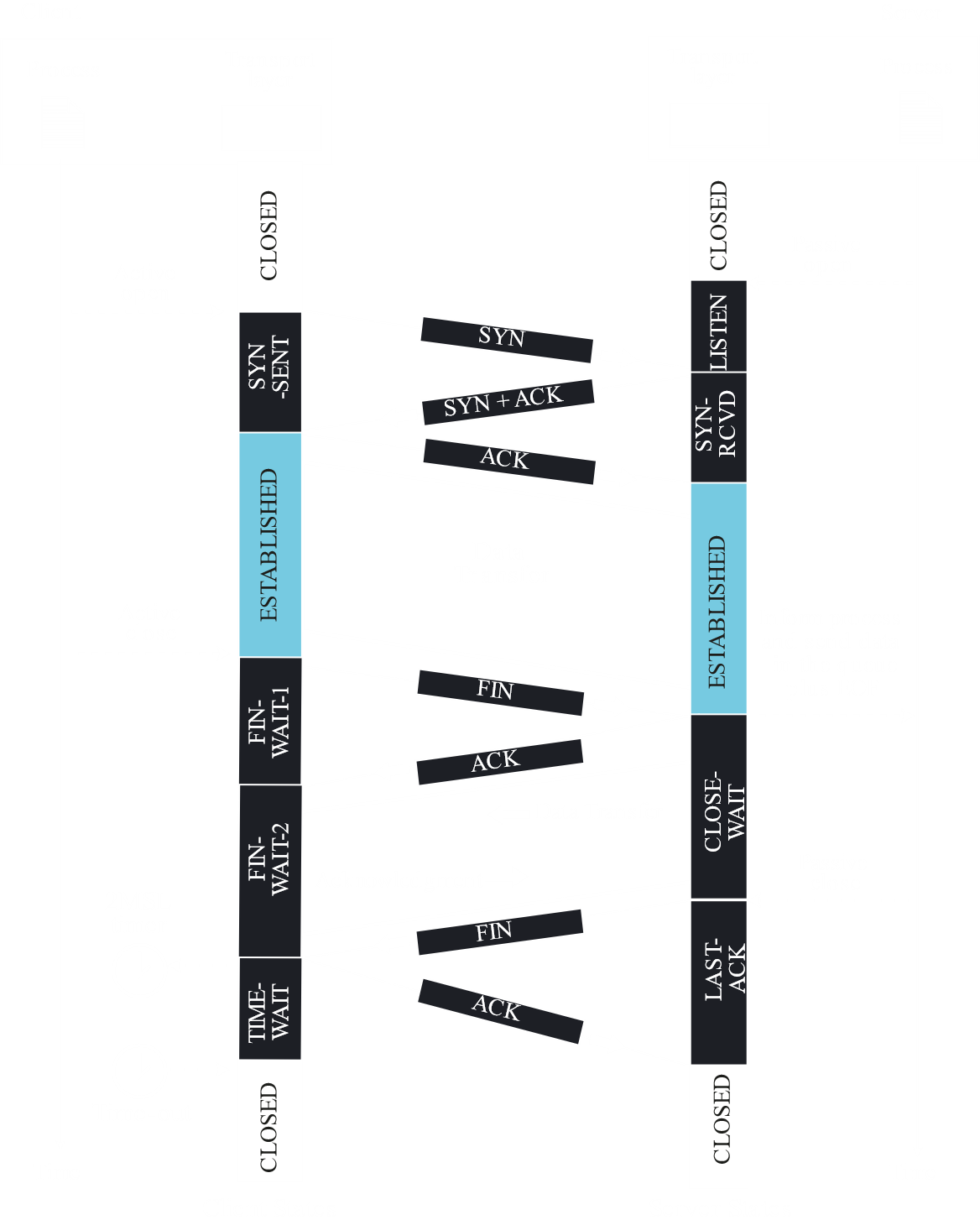
## 15.5 State Transition Diagram

The entire process of connection establishment and termination that we saw in the previous section is managed in TCP with the help of the **Finite State Machine** described in the State Transition Diagram below:



The edges in this diagram are in the format **input/output**. Thus, if we consider a client device that is in the SYN-SENT state, the client receiving a SYN + ACK at this state will cause it to give an ACK as output and move to the ESTABLISHED state.

The same diagram as above can be viewed as a **time-line** diagram:



One important thing to notice is the **2MSL Timer**. When the client receives a **FIN message** from the server and sends back an **ACK message**, it does not immediately go to the CLOSED state. This is because the ACK message might get **lost**, which will cause **another FIN message** to arrive.

The 2MSL timer has enough time to accommodate an ACK message being lost and another FIN message arriving. If the FIN message arrives, then another ACK is sent and the timer is **restarted**. If one does not arrive, it means that it is safe to move to the CLOSED state. If this is not done, the server might not receive an ACK, which will cause it to **remain open forever**.

## 15.6 Windows in TCP

TCP uses **send and receive windows**. We have already discussed this in a previous course, so this section should serve as a reminder.

Since there is **bi-directional communication**, both the client and the server need to maintain a send and receive window, so there are **four windows** in total. For simplicity, we will assume that the send window is on the client side and the receive window is on the server side.

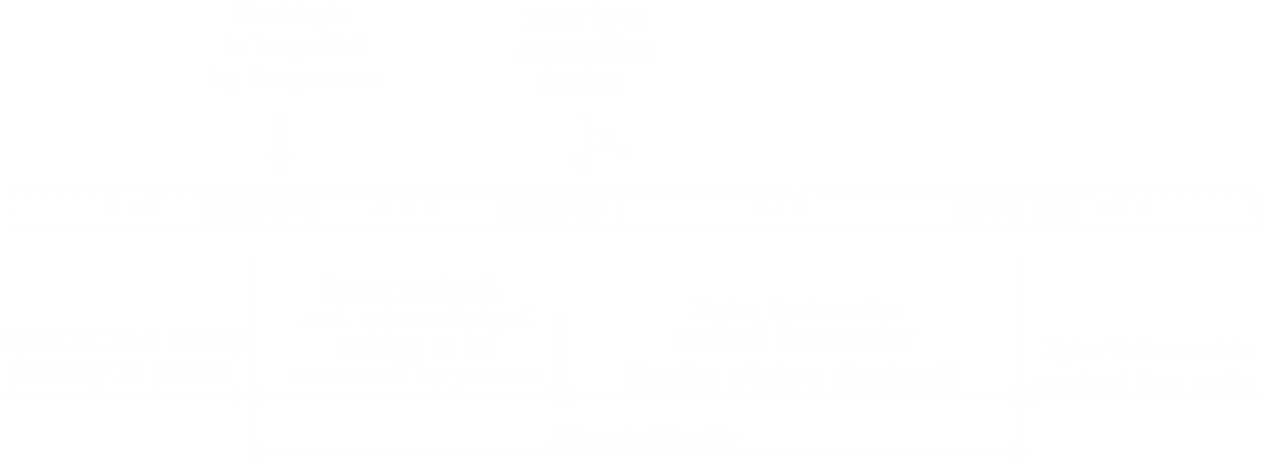


The **Send Window** is a **buffer** containing the **bytes to be sent**. The  **pointer** marks the start of the section of bytes that **have been sent** but have not received any **acknowledgement**. For each acknowledgement, the pointer **moves forward**, called **closing the window**, and the bytes behind it are **removed** from the buffer.

The  **pointer** marks the start of the section of bytes that are **to be sent** next. This continues to the **end of the window**, with the bytes that **cannot be sent** yet being **outside the window**. As it becomes possible to send more bytes, the end of the window is **extended**, called **opening the window**.



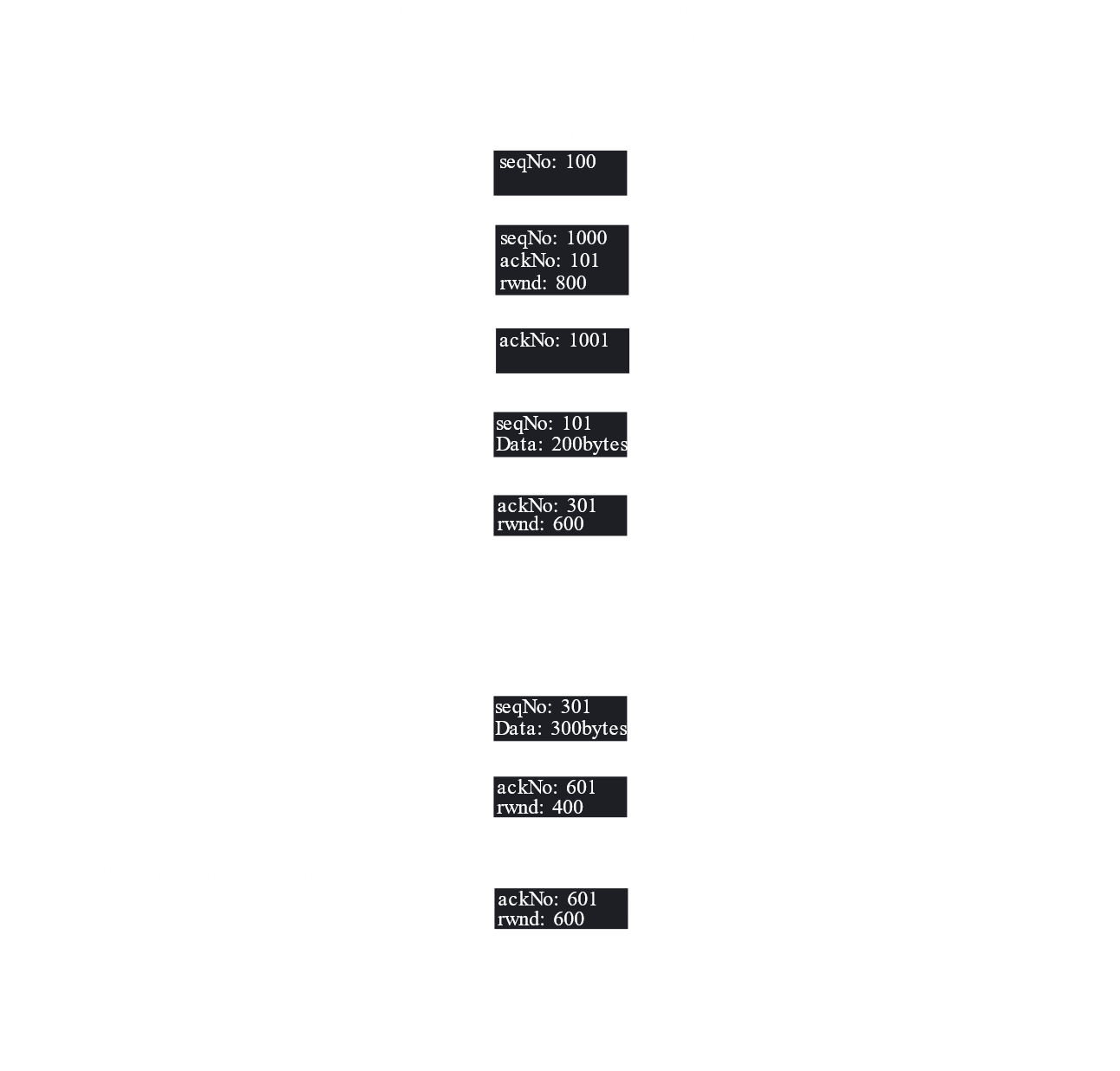
The number of bytes that can be sent is determined by the **rwnd value** sent by the receiver. If the value sent is **larger** than the current send window size, then the window will **open**. If it is **smaller**, then the window will **shrink**, which is to say that the end of the window will move left. However, note that even though this last operation is **possible**, it is not something that ever **happens**. The receiver does not send rwnd values in a way that will cause shrinking.



The **Receive Window** on the other hand, is a little simpler. There is a **buffer**, and as it **receives bytes**, it sends out acknowledgements. However, bytes are not immediately pushed to the process, but are rather **kept in the buffer** until the process **pulls** them, at which the window **closes**. As more space becomes available in the buffer, the window **opens**, which increases the **rwnd value**.

## 15.7 Flow Control

**Flow Control** follows directly from the previous section. As discussed, the **send window size** varies depending on the **rwnd value** sent by the receiver. This value is the remaining space in the buffer in the **receive window**.



Notice that **shrinking** never takes place, even though the **send window** does get **smaller**. This only happens when an **ACK message** has a **smaller rwnd value**, which causes the send window to **close**, but not **open**, thus resulting in a smaller window.

Also notice that towards the end the receiver sends **two acknowledgements**, even though it did not receive any new data. The second acknowledgement is due to the fact that the **receive window opened**. It has the **same ACK value** but an **increased rwnd value**.

We mentioned earlier that to **avoid shrinking**, the receiver must be careful with the **rwnd value** it sends. To do this, an equation is followed:

**New ACK Number + New rwnd Value Last ACK Number + Last rwnd Value**

### Silly Window Syndrome

Say the sender is **producing data** very **slowly** or the receiver is **receiving data** very slowly. The worst-case scenario could be that just 1 byte of data is being transferred with each packet. Even the headers would be much larger than the data in this case. This situation is called the **Silly Window Syndrome**.

If the **sender** is causing the issue, then it can be solved with **Nagle’s Algorithm**, which simply says to **accumulate data** until the **maximum segment size** as decided beforehand by both parties. Alternatively, the segment could be sent when an **acknowledgement** for the previous segment is received. The first scenario occurs if the **application is faster** than the network and the second scenario occurs if the **network is faster** than the application.

If the **receiver** is causing the issue, meaning the receiving process is consuming bytes from the buffer one by one, then **Clark’s Solution** can be used. This dictates that the **rwnd value** be advertised as until there is enough space to accommodate the **maximum segment size** or until **half the buffer** is empty. Alternatively, we could just **not send an acknowledgement** until we have some space available. This could of course cause the sender to keep **retransmitting**, and to avoid that, the maximum delay should not be more than .

## 15.8 Error Control

A **checksum** is used to detect errors in a received packet. If there are errors, an **acknowledgement** is not sent. Not receiving an acknowledgement within a certain time causes a **time-out** to occur on the sender’s side, which triggers a **retransmission**.

There are some rules that need to be followed when generating acknowledgements:

1. We should try to send the ACK for the previous received packet with an outgoing packet, i.e. not generating a separate packet for the ACK itself. This is called **Piggybacking**.
2. **Delay sending** an ACK until another segment arrives or 500ms have passed. This reduces extra traffic. If a new segment arrives, **two ACKs** are sent together.
3. There should never be more than **two unacknowledged** segments, one after another.
4. When an **out-of-order** segment arrives, it should be accepted and an ACK with the **expected segment** should be sent. This causes **fast retransmission**, which we will discuss later.
5. When a **missing segment** arrives, it should be accepted and an ACK with the expected segment should be sent.
6. If a **duplicate segment** arrives, an ACK should be **immediately sent**.

### Fast Retransmission

The sender has a timer, the **retransmission timer**, which is related to packet retransmissions. If the sender does not get an ACK for the **first unacknowledged packet** in the send window by the time this timer runs out, that packet is **retransmitted**.

Say packet 20 gets lost, but packet 21 is **successfully received**. Normally, the sender would wait until the retransmission timer runs out to resend packet 20. However, since the receiver sends an ACK asking for packet 20 when it receives packet 21 (according to rule 4), **fast retransmission** occurs, which causes packet 20 to be retransmitted **immediately**.

However, it is still possible that the ACK was sent by the receiver after it received packet 19. To avoid confusion, the sender waits until it receives the same ACK **three times**, sent after three consecutive unexpected segments arrive, at which point it disregards the retransmission timer. When the receiver gets the missing packet 20, it sends the next ACK immediately, which asks for packet 24 (since 21, 22 and 23 have been received in those three transmissions).



### Deadlocks

Say the receiver sends a **rwnd value** of 0, meaning the sender cannot send anything. After a while, another ACK is sent which has a positive rwnd value, but this ACK is lost. This will cause the sender to simply not send packets ever again. Such a situation is called a **deadlock**.

To get around this, the sender also has another timer, called the **persistence timer**. At the end of this time, the sender checks with the receiver again.

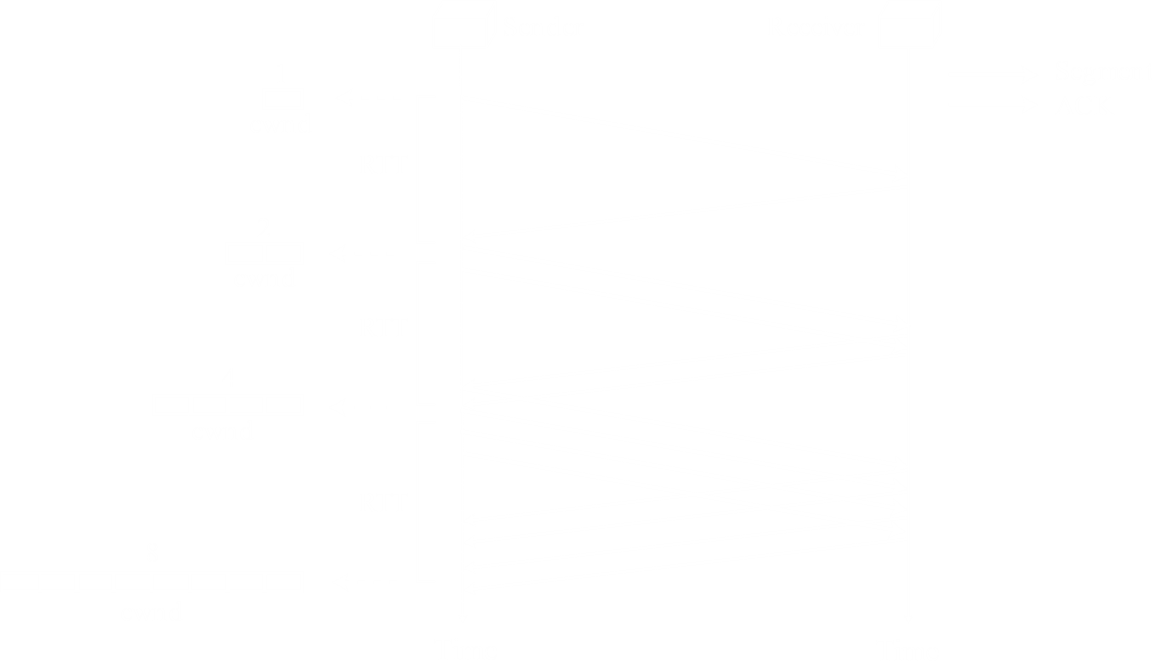
## 15.9 Congestion Control

**Congestion** occurs when a network is carrying **too much data**, which causes reduced network quality in the form of things like **packet loss**. To help prevent this from happening, **congestion control** was introduced.

For congestion control, the **sender** maintains a new window, the **congestion window**. The size of the **send window** now depends on both the congestion window and the **receive window**. The size of the send window will be the **minimum** of the two. For simplicity, we will assume that the congestion window is always smaller than the receive window.

### Exponential Increase

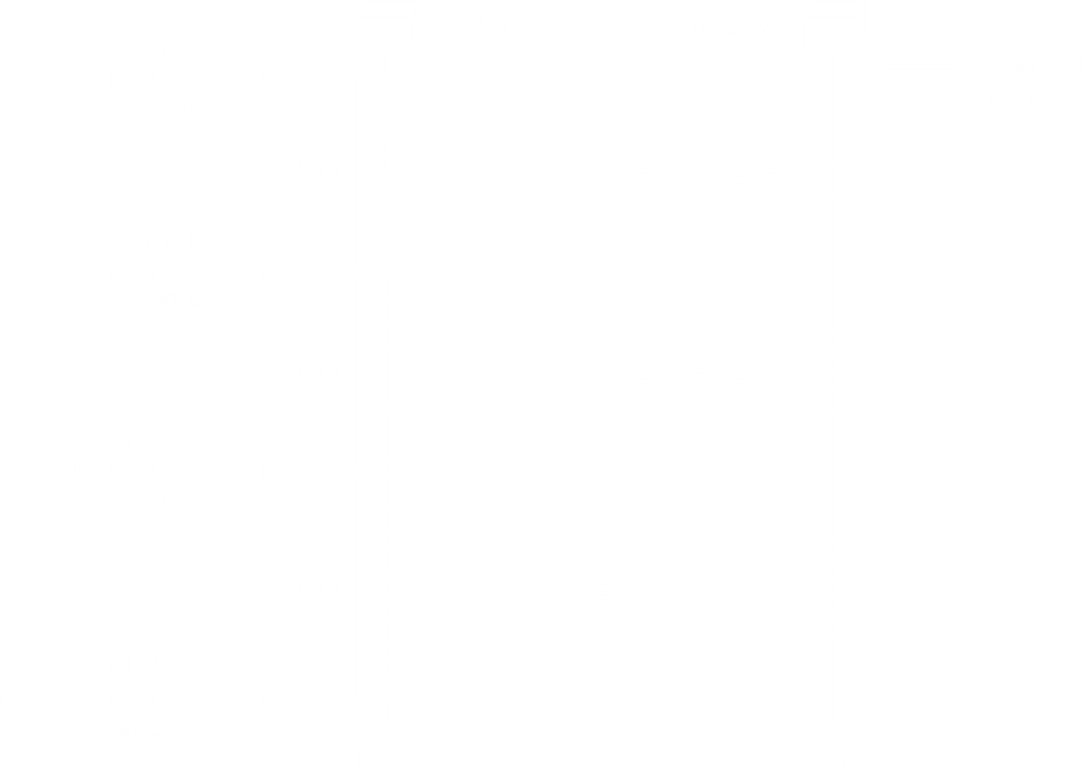
Congestion control occurs is several **phases**. The first phase is the **exponential increase phase** or the **slow start phase**. Here, the sender maintains a very **small congestion window** to start with, with a size of . For every set of ACKs it receives, the window size is **doubled**.



Notice that when the window size is increased, the corresponding number of packets is being sent. Only when **all the packets** are acknowledged does the window size double again.

### Additive Increase

The above policy is maintained until the congestion window size becomes equal to a **threshold value**, as defined by the network. At this point, we enter the second phase, the **congestion avoidance phase** or the **additive increase phase**. In this phase, for a **complete set of ACKs**, the window size is **incremented by** .



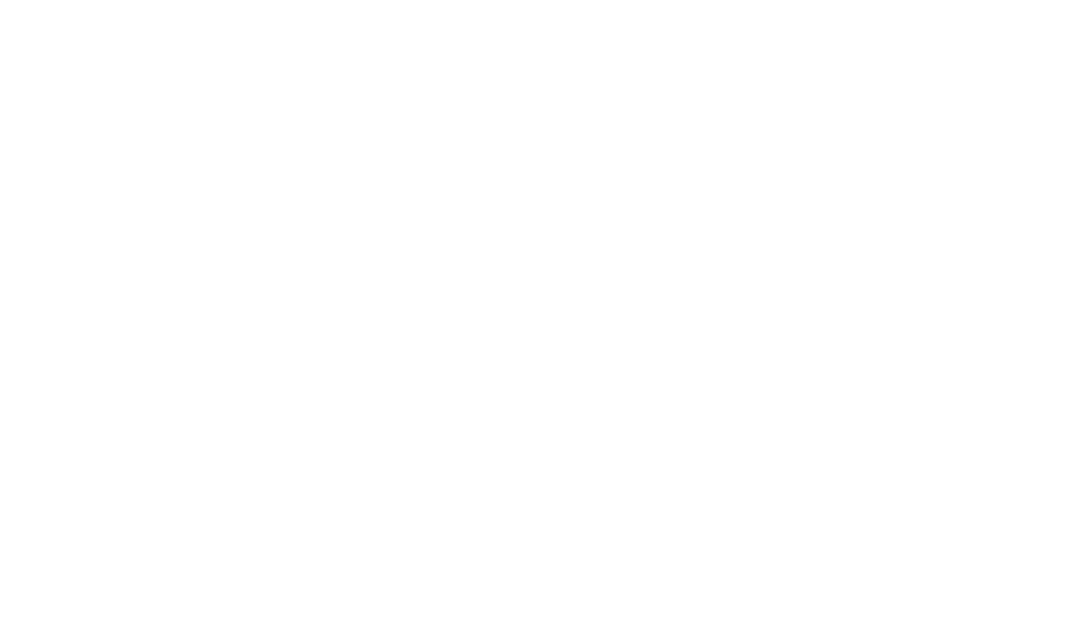
### Multiplicative Decrease

The above policy is maintained until a **congestion is detected**. Once this happens, we enter the **congestion detection phase** or the **multiplicative decrease phase**.

A congestion is detected when there is a need for a **retransmission**. The fact that we are retransmitting a packet means that somewhere in the network, the **packet was lost**. This can only have happened if there was a node along the path which had a **full buffer**, which resulted in the packet being dropped.

There are two ways in which a retransmission might occur. The first is if the **RTO timer runs out**. This means that either the receiver never received the packet or the ACK for the packet was lost. In either case, this is the result of **serious congestion** in the network. The second is if we receive **three duplicate ACKs**. This is **less serious**, since it means that there was a packet lost in between, but packets after that one were received properly.

The congestion policy handles the two retransmission scenarios differently due to the different urgencies.

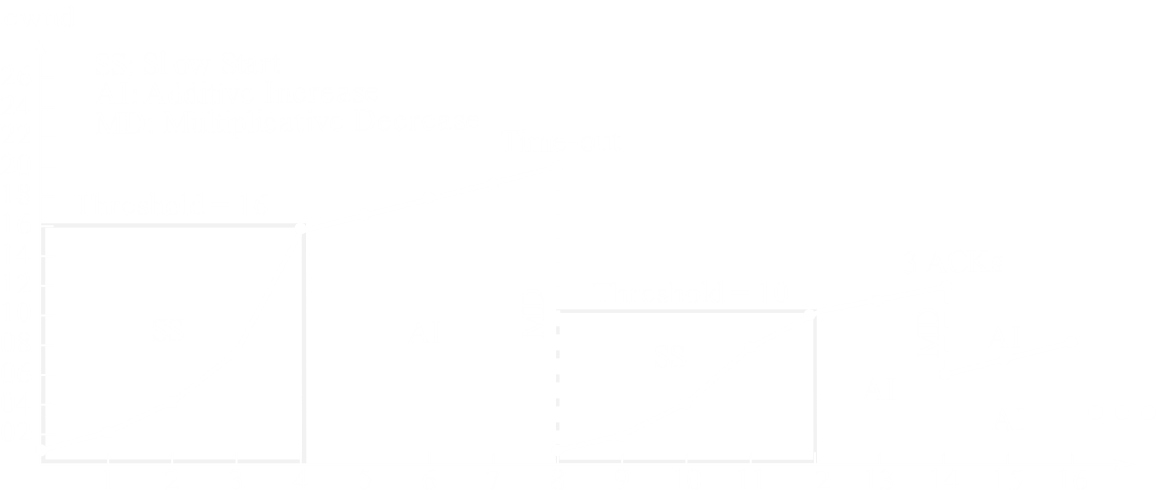


Regardless of the retransmission reason, the **threshold value** will immediately become **half the current window size**.

If the retransmission occurs due to an **RTO timeout**, the window size is then set to and the **slow start phase** starts over.

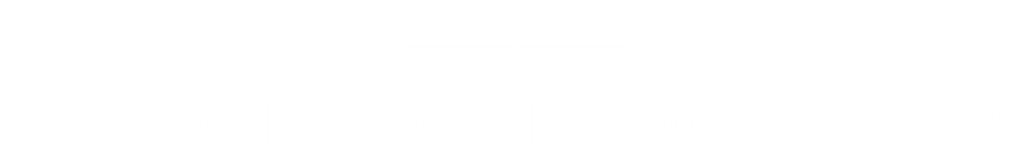
If the retransmission occurs due to **three duplicate ACKs**, the window size is set to the threshold value and the **congestion avoidance** phase is started again.

Example



## 15.10 TCP Timers

In TCP, we have four different types of timers:



### Retransmission

Initially, a separate **retransmission timer** was maintained for every segment, but in modern TCP only a **single timer** is maintained. If the timer expires, all the **outstanding segments** are retransmitted.

The value of the retransmission timer is the same as the **round-trip time** (RTT). However, the RTT value can vary depending on the amount of network traffic there is. Because of this, the RTT must be **calculated**.

In TCP, every segment might not receive an ACK. We can have a single ACK for multiple segments. Due to this, TCP has created a restriction that there can only be a **single RTT calculation** at any given time.

However, the measured RTT () value is not directly used. From the RTT, we calculate the **smoothed RTT** () and the **RTT deviation** ().

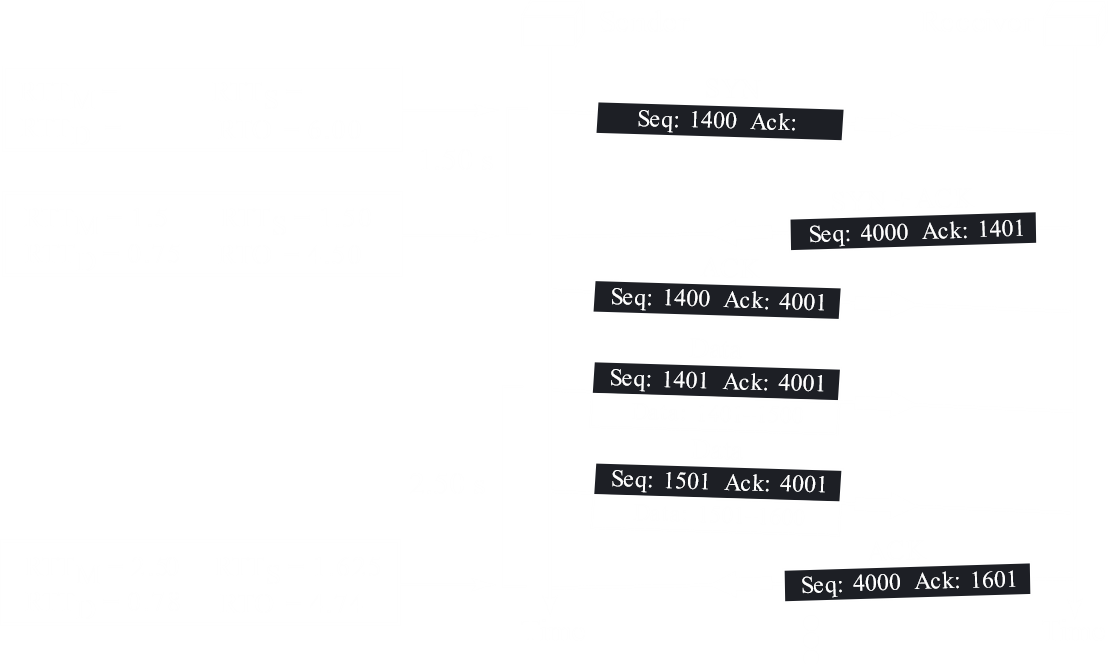
After the **first measurement**,

For every measurement after that:

Here, and

Finally, the RTO value is calculated as

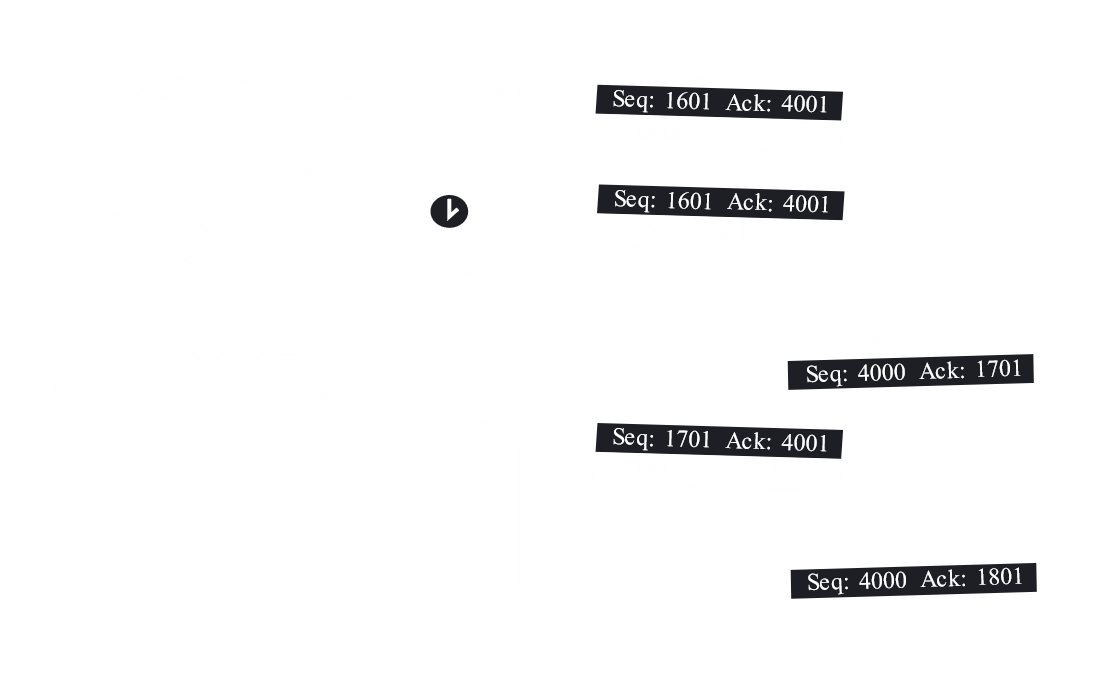
Example



If we actually **face a retransmission** however, the RTO value is **doubled**. This is because, since the packet is lost, the exact required RTO taking into account the congestion cannot be calculated. Thus, this is used as a thumb rule. This is called **Karn’s Algorithm**. Note that the and values **do not change**.

The new RTO value is maintained until **another measurement** can be done.

Example



### Persistence Timer

As discussed previously, when the **receiver window size** becomes **zero**, the **sender** starts a **persistence timer**. When this timer runs out, the sender **probes** the receiver to check if it is ready to receive more packets yet. This helps avoid a **deadlock** scenario, where the receiver’s ACK updating the window size is lost and the sender keeps waiting forever.

The persistence timer initially has a value which is the **same as the retransmission timer**. For every **negative response** from the receiver, the value is **doubled** up to a maximum value of **60s**.

### Keepalive

The **Keepalive Timer** is used by the **server** to check whether the client is still active. If there is **no communication** by the time this timer runs out, we **probe** the client to check if it is still active. Probes are sent every **75s** for a total of **10 probes**, meaning 750s. If there is still no response, the connection is **terminated**.

We might not get a response in a scenario where the client **crashed**, which caused them to be unable to terminate the connection. If there is any communication at all, the timer is **reset**. The value for this timer is **2 hours**.

### Time-Wait

We have discussed the **Time-Wait Timer** earlier. It is used during TCP **connection termination**, where the sender waits a while after sending the ACK to close the connection, which allows for enough time for the ACK to be lost and a retransmission to occur.