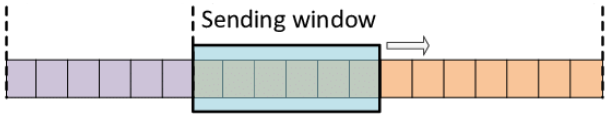
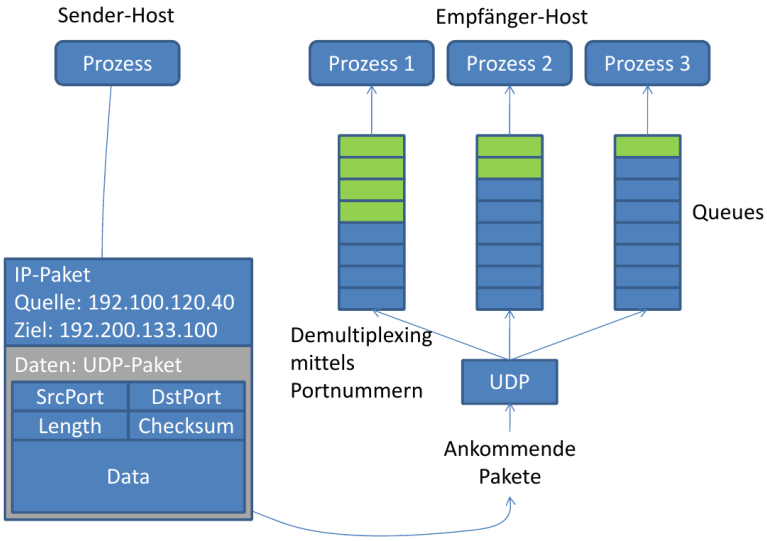
TCP / UDP



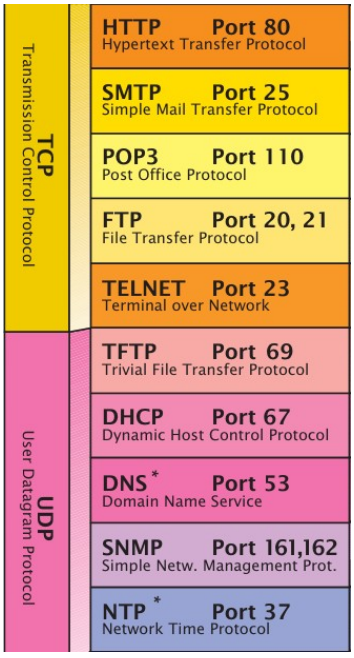
Einfacher (De)Multiplexer

Ein Multiplexer (kurz: MUX oder Mux) ist eine Selektionsschaltung in der analogen und digitalen Elektronik, mit der aus einer Anzahl von Eingangssignalen eines ausgewählt und an den Ausgang durchgeschaltet werden kann. Multiplexer sind vergleichbar mit Drehschaltern, die nicht von Hand, sondern mit elektronischen Signalen gestellt werden. Der Unterschied zum Relais besteht darin, dass die Verbindungen nicht mechanisch, sondern (heutzutage) durch integrierte Halbleiterschaltungen zustande kommen.



Bei zyklischem zeitlichem Durchlauf können mit einem Multiplexer parallele Datenströme in serielle gewandelt werden. Das Gegenstück zum **Multiplexer** ist der **Demultiplexer**, mit dem die zusammengefassten Datenkanäle wieder aufgetrennt werden. Analoge Multiplexer arbeiten bidirektional, das heißt, sie können auch als Demultiplexer verwendet werden.

Neben mehreren Eingängen (Queues) und einem Ausgang verfügt ein Multiplexer über ein oder mehrere Steuersignale, über die festgelegt wird, welcher Eingang (Ports) ausgewählt wird.

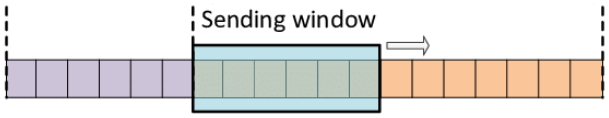


Sockets

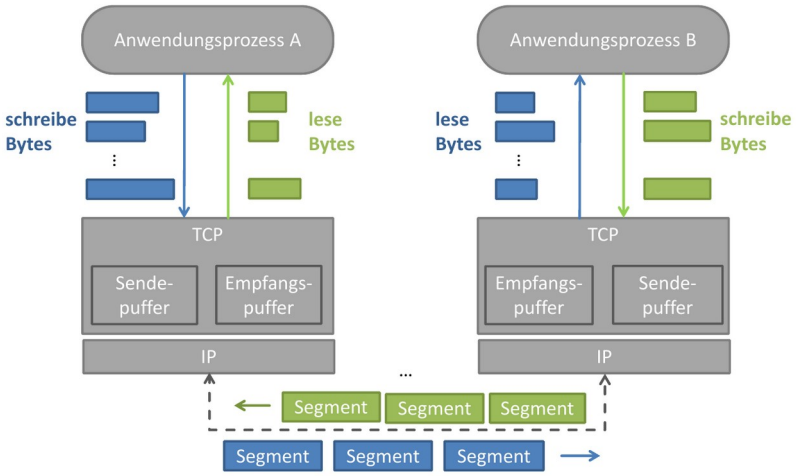
Ports sind 16 Bit breite Zahlen, die eindeutig eine Anwendung auf dem Host adressiert. Ein Socket ist ein vom Betriebssystem bereitgestelltes Objekt aus IP-Adresse und Port-Nummer, das als Kommunikationsendpunkt dient. Ein Programm verwendet Sockets, um Daten mit anderen Programmen auszutauschen. Das andere Programm kann sich dabei auf demselben Computer (Interprozesskommunikation) oder einem anderen, via Netzwerk erreichbaren Computer befinden. Die Kommunikation über Sockets erfolgt in der Regel bidirektional, das heißt, über das Socket können Daten sowohl empfangen als auch gesendet werden.

Quelle: https://www.extrahop.com/company/blog/2017/tcp-windowing/

TCP / UDP



Schicht 4 arbeitet mit Queues (vergleichbar mit Hochregalen). Jedes mal, wenn ein Port für eine Anwendung reserviert wird, wird gleichzeitig ein solcher Zwischenspeicher reserviert. Das macht das Betriebssystem. Wichtig ist, dass sowohl der Absender-Socket und der Empfangs-Socket einen solchen Zwischenspeicher besitzt. Hier fangen dann die Probleme an, denn dieser Speicher muss nicht gleich groß sein, die maximale Größe der zu übertragenen Datagramme (UDP) oder Segmente (TCP) müssen nicht übereinstimmen, etc…



Unterschiede zwischen TCP und UDP

Transmission Control Protocol (TCP)

- Verbindungsorientiert
- PDU: Segmente
- Zuverlässig: verlorene Segmente werden neu gesendet
- Datenrekonstruktion beim Empfänger (Reassemblierung)
- Flusskontrolle (Steuerung der Übertragungsgeschwindigkeit)
- Overhead min. 20 Octets
- Liber Tine Bsp.: HTTP(S), SMTP, FTP, SSH
- RFC793

User Datagram Protocol (UDP)

- Verbindungslos
- PDU: Datagramme
- "Best effort" Ansatz, verlorene Segmente sind erst mal weg
- Keine Flusskontrolle oder geordnete Datenrekonstruktion
- Kleiner Overhead von 8 Octets
- Bsp.: DNS (i.d.R.), VoIP, Videostr., Spiele, TFTP, DHCPv4, SNMP
- RFC768

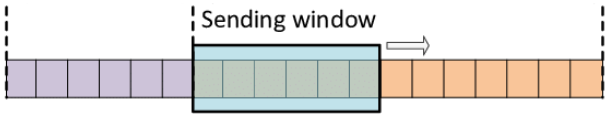
Verbindungsaufbau und -abbau

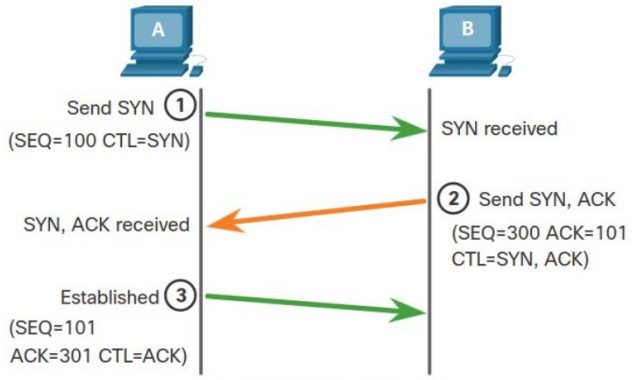
Beim Verbindungsaufbau werden nacheinander die Flags SYN → SYN/ACK → ACK gesetzt:

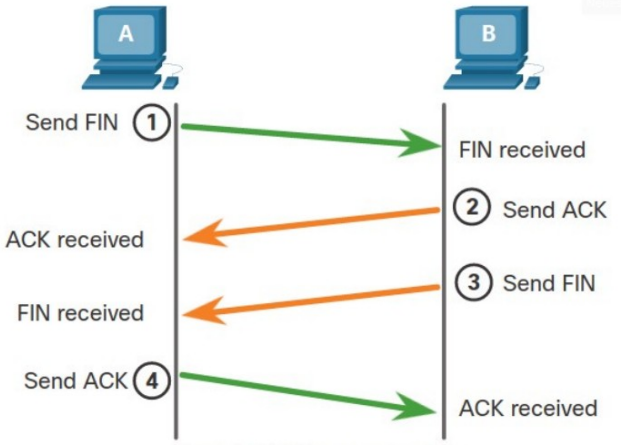
Beim Verbindungsaufbau (auch Double Half Close bezeichnet) werden nacheinander die Flags FIN → ACK und nochmal von der anderen Seite FIN → ACK:

Quelle: https://www.extrahop.com/company/blog/2017/tcp-windowing/

TCP / UDP







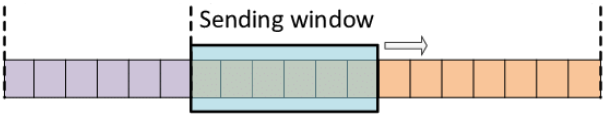
Aufgabe:

Nachfolgend soll ein Echo-Server und ein Echo-Client mit Java programmiert werden. Der Quellcode ist nachfolgend zu sehen:

Server:

```
import java.net.*;
import java.io.*;

public class Server
{
    //initialize socket and input stream
    private Socket          socket   = null;
    private ServerSocket    server   = null;
    private DataInputStream in       = null;

    // constructor with port
    public Server(int port)
    {
        // starts server and waits for a connection
        try
        {
            server = new ServerSocket(port);
            System.out.println("Server started");

            System.out.println("Waiting for a client ...");

            socket = server.accept();
            System.out.println("Client accepted");
            // takes input from the client socket
            in = new DataInputStream(
                new BufferedInputStream(socket.getInputStream()));

            String line = "";

            // reads message from client until "Over" is sent
            while (!line.equals("Over"))
            {
```


Quelle: https://www.extrahop.com/company/blog/2017/tcp-windowing/



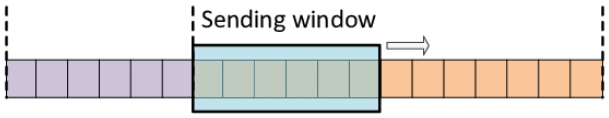

```
                try
                {
                    line = in.readUTF();
                    System.out.println(line);

                }
                catch(IOException i)
                {
                    System.out.println(i);
                }
            }
            System.out.println("Closing connection");

            // close connection
            socket.close();
            in.close();
        }
        catch(IOException i)
        {
            System.out.println(i);
        }
    }

    public static void main(String args[])
    {
        Server server = new Server(5000);
    }
}
```

Client:

```
import java.net.*;
import java.io.*;

public class Client {
    // initialize socket and input output streams
    private Socket socket            = null;
    private DataInputStream  input   = null;
    private DataOutputStream out     = null;

    // constructor to put ip address and port
    public Client(String address, int port) {
        // establish a connection
        try {
            socket = new Socket(address, port);
            System.out.println("Connected");

            // takes input from terminal
            input  = new DataInputStream(System.in);

            // sends output to the socket
            out = new DataOutputStream(socket.getOutputStream());
        } catch(UnknownHostException u) {
            System.out.println(u);
        } catch(IOException i) {
            System.out.println(i);
```

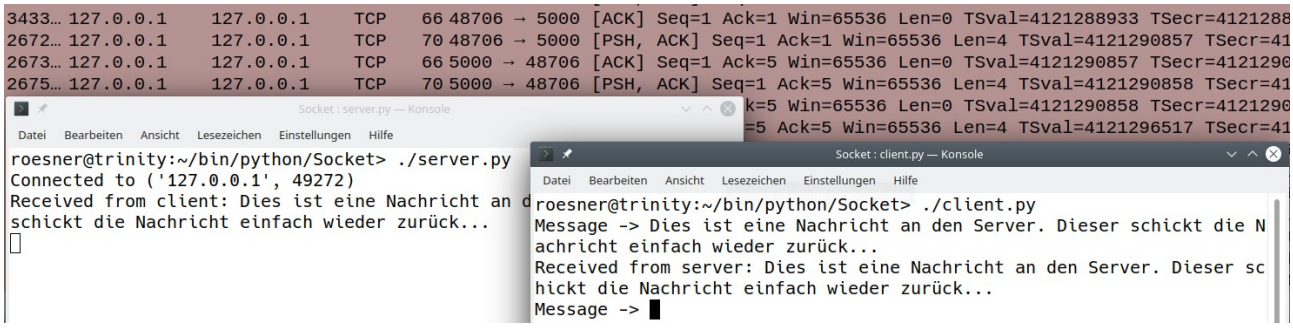

Quelle: https://www.extrahop.com/company/blog/2017/tcp-windowing/

TCP / UDP




```
        }

        // string to read message from input
        String line = "";

        // keep reading until "Over" is input
        while (!line.equals("Over")) {
            try {
                line = input.readLine();
                out.writeUTF(line);
            } catch(IOException i) {
                System.out.println(i);
            }
        }

        // close the connection
        try {
            input.close();
            out.close();
            socket.close();
        } catch(IOException i) {
            System.out.println(i);
        }
    }

    public static void main(String args[]) {
        Client client = new Client("127.0.0.1", 5000);
    }
}
```

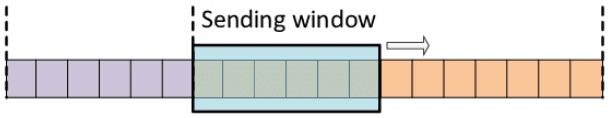
Speichern Sie den Quellcode ab und lassen ihnübersetzen. Starten Sie in einem Terminal den Server und in einem anderen Terminal den Client. Testen Sie die Kommunikation. Danach machen Sie das selbe nochmal nur zeichnen Sie die Kommunikation mit Wireshark auf. Es muss dabei nur auf das Loopback-Device gescannt werden. Wenn Sie fertig sind und die Pakete nicht finden, geben Sie als Filter folgenden String ein: **tcp.port eq 5000**

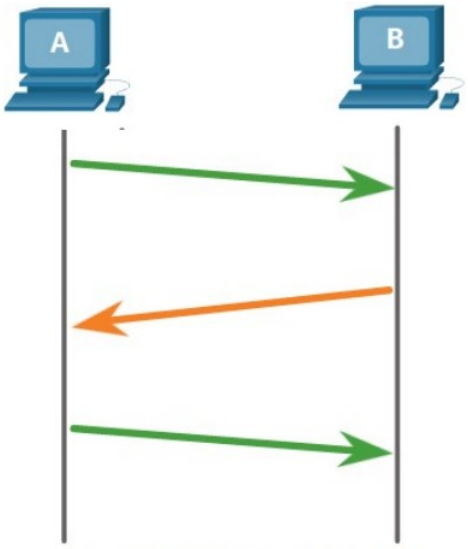


Zeichnen Sie den Verbindungsaufbau, den Datentransfer und den Verbindungsabbau mit allen Daten (Flags und dazugehörenden Werten) auf.

Quelle: https://www.extrahop.com/company/blog/2017/tcp-windowing/

TCP / UDP





Was ist TCP Windowing?

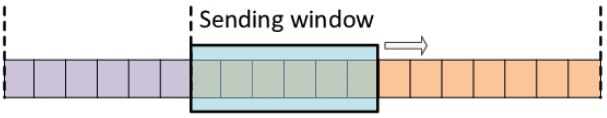
Wenn das Transmission Control Protocol (TCP) auf Schicht 4 ein Schiebefensterprotokoll verwendet, um vielgestaltige Probleme zwischen Clients und Servern zu verringern, die versuchen, zu große oder zu kleine Datensegmente zu senden, spricht man von „TCP-windowing“.

TCP definiert Verbindungen zwischen Hosts über das Netzwerk auf der Transportschicht (L4) und ermöglicht die zuverlässige Kommunikation zwischen Anwendungen auf separaten Geräten. Es wurde entwickelt, um häufig auftretende Netzwerkprobleme zu lösen, einschließlich unzuverlässiger und ungeordneter Paketzustellung, verlorener oder doppelt gesendeter Pakete und Netzwerküberlastung. Beispielsweise definiert TCP, dass ein Server die empfangenen Pakete bestätigt, sodass der Client verlorene Pakete nur bei Bedarf erneut übertragen muss. Es garantiert zudem die geordnete Paketzustellung an die empfangende Anwendung, indem die Sequenznummern der gesendeten und empfangenen Datenblöcke verfolgt werden.

In Bezug auf Effizienz und Netzwerküberlastung gibt es einige Szenarien, die zu Problemen führen können. Clients, die größere Segmente senden, als ein Server zu einem bestimmten Zeitpunkt verarbeiten kann, können zu übermäßigen Neuübertragungen führen. Wenn ein Server jedoch wiederholt eine kleine Empfangsfenstergröße (die Datenmenge, die er empfangen kann) ankündigt, weil er Daten aus seinem Puffer zu langsam verarbeitet, sendet der Client kontinuierlich kleine Segmente anstelle einiger größerer. Diese häufigen kleinen Datenblöcke sind aufgrund der Größe der für jedes Segment erforderlichen TCP-Header (20 Byte) im Verhältnis zu den darin enthaltenen Daten viel weniger effizient.

Quelle: https://www.extrahop.com/company/blog/2017/tcp-windowing/

TCP / UDP



Betrachten wir das TCP-Fenster aus der Perspektive eines Servers als Empfänger und eines Clients als Absender.

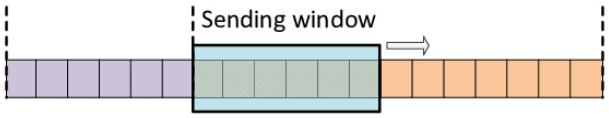
Sliding Window Protocol: Empfänger

Wie oben erwähnt, ist die TCP-Fenstergröße (SWS == sliding window size) (auch als Empfangsfenster bezeichnet) die Menge an freiem Speicherplatz im Empfangspuffer des Servers (Empfänger).

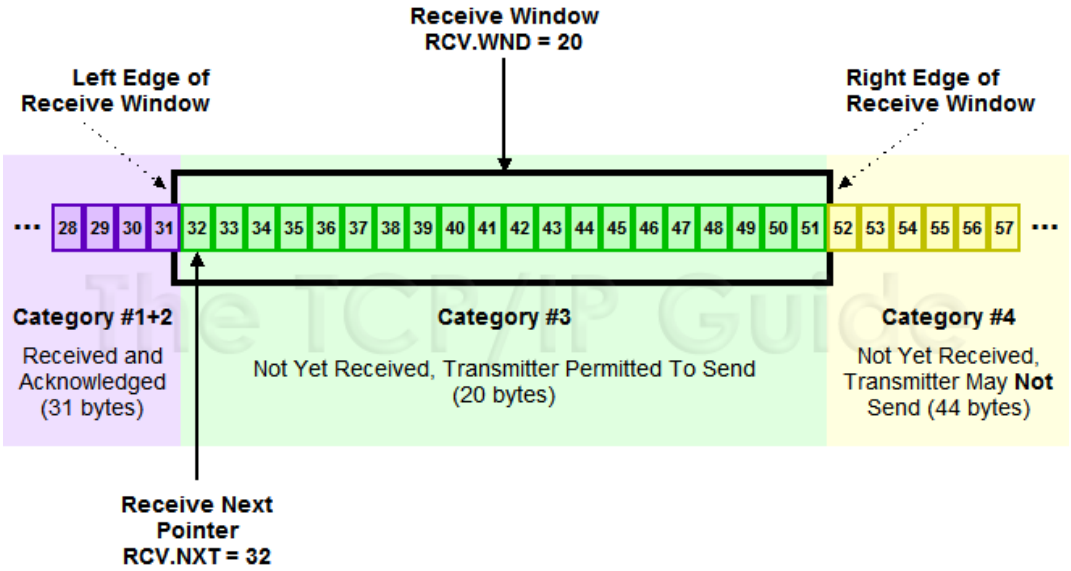
Dieser Wert wird im TCP-Header an den Absender zurückgegeben und begrenzt die Datenmenge, die der Absender übertragen kann, bevor er auf eine weitere Bestätigung wartet. Die Größe des Empfangsfensters wird durch die Puffergröße des Servers und die Datenmenge bestimmt, die er empfangen, aber noch nicht verarbeitet hat. Neben der Fenstergröße enthält die Bestätigung des Empfängers auch ein Feld für die nächste Sequenznummer, die der Server erwartet, und informiert den Client darüber, dass alle Bytes vor dieser Nummer empfangen wurden.

Quelle: https://www.extrahop.com/company/blog/2017/tcp-windowing/

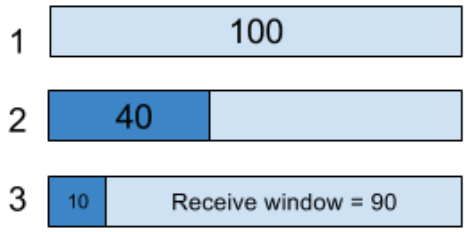
TCP / UDP



Beispiel Empfangsfensterdiagramm.

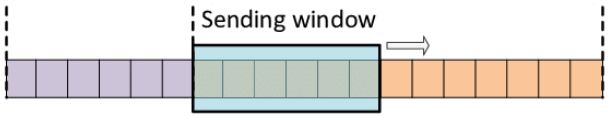


Nehmen wir als einfaches Beispiel an, wir haben einen Server mit einem leeren Empfangspuffer, der 100 Bytes (1) aufnehmen kann. Nach dem Herstellen einer Verbindung überträgt ein Client 40 Bytes. Der Server empfängt alle (2), verarbeitet 30 Bytes (3) und gibt eine **Bestätigungsnumme**r von **41** und eine **Fenstergröße** von **90** an den Absender zurück.



Als nächstes sendet der Client **90** Bytes. Der Server empfängt sie (4) und verarbeitet weitere 30 Bytes aus seinem Puffer (5). Der Server gibt eine Bestätigungsnummer von 131 und eine Fenstergröße von 30 zurück.

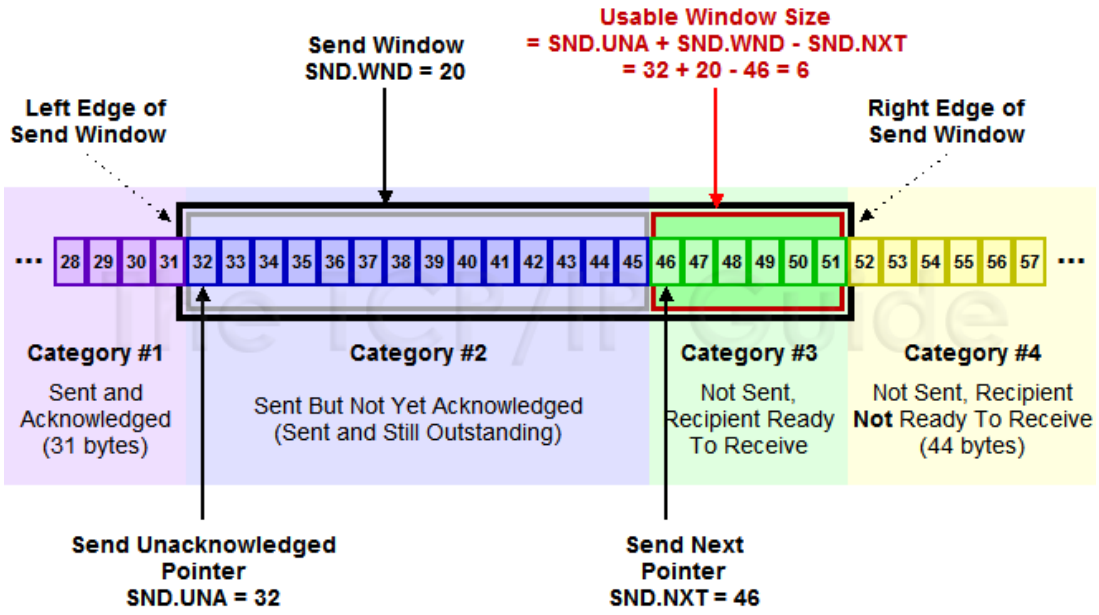
Quelle: https://www.extrahop.com/company/blog/2017/tcp-windowing/



Sliding Window Protocol: Absender

Wie zu erwarten, verfügt der Client über ein Sendefenster, das dem Empfangsfenster des Servers entspricht. Die Größe des Sendefensters gibt die Gesamtzahl der Bytes an, die der Client gleichzeitig **nicht bestätigt** haben muss. Im Allgemeinen ist das Sendefenster durch das Minimum seiner Sendepuffergröße und die Empfangsfenstergröße des Servers begrenzt, es wird jedoch auch von der Netzwerkauslastung und der Anzahl der gesendeten, aber nicht bestätigten Bytes beeinflusst. Der Absender hat auch ein verwendbares Fenster, das eine Teilmenge des Sendefensters für die nicht gesendeten Bytes ist, für die der Puffer des Empfängers noch Platz hat. Mit anderen Worten: Das verwendbare Fenster, ist das Sendefenster abzüglich aller gesendeten Bytes, die noch nicht bestätigt wurden.

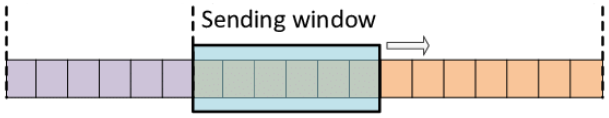
Beispiel Sendefensterdiagramm.

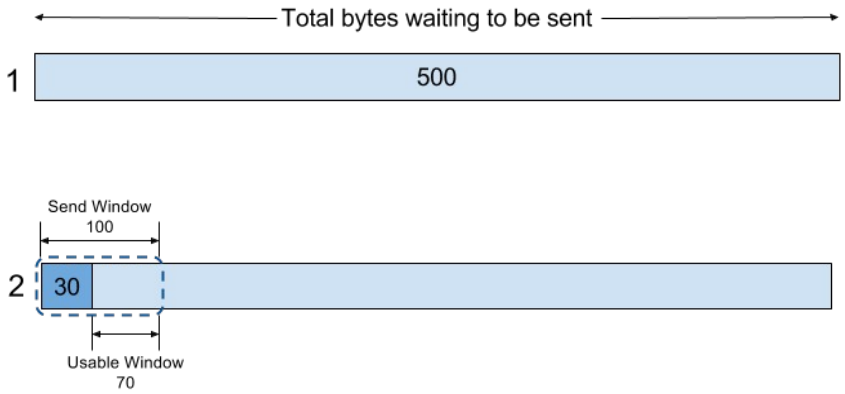


Der "gleitende" Aspekt des Protokolls ist anhand eines vereinfachten visuellen Beispiels für den Betrieb des Absenders leichter zu erkennen:

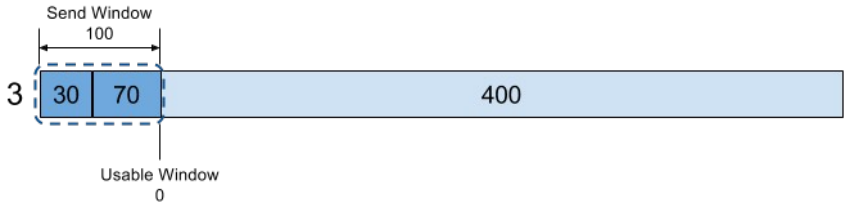
Eine Anwendung auf einem Client verfügt über 500 Bytes, die an einen Server gesendet werden müssen (1). Der Server hat bereits 50 Bytes in seinem 150-Byte-Empfangspuffer, daher legt er eine Empfangsfenstergröße von 100 fest. Nach dem Herstellen einer Verbindung sendet der Client 30 Bytes aus dem 100-Byte-Sendefenster. Da die 30 Bytes noch nicht bestätigt wurden, hat der Client immer noch ein nutzbares Fenster von 70 Bytes (2).

Quelle: https://www.extrahop.com/company/blog/2017/tcp-windowing/

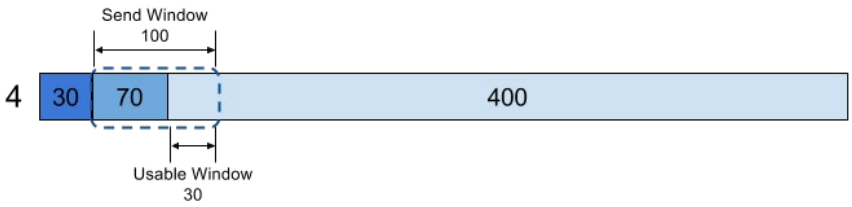




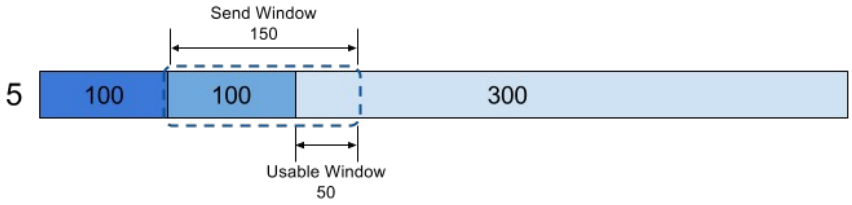
Angenommen, der Client sendet die nächsten 70 Bytes, bevor er auf eine Bestätigung wartet (3). Dies verbraucht das aktuell verwendbare Fenster, da der Client den vollen Betrag gesendet hat, den der Empfänger verarbeiten kann.



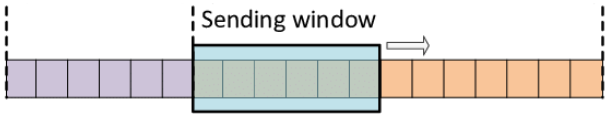
Als nächstes bestätigt der Server die ersten 30 Bytes. Der Absender verschiebt sein Sendefenster um 30 nach rechts und aktualisiert sein verwendbares Fenster, um die bestätigten Bytes zu berücksichtigen (4).



Wenn der Server dann alles in seinem Puffer verarbeitet hat, bestätigt er die nächsten 70 Bytes und aktualisiert sein Empfangsfenster auf 150 Bytes. Der Client vergrößert das Sendefenster und das verwendbare Fenster entsprechend und schiebt beide nach rechts. Der Client überträgt dann weitere 100 Bytes (5).



Quelle: https://www.extrahop.com/company/blog/2017/tcp-windowing/



Dieser Vorgang wird fortgesetzt, bis alle Bytes des Absenders übertragen wurden

In der Realität werden Datenblöcke, die als Segmente bezeichnet werden und jeweils einen TCP-Header enthalten, zwischen einem Sender und einem Empfänger übertragen. Darüber hinaus hängt die Geschwindigkeit, mit der das Sendefenster durch den Puffer des Clients gleitet, davon ab, wie schnell der Client Daten von der sendenden Anwendung erhält, wie schnell der Server die Daten in seinem Empfangspuffer verarbeitet, ob Pakete bei der Übertragung verloren gehen und von den Besonderheiten der Implementierung des „window sliding protocols“ und der Überlastungssteuerungsalgorithmen.

Quelle: https://www.extrahop.com/company/blog/2017/tcp-windowing/