Chapter 6(Module F): Transport of Applications and Traffic Management Techniques

In the previous chapter, we have seen how routing supports packet delivery between the two arbitrary hosts across the Internet. In this chapter, we will take a step further and study the data delivery between two application processes across an internetworking environment. First, we will investigate the major problems that we may face in order to achieve this task. Based on these issues, we will formulate a list of features that are necessary for process-to-process data transfer.

To see how these features are materialized, we will learn the two major protocols of TCP/IP, namely Transmission Control Protocol and User Datagram Protocol. Through the implementations and operations of these protocols, how different operations are conducted will be illustrated.

Furthermore, we will examine the issue of traffic management to see how multiple traffic flows can be regulated in a common network. To this end, we will study traffic policing and shaping, the two commonly used mechanisms for traffic management, and the two popular techniques to implement these regulations, namely the leaky bucket and token bucket.

6.1 Transporting data between application processes

With the internetworking functions in the previous chapter, data can be delivered end-to-end from one device to any other device in the Internet as illustrated in Figure 1. However, these functionalities are not enough for communications between application processes. For instance, there may exist many connections to a single server from different hosts at the same time and even one host may have simultaneous data sessions with one server. As far as the internetworking concerns, it does not differentiate one application process from another and hence can not support multiple parallel data sessions at the same time.

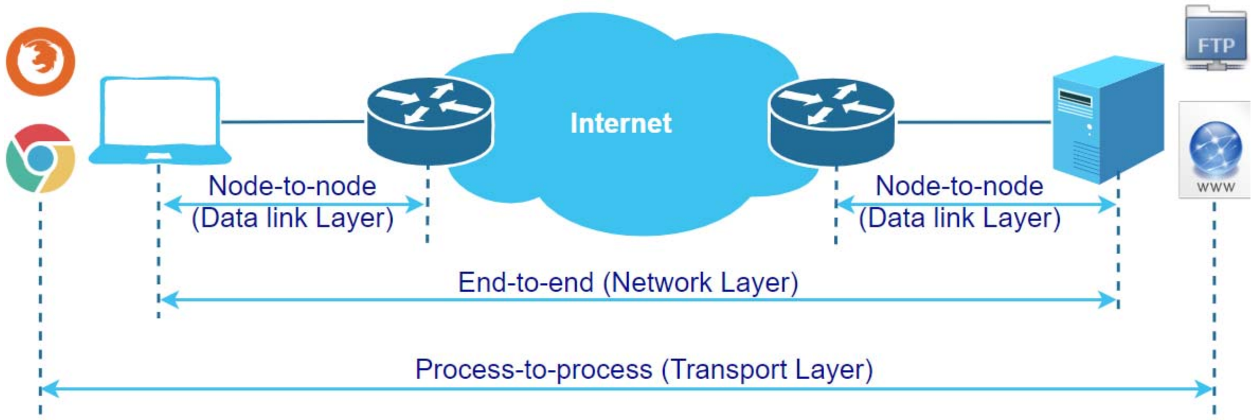

Figure 1: Process-to-process data transfer.

Moreover, it is demonstrated in the previous chapter that Internet Protocol is a connectionless, or a best-effort protocol. This means that IP will try its best to transfer data through the internetworking environment, but it does not guarantee the correctness and the delivery of these data packets. For example, packets can be lost on the way, can be duplicated, and received in errors or in a different order as they were sent. To this end, one can argue that the link-layer protocols is already capable of guaranteeing the packet delivery in each link along the path so that the transmission imperfections mentioned above are not relevant anymore. However, in practice, it is hardly true. Even each link along the path are ideal links without any errors and the related protocols on these links can guarantee in-order, error-free transmission, it does not guarantee the correctness and delivery of data end-to-end. For example, on the way to their destination, user packets can take different paths and hence, arrive in different order as they were sent. In addition, in spite of the error control and retransmissions at the link level, packets can get lost on the way due to congested output queues as discussed in the previous chapter. In such case, there is nothing that the reliable link-layer protocol can do to recover these dropped packets. As a result, there is a need for another layer on top of the network layer to regulate the data transfer at the process to process level, to hide the unreliability of the underlying internetwork communications and to provide application processes with a service level they desire.

Figure 1 illustrates the relationship and the scope between the process-to-process, end-to-end and node-to-node data transfer services at the transport, network and data link layer, respectively. The set of services that is required for process-to-process data transfer may include the following features:

- **Application process multiplexing and de-multiplexing:** within a host, network communications may be shared among multiple processes and among these multiple data sessions, some of them may connect to the same server. As a result, there must be a facility to discriminate the data to and from each process. For example, one can open multiple browsers, each googling a different keyword and expect the return results to be displayed correctly on the appropriate browser.
- **Data reordering:** when packets are travelling through the Internet, they may take different paths to their destination. Due to the difference in network conditions, these packets may not arrive in the order as they were sent, which makes the resembling of data difficult. Consequently, it is important for an application process to abstract this imperfection so that it can consider the data transmission over an internetwork as if the two communicating hosts are directly connected via a virtual pipe where data enter on end of the pipe in-order and pop out of the other end also in-order.
- **Connection-oriented data communication:** in the previous chapter, it was shown that the service offered by IP is connectionless. This type of service may be efficient in terms of reducing overhead but may not be desirable from the perspective of the application processes since there is no way to synchronize between the communication hosts and no feedback channel to support advanced features beyond the basic packet transfer. In connection-oriented data communication, a virtual connection is established between the two communicating hosts before data are sent. This virtual connection is then maintained and used for data transfer. Finally, when the communication session finishes, it will be released. From the perspective of the communication process, this virtual connection transforms the data transmission over multiple links into an ideal environment where the communicating processes directly speak to each other. The existence of such virtual connection allows the two communicating hosts to inform each other about the coming data transfer, to exchange and adjust communicating parameters before and during the communication. For example, in the event of network disruption during the data transfer, the connection can be re-established and continue from where it was halted. In addition, it also provides the facilities to support additional features such as reliable data transfer as described follows.
- **Reliable data transfer:** in an internetworking environment, many imperfections may occur, making the end-to-end transmission erroneous. Since data integrity is one of the most important characteristics of communications, it is desirable for an application process to have an already supported reliable service for transferring its data.
- **Timely delivery:** with the explosion of real-time application such as IP telephony, teleconferencing, Internet gaming and live video streaming, the basic data delivery service is not enough even though no error occurs. For real-time applications, each packet may have a constraint on its lifetime, and it is not useful to deliver a packet when its lifetime expires. Consequently, the application process may demand for a data communication service that can support the delivery of its data within a predefined time constraint.
- **Flow control:** as mentioned in chapter 3, flow control concerns with the regulation of the data transfer speed between the two communication hosts. One may argue that if a flow control mechanism is implemented at the link layer, why it is needed at the end-to-end communication. The truth is that flow control on the basis of individual links is not

equivalent to the *end-to-end* flow control. For instance, due to the capacity differences, the maximum supported bit rate on the links at the two communicating hosts may be different from each other. If the link connected to the source host has a higher bit rate than that of the receiving host, the sending host could overwhelm its receiver, causing packet loss and inefficiency. As a result, end-to-end flow control is an important functionality for data transport between application processes.

- **Congestion control:** Internet is a collection of networks that enables the data transfer between any pairs of hosts connected to it. As such, congestion control concerns the regulation of the sender traffic to prevent it from selfishly using the network for itself by pumping as much data as possible into the network and overloading the network. Different from flow control, which involves only the two communication hosts in each communication session, the congestion control is related to the cooperation between the sending hosts and the network to adaptively adjust the sending rates accordingly to the instantaneous network condition.

The above list is by no mean an exhaustive list, but it illustrates the most common and important services an application may need. Besides, not all of the functionalities mentioned above are crucial to be implemented in all cases. For instance, for a telemetry application which updates sensing readings every five minutes, reliability, and connection-oriented may not needed because missing a few readings can be compensated by the frequent updates. Flow control may not be relevant as well because the data content of each update is small anyway. In this case, these features can be traded off for the simplicity, lower networking and processing overhead, and power efficient for the sensor nodes.

In addition, the implementation of these features may not be straightforward as they are materialized only in end systems and not in network devices such as routers. On one hand, this approach pushes the complexity to the edge of the network, making networking devices such as routers and switches cheaper. Hence, this approach makes deploying and expanding networks easier and more cost effective while still support adequate functionalities to the applications. On the other hand, since the end devices have no control over the routers, they must be able to sense the network conditions and adjust themselves accordingly to the response of the network and the feedbacks of the host on the other end of the communication session.

In this chapter, we will look at the materialization of these functionalities in commonly used transport-layer protocols within the TCP/IP protocol stack. We will start our discussion with the simple User Datagram Protocol and then move to the more sophisticated Transmission Control Protocol where most of the above functionalities were implemented.

6.2 User Datagram Protocol (UDP)

At the transport layer, TCP supports transmission of segments using two major protocols, namely **User Datagram Protocol (UDP)** and **Transmission Control Protocol (TCP)**. Among the two, UDP is the simplest and easiest to understand, and hence, will provide a good preamble before we jump into the more complicated TCP. UDP is a connectionless protocol and from the lengthy list of features above, UDP supports only two functions: multiplexing and de-multiplexing of application processes and error detection. At first, one may argue that it does not make sense to have another protocol on top of IP but barely support any additional functionalities. However, UDP makes its ways to applications due to its simplicity. The followings are some of the benefits that UDP provides.

- **Light weight:** as UDP is a connectionless protocol, there is no connection setup, maintenance and teardown. Consequently, UDP requires very minimal amount of resources for communications. For instance, as we shall see, in a connection-oriented protocol such as TCP, each communicating host needs to maintain the connection states, which is unsuitable for bare bone devices such as small IoT devices and computers without a full protocol stack. An example of the latter case is the acquirement of boot files over a network. In such scenario, the computer needs to retrieve its operating system through a network connection using just a minimal protocol stack. Another example about the use of UDP is the DHCP protocol that we studied in the previous chapter. In this case, the newly participated host just wants to get its IP address as quickly as possible and without connection setup, maintenance and teardown, UDP can help to speed up this process significantly. Moreover, without a workable IP address, the host cannot establish a connection to the DHCP server anyway.

- **Small header overhead:** due to its limited set of features, UDP requires only a very compact header for supporting communications (only 8 bytes long). This is a very appealing characteristic for transaction-based communication such as sending telemetry readings from IoT devices, which just need to wakeup for a short time, send their sensing values and then go back to sleep mode. This characteristic is also very beneficial for simple query-based protocols such as Domain Name System as it reduces the processing and networking overhead and significantly simplify the implementation. For such protocol, if the source host does not get any reply after a certain period, it simply resends the query again.

- **Application-based communication management:** the feature-poor characteristic of UDP put the burden of keeping track and controlling the communications on the shoulder of the application. However, its simplicity allows a perhaps unexpected benefit that it allows the application to have the full control on how it wants the communication session to be played out. This control makes it possible for developers to implement new communication features into their applications without depending on the existing supported features of TCP/IP. For instance, if a developer aims for an application that can tolerate a certain amount of packet loss but is very keen on the arrival order and delay, he can build his own communication protocols based on UDP. A prime example of this customization on top of UDP is the realization of real time communications in Real-Time Transport Protocol (RTP) [1] to support Internet Telephony and video conferences. By having direct control over the sending and receiving of messages, RTP can reduce the delay due to connection establishment and can monitor the packet delivery, delay and jitter in order to adapt its media encoding accordingly to the network variations.

6.2.1 UDP header format

Figure 2 illustrates the header format of a UDP segment. It can be seen in this figure that an UDP header format is very simple, only 8 bytes long and consists of only 4 fields as the follows:

- **Source and Destination Port (16 bits each):** these fields provide the identities of the application processes at the source and destination hosts and are used for the multiplexing and de-multiplexing of application process at the two ends, which will be described latter.

- **Length (16 bits):** this field contains the length in bytes of the UDP segment including both its header and payload.
- **Checksum (16 bits):** this field is used for error checking of the received segment. The procedure for calculating of this field is based on the Internet Checksum already presented in chapter 3. However, the specific information that is used for checksum calculation is a bit involved and will be elaborated briefly next.

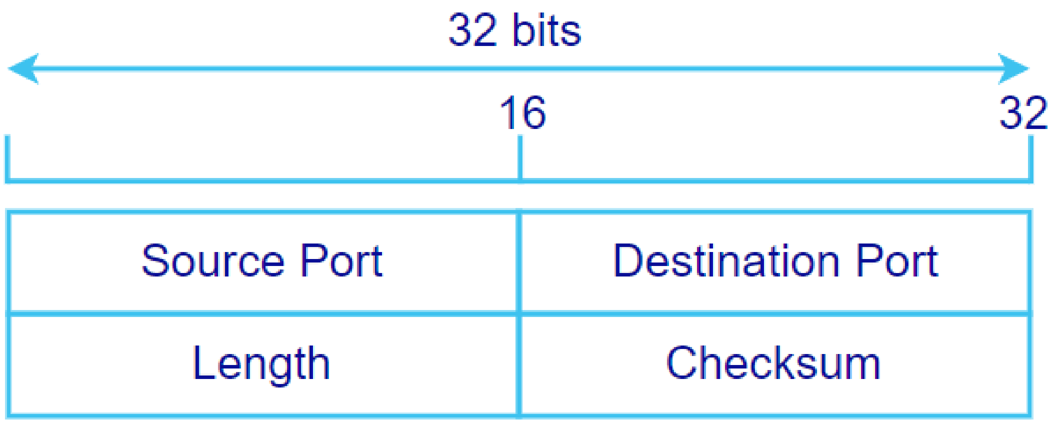

Figure 2: UDP header format.

Checksum calculation

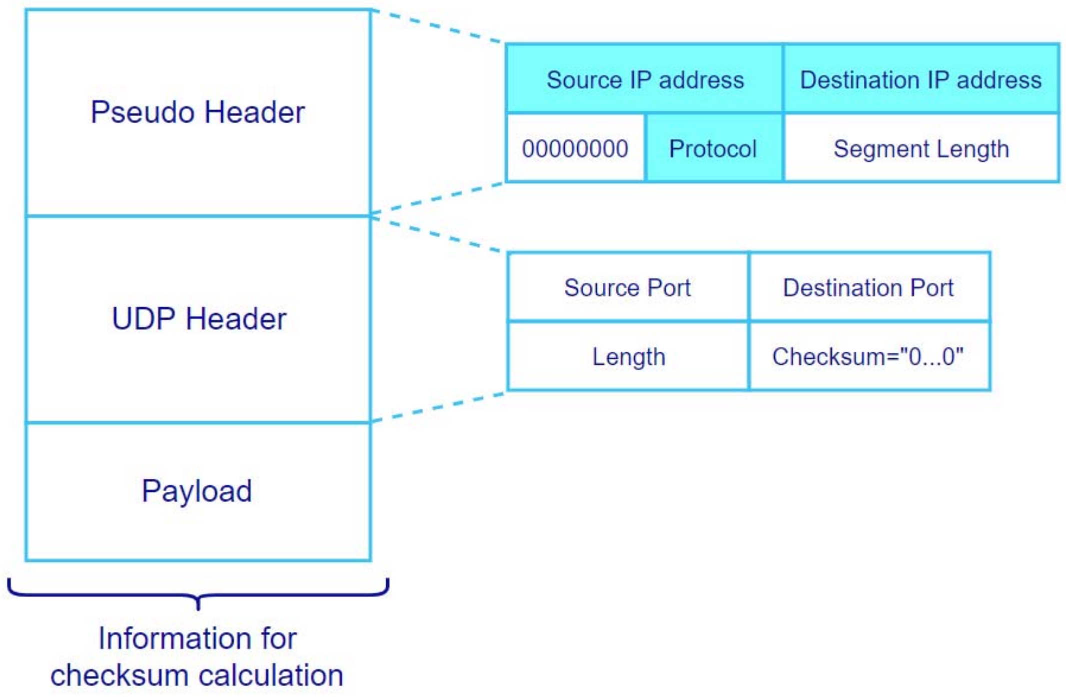

Figure 3: Information used for checksum calculation with an IPv4 packet.

To calculate checksum for an UDP segment, the Internet checksum algorithm is executed on not only the UDP header and payload but also some information from the IP header as well. In the case of IPv4, as illustrated in Figure 3, the information used for checksum calculation includes the payload, the UDP header (where the checksum field is set to all zeros) and a **pseudo header**. This pseudo header includes the source and destination IP addresses, the Protocol field from the IP header, the segment length which is the same as the Length field in the segment header and a padding of 8 bits 0s to make the pseudo header well ended in a multiple of 16 bits. If the payload is not a multiple of 16 bits, it will be padded with zeros to the necessary length (these padding bits will be removed after the checksum calculation).

The segment header, payload and the pseudo header are used to calculate the checksum value, which will replace the dummy zero checksum that used during the calculation. At the receiver, a pseudo header is constructed with the destination IP address as the receiving host's address. The new pseudo header is then concatenated with the segment for Internet checksum calculation. If the output of this calculation is all zeros, the receiving host can be assured that the received segment is not corrupted.

In this calculation, it is noticed that the source and destination IP addresses are also involved which raise the question of the reason behind the use of the pseudo header at all. The answer is that these are used for verifying that the segment is indeed arrived at the intended destination. If somehow the segment is delivered to a wrong destination, the pseudo header would fail the checksum and hence, the segment will be discarded. Similarly, if somehow the packet IP addresses are changed, the checksum would be able to help detecting this misbehaviour.

6.2.2 Application Multiplexing and De-Multiplexing

Today, it is normal for a computer to multitask, for instance, you can listen to music at the same time of surfing a few web pages while waiting for your downloading files to finish. With so many data messages going back and forward, it is important that there is a way for the source node to mark which process it wants to communicate with at the receiver so that the receiver can identify which segment is destined for each of its application processes.

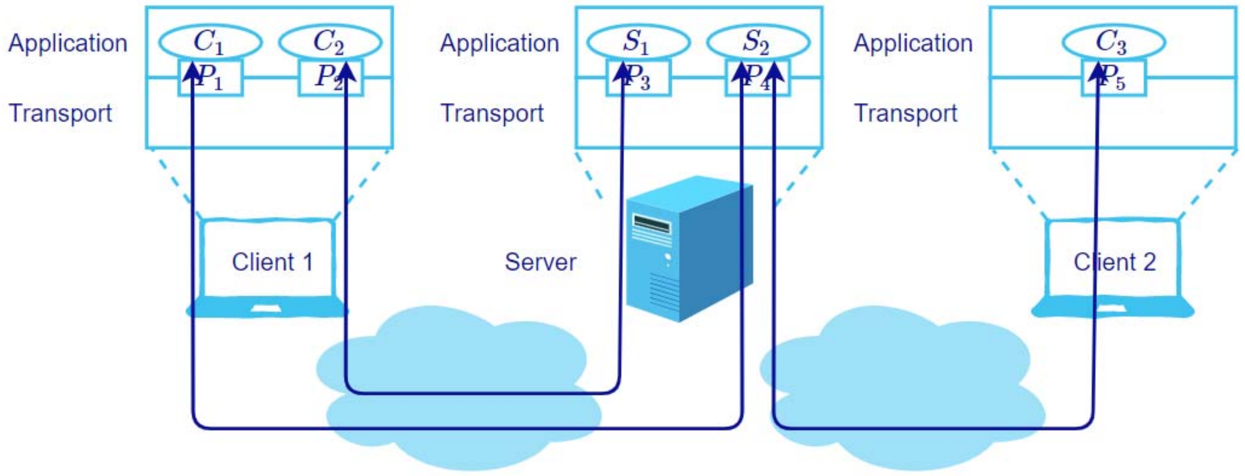

Figure 4: Application process multiplexing and de-multiplexing.

In TCP/IP, within a host, each network application process is specified by an identifier at the transport layer namely a port. A **port** is basically an abstract resource that is assigned temporary to an application process for network communication. Since application processes must be uniquely identified, ports also have to be uniquely within a host. As an IP address help to uniquely identify a host on the Internet, the combination of IP address and port makes it possible to uniquely determine any network application process in the Internet. As a result of this uniqueness, this combination of an IP address and a port number is also referred to as a **socket**. While IP addresses needs to be globally uniquely different from each other, it is not needed for port numbers to be globally unique since the socket's IP address already helps to separate one host from another. As an illustrative example, the IP addresses can be equivalent with the addresses of houses while port number is equivalent to the identifier of a room within the house.

Table 6.1: Well-known ports and their associated applications.

Port number	UDP/TCP	Application Protocol	Functionalities
20, 21	TCP	FTP	File transfer
22	TCP	SSH	Secure remote control and file transfer
25	TCP	SMTP	Electronic mail
53	UDP	DNS	Domain name system
67, 68	UDP	DHCP	Host configuration
80	TCP	HTTP	Web service
123	UDP	NTP	Time synchronization service
161	UDP	SNMP	Network management
179	TCP	BGP	Inter-AS routing

Figure 4 illustrates the multiplexing and de-multiplexing of application processes at the transport layer. At transport layer, a locally unique port number (such as P_1) is assigned to an application process (in many cases, an application process can attach to multiple ports). The port number can be considered as an interface through which the application process sends and receives information to and from the network. In a way, this concept is analogous to the network interface where data come and go from a host. Using a combination of an IP address and a port number, a host can locate and communicate to any other application processes in the vast Internet. The assembled segments from the application processes in each host are them transferred to the network layer where they are multiplexed together and share the same communication resource. At the destination host, the destination port is used to de-multiplex the received data, i.e., to identify and deliver data to the destined destination application process from all the concurrent processes. As depicted in this figure, an application process, such as S_2 can communicate with multiple other application processes (C_2 and C_3) at the same time. In this case, the application process S_2 makes use or the source socket (source IP address and port number) to separate the data from the sent processes.

TCP/IP uses 16 bits for port numbering, which translates into about 64K possible ports or concurrent application processes within a host. The port numbers less than 1024 are called **well-known ports** and are reserved for commonly used applications. A list of some well-known ports and their associated applications is described in Table 6.1 for both UDP and TCP. Typically, to realize a network service, a server opens the appropriate ports in this well-known port range and uses its application process to listen for incoming requests. To communicate with an application process on a server, a client opens a port higher than 1024 and uses this port to initiate requests to a supported server (at a well-known destination port) for a service that it desires. To send a reply to this request, the server can use the source socket to locate the exact application process at the client. If the server does not use the standard well-known ports for its services (possibly due to security purposes), the client must be configured with the correct port for the communication to be successful. After the application processes at either the client or server finish their job, the appropriate port is released and can be reused by another application process at a latter time.

6.3 Transmission Control Protocol (TCP)

In contrast to the basic services supported by UDP, its big brother TCP came with a wide range of offerings including connection-oriented data transfer, reliable and in-order data delivery, flow control and congestion control beside the basic multiplexing and de-multiplexing of application processes. In addition, TCP supports full duplex data transmission on a single connection, i.e. it allows two-way data transmission between a pair of hosts simultaneously. With these feature-rich offerings, it is not surprise to know that TCP is widely used in many applications and contributes to the majority of the representative well-known services in Table 6.1 as well as those defined by Internet Assigned Numbers Authority (IANA) [2].

6.3.1 TCP header format

Being fitted with so many features, the header structure of a TCP segment is significantly more complex and longer than the previously mentioned UDP header structure. As illustrated in Figure 5, a TCP header includes 10 mandatory fields and is at least 20 bytes long, which introduces 2.5 times more overhead into a transmitted segment than UDP (8 bytes header). Perhaps, to this point, it is clear why UDP is usually preferred in applications with small transactions. The explanations of the TCP header fields are briefly described as the follows.

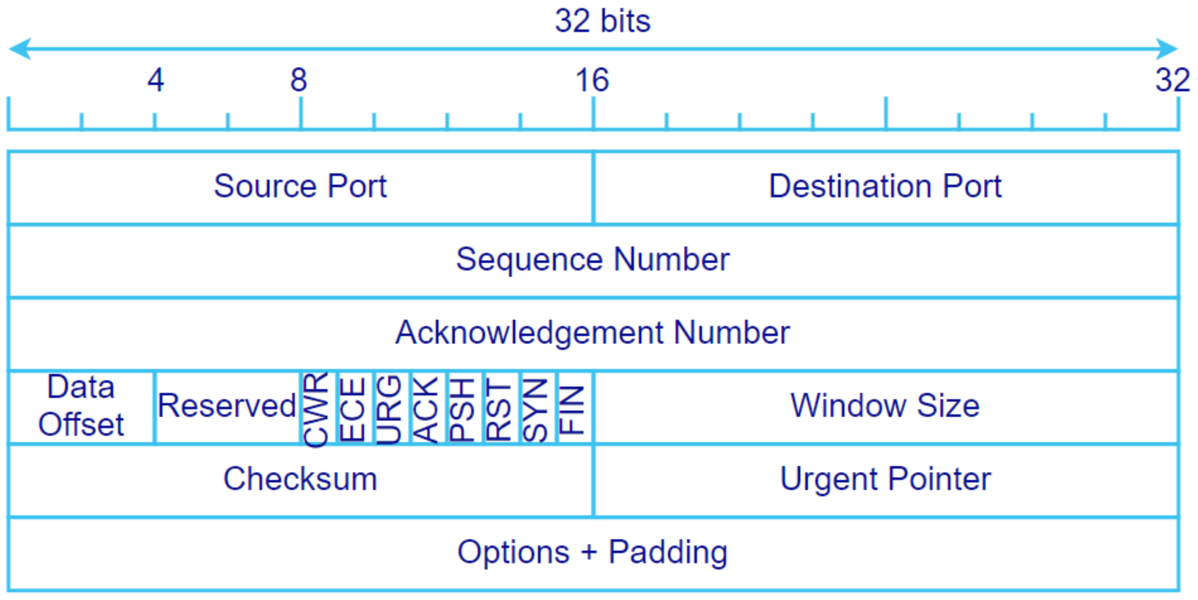

Figure 5: TCP header format.

- **Source and Destination Port (16 bits each):** serving the same functionality as those in the UDP header, these fields provide the identities of the application processes at the source and destination hosts, and are used for the multiplexing and de-multiplexing of application process at the two communication ends.
- **Sequence Number (32 bits):** this field contains the order of the first by in the current segment with regarding to the overall data stream. In the connection setup, when the SYN flag is set, this field indicates the initial sequence number the source node chooses for this connection. Together with the acknowledgement, receive window and the flag fields, this field is used to implement reliable data transfer, re-ordering of received data and flow control according to ARQ mechanism as will be described latter.
- **Acknowledgement Number (32 bits):** if the ACK flag is set, this field contains the sequence number of the next byte that the sender of this segment expects. This field is used to acknowledge all data prior bytes to this number.
- **Data Offset (4 bits):** this field specifies the length of the TCP header. Beside the 10 mandatory fields, a TCP segment header can contain options which make the start of the data unclear. This field helps to identify the first byte of the data payload (hence its name). In many other references, this field is also referred to as Header Length.
- **Reserved (4bits):** the functionality of the bits in this field is not defined. These bits are reserved for future use and must set to zero.
- **Flags (8 bits):** each bit of this field is a flag with a specific meaning. Only the last 6 flags are discussed in this chapter, the first two flags are not directly relevant to the functionalities mentioned above and will not be presented.
    - **URG (Urgent):** this flag indicates that the segment contains urgent data that need to be processed by the destination application process. The urgent data will be placed at the beginning of the payload. When this flag is used, the Urgent Pointer field points to the first byte of the non-urgent data, i.e. the end of the urgent data.
    - **ACK (Acknowledgement):** this flag indicates that the current segment contains acknowledgement information which is specified in the Acknowledgement Number field.
    - **PSH (Push):** this flag suggests that the destination application process should send the buffered information for this connection as soon as possible. In TCP, the application process writes data bytes into a send buffer, but the TCP endpoints may want to wait until the buffer is large enough rather than sending data as soon as they arrive at the send buffer. With push function, the sending process can advise the receiver process to send out its buffered information the soonest without waiting any further.
    - **RST (Reset):** this flag helps to reset the TCP connection when some errors are encountered.
    - **SYN (Synchronization):** this flag is used in the connection setup procedure to synchronize the initial connection parameters.
    - **FIN (Final):** this flag is used for connection teardown.
- **Window size (16 bits):** this field is used to inform the receiver node about the available buffer size for communication at the sender of this segment. Consequently, it indicates the amount of data that the sender of this segment is willing to accept for future

segments and is used for flow control in the reverse direction. This field is also referred to as Advertised Window.

- **Checksum (16 bits):** this field is used for detecting errors that occurs in the communication session between the two ends. TCP checksum is calculated the same way as UDP checksum, i.e. involving the segment header, its payload and the appropriate pseudo header.
- **Urgent Pointer (16 bits):** this field is interpreted by the receiver if the URG flag is set and it indicates the end of the urgent data in the segment payload.
- **Options and Padding (variable, must be multiple of 32 bits):** this field is used to append further functionalities into TCP. If the added options are not a multiple of 32 bits, padding bits will be added to satisfy this requirement. For example, options can be added in the connection establishment to inform the other end about the **maximum segment size (MSS)** that the sender of the segment is willing to accept. In addition, options can also be added to extend the window size.

6.3.2 Connection-oriented Data Communication

Different from UDP, TCP is a connection-oriented protocol; consequently, before transmitting any data segment, the two TCP endpoints must establish a connection, and after data transmission finishes, they must terminate this connection. This connection is a virtual connection between the two endpoints rather than a physical one and is the basis for helping the application process to hide the imperfections of the underlying network.

In order to maintain this connection, TCP defines several states and the transitions between states that a TCP endpoint must follow during its lifetime. A simplified state diagram of TCP state transition is illustrated in Figure 6. In this diagram, the different states in TCP are enclosed in rectangular boxes and the arrows illustrates the transitions between the states. The notation beside each arrow denotes the event that cause the transition and the action the TCP endpoint would take in response; this notation is expressed the form of $\frac{Event}{Action}$.

The events that cause the state transition can be classified into two types: the receipt of a segment from the other TCP end point and the commands from the local application process. Sometimes, the TCP endpoint does not need to execute any action in response, then, only the event is expressed in this notation. For example, at the beginning of the state diagram, from CLOSE state, a local application process can invoke an active open to a sever and send a SYN packet, then, it changes its state from CLOSE to SYN_SENT.

At the first look, this state diagram seems to be dauntingly complicated; however, upon diving a bit further, understanding this state diagram is not that hard. In particular, the state diagram can be divided into three main parts for the ease of understanding: connection establishment, data transfer and connection termination. It is noted that once the connection is established, the data transferring details are captured inside the ESTABLISHED state and hence will not be discussed in this subsection. Instead, this data transfer details will be studied in the next sections which will elaborate details about the data sequencing, acknowledgement, and flow control.

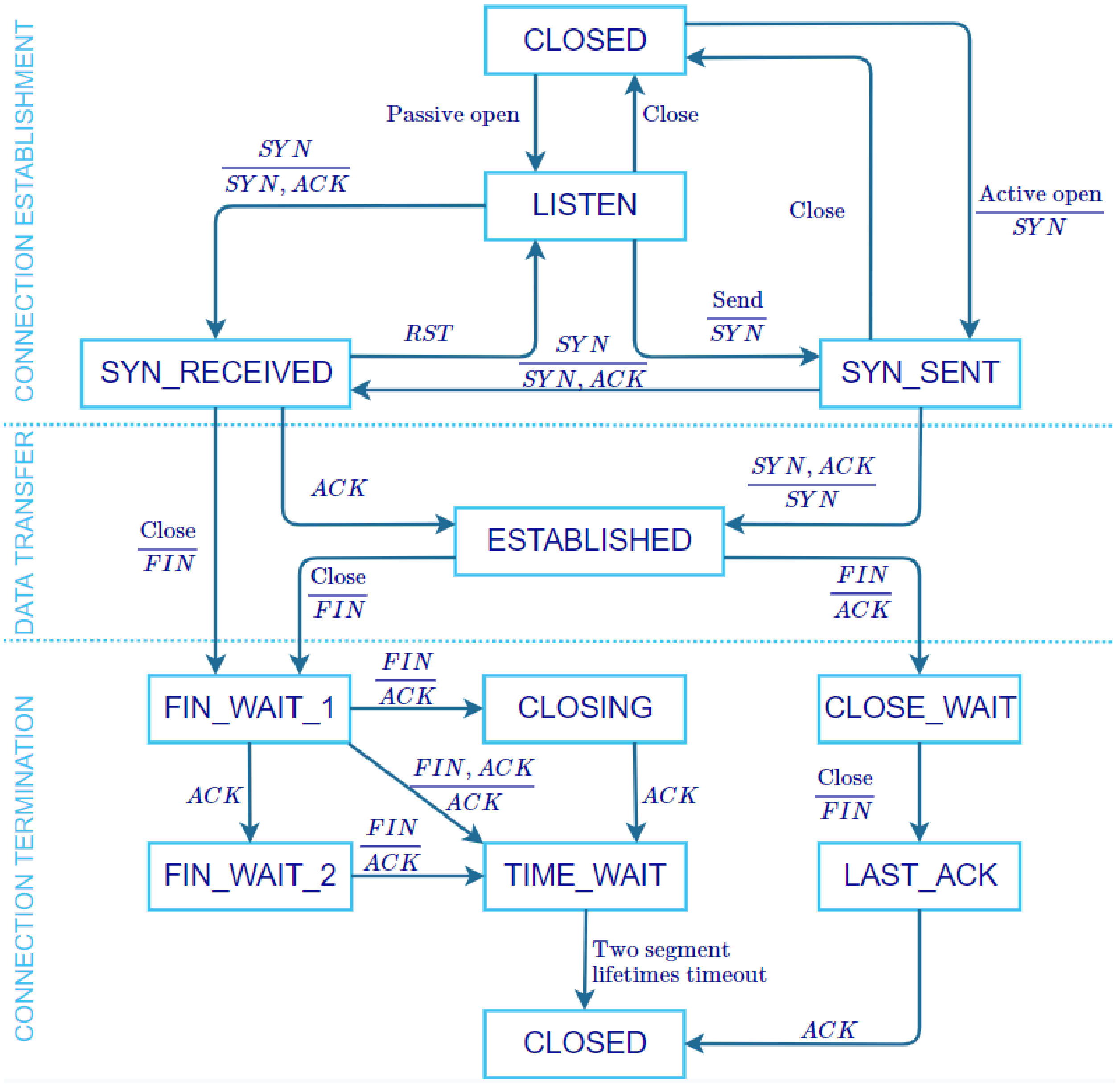

Figure 6: Simplified state transition diagram in TCP.

Connection setup and three-way-handshake

To support a TCP connection, we first need a server that is waiting for connections. Tracing the state transition in Figure 6, from the CLOSED state, the server executes a passive open command and enters LISTEN state where, as its name indicated, the server keeps listening for incoming connections. In TCP, connection setup is achieved using a **three-way-handshake** procedure as illustrated in Figure 7. The motivation behind this procedure is for the two TCP endpoints to agree and synchronize on the connection parameters to ensure the reliability of the data transfer that occurs next. The three-way-handshake includes three steps as follows.

- **Step 1:** the client invokes an active open by sending a segment to the server requesting for opening a new connection. This segment is identified with a SYN flag sets to 1 and a randomly pick a Sequence Number x and uses it as the initial Sequence Number for this connection. Tracing the TCP state diagram, this transition moves the client from CLOSED state to SYN-SENT state.

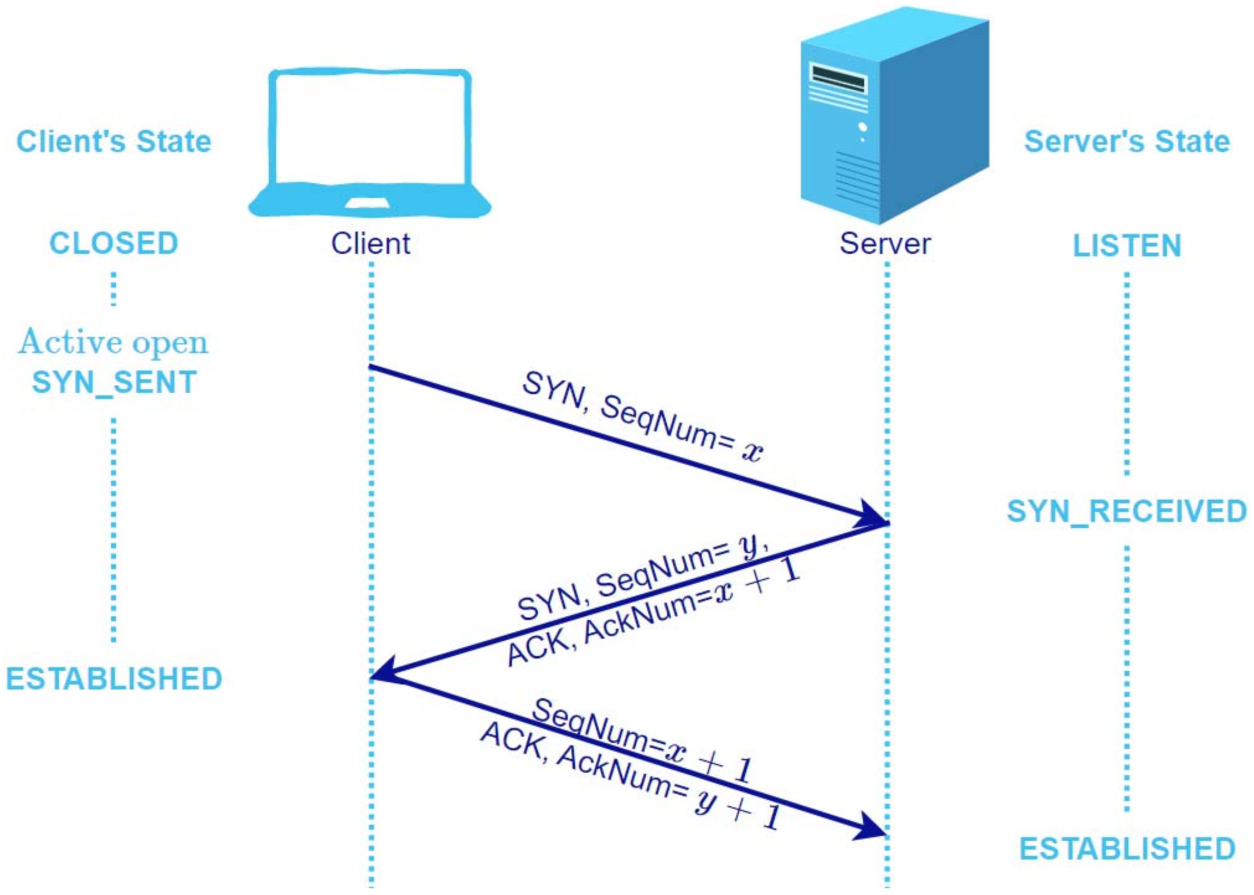

Figure 7: TCP three-way-handshake for connection setup.

- **Step 2:** when the server receives the above segment from the client, it transits from LISTEN to SYN_RECEIVED state and emits a segment with SYN flag sets to 1 accompanying with a randomly pick Sequence Number y. To acknowledge the previous segment sent by the client, the server also sets the ACK flag to 1 and specifies its next expected Sequence Number from the client in the Acknowledgement field as $x + 1$. The "$+1$" is used to confirm the receipt of the initial SYN packet from the client. From now on, the server is aware of and can keep track of the sequence numbering from the client.
- **Step 3**: upon receiving the server's reply from step 2, the client reacts with an acknowledgement segment to the server, confirming this receipt by setting ACK flag to 1 and filling the Acknowledgement field with $y + 1$. After sending this segment, the client moves from SYN_SENT to ESTABLISHED state and is ready for data transmission.

When receiving the ACK packet in step 3 from the client, the server also moves to ESTABLISHED state from SYN_RECEIVED and is ready for data transmission. One may ask why the two endpoints need to pick a random Sequence Number in step 1 and 2 instead of using a pre-defined starting Sequence Number such as 0. To answer this question, recall that a port is assigned to an application process temporary for communication purpose and it will be released back to the transport layer when the application process finishes its data transfer. Then, the port can be re-used again for another application process. As such, it is important for the new application process to have a random initial Sequence Number to prevent the case that a segment from the previous connection (on the same port, to the same destination) is delayed and would be incorrectly accepted in the data transfer phase of the new connection.

Connection Teardown

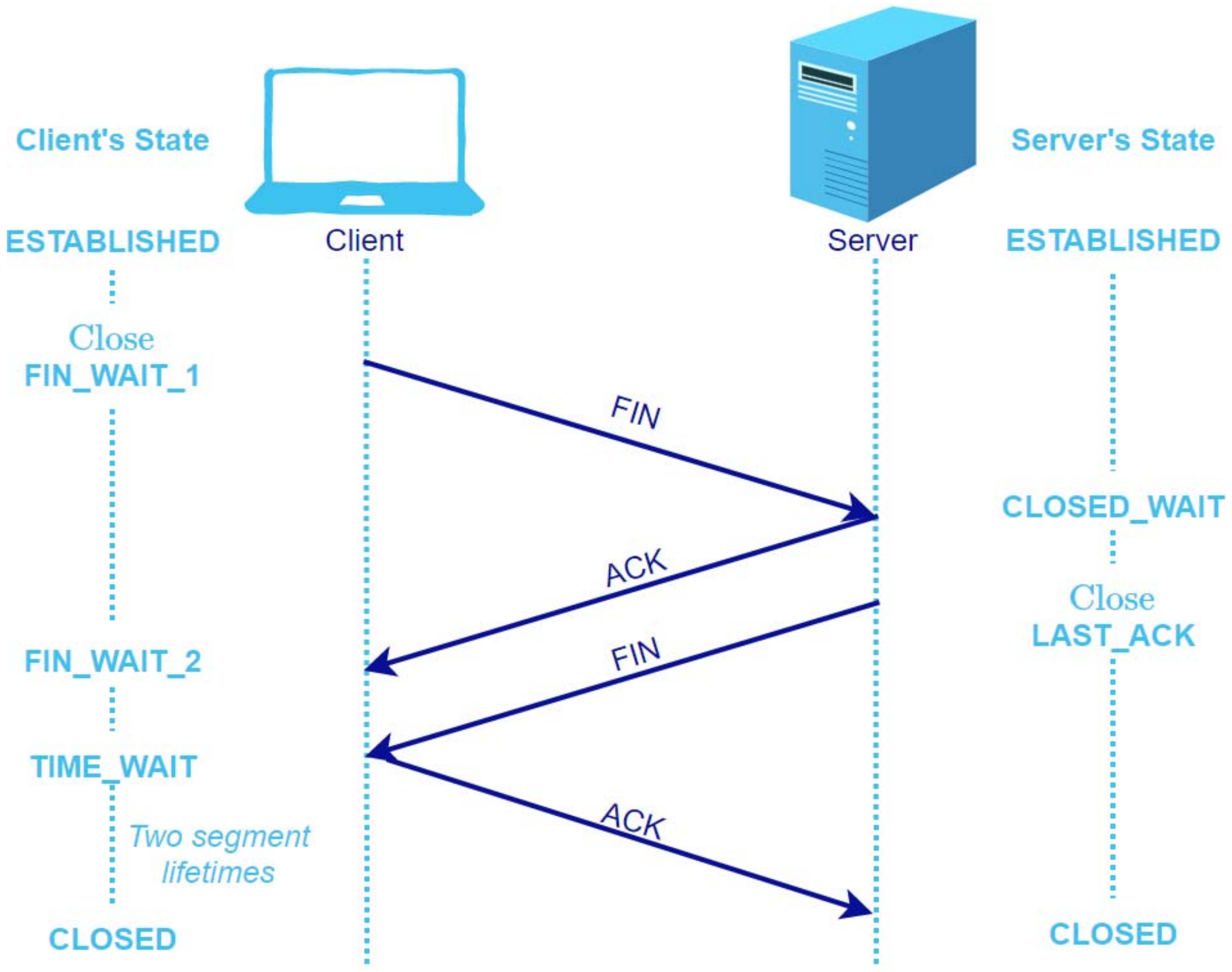

Figure 8: Connection termination in TCP.

Since a connection in TCP is setup from both the client and server, terminating a connection requires a grateful cease from the two sides. Figure 8 illustrates a typical termination procedure in TCP. Both TCP endpoints start with ESTABLISHED state where data transfer happens. In this case, the client initiates the connection termination by sending a segment with the FIN flag set to 1, its state will then move to FIN_WAIT_1. Upon receiving this segment, the server acknowledges this willing to terminate the half connection from the client by sending a segment with ACK flag set to 1 and move to CLOSED_WAIT state. When the client receives this segment, it knows that its half of the connection has been successfully closed. When the client receives this ACK segment, it moves to FIN_WAIT_2 state and waits for the termination from the server.

It is important to know that at this point, the half of the connection from the server is still open and data can still be sent from the server to the client. When the server wants to close is half of the connection, it sends a segment with the FIN flag set to 1 and move its state to LAST_ACK where essentially it waits for the last acknowledgement segment of the connection. When the client receives this segment, it sends its response with the ACK flag set to 1 and move to TIME_WAIT state. In this state the client waits for two segment lifetimes before it closes the connection. When the server receives the acknowledgement from the client, it moves its state to CLOSED and closes the connection.

In this connection tear down process, the transition from TIME_WAIT to CLOSED on the client side requires a bit of further explanation. One may ask should the client just change the state of the connection into CLOSED right after sending its ACK? Well, it can do so but this action is not foolproof and can cause an unexpected error as explaining below. Assuming that the last ACK from the client is lost in the network. In this case, the server would assume that its FIN message was lost and will resend this segment. This FIN segment could be delayed in the network and upon its arrival at the client, it would terminate a latter connection that was initiated latter on the same port. As a result, a delay of two segment lifetimes is a preventive measure for this exception. The first segment lifetime corresponds to the one-way delay of sending the ACK from the client to the server. The second segment lifetime corresponds to the one-way delay of the resending of the FIN message from the server in case the ACK was lost. In combination, the two segment lifetimes guarantee that all segments from the ceasing connection are no longer available in the network and no confusion would be possible.

6.3.3 TCP Reliable Data Transfer and Data Reordering

To support reliable data transfer and data reordering service, TCP uses a variation of Selective Repeat ARQ algorithm with sliding window that we already studied in chapter 3. There are two differences though. First, the window used in TCP and its sequence number denote the byte order of the segment within the data stream and not the segment order as previously mentioned. For instance, when a TCP transmitter sends two consecutive segments, the sequence number of the second segment equals to the sequence number of the first segment adding with the length of the data within the first segment. With this sequence numbering, the received segments can be re-ordering even if they do not arrive in order. Consequently, sequence number in the ACK segments denotes the next data byte that the receiver is expecting. Second, in TCP, the ACK segments have the cumulative acknowledgement and behave similar to Go-Back-N, i.e. an ACK segment will cumulatively acknowledge all data bytes up to one byte before the sequence number it carries and the receiver only sends ACKs of up to the highest in-order segments.

Figure 9 illustrates an example of the reliable data transfer scheme in TCP. In this example, the sender sends four consecutive segments, 1000 bytes each. It is noted that the Sequence Number is filled according to the byte order of the payload data and not on the order of the segments. Supposed that the first ACK segment is lost while the second ACK arrives intact. In this case, the second ACK will have a cumulative effect and acknowledges all the data one byte before its sequence number, including the data in the first segment. Supposed that the third segment is lost while the fourth segment arrives safely. Since the fourth segment is out of order, it is still cached at the receiver and the receiver acknowledges again the second segment, i.e. a duplicate ACK as the previous one. In this case, TCP uses these duplicate ACKs as an indication of segment loss. In particular, if the sender receives **three duplicate ACKs**, it will retransmit the lowest order unacknowledged segment. TCP also uses a timeout timer to detect segment loss, the calculation of this timeout timer will be described in the next subsection. Since multiple segments can be sent back-to-back by the transmitter, the use of three duplicate ACKs would result in a quicker loss detection than waiting for a timeout timer. As such, this method of using duplicate ACK to identify segment loss is referred to as **fast retransmit**.

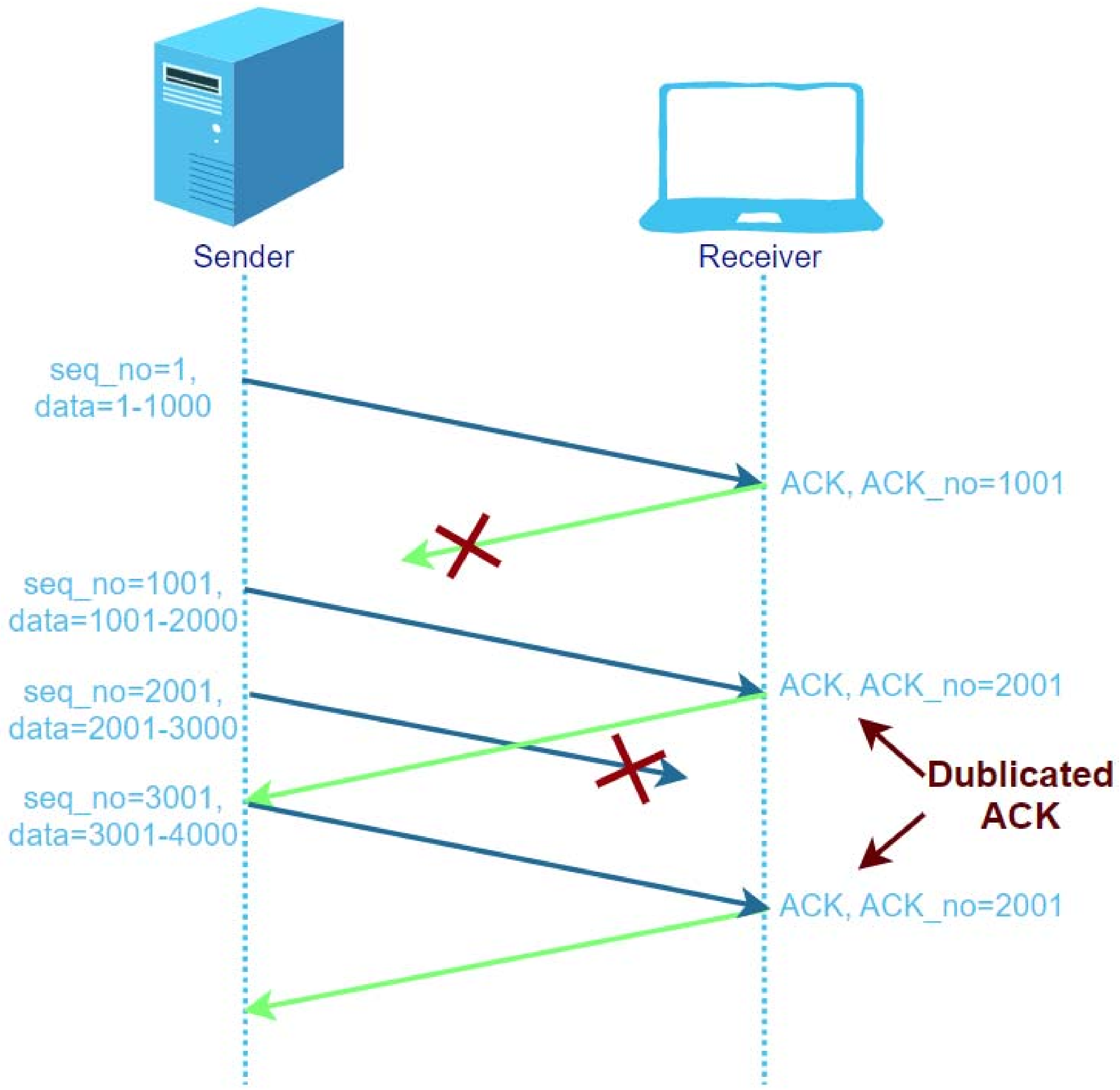

Figure 9: Example of reliable data transfer scheme in TCP.

It can be seen that the reliable data transfer of TCP is more of a hybrid between Selective Repeat and Go-Back-N ARQ. The initial motivation of this implementation was to keep the receiver as simple as possible. However, this implementation makes TCP recovery against lost very slow when there are consecutive lost segments as it will take one timeout timer for each lost segment. As a result, an option was added to TCP namely **selective**

acknowledgement (SACK). When it is used, SACK utilizes the option field of the TCP header to explicitly report the successfully received but not in-order segments. Using this feedback information, the TCP sender can just fill in the gap and the recovery from segment lost will be speedup significantly.

6.3.4 TCP Round Trip Time Calculation

In the previous subsection, we mentioned about the use of timeout timer but did not elaborate on how such parameter is evaluated. From chapter 3 we already knew that timeout depends on the **round trip time (RTT);** hence, we must first know how to determine the RTT in a TCP session. However, different from the RTT in protocols at data link layer, which is normally fixed, A TCP connection spans over a number of links and hence its RTT varies depending on the network conditions. Moreover, within the timespan of a TCP connection, the traffic flows in the network may fluctuate drastically, making the RTT change accordingly. As a result, TCP RTT must be measured continuously during the connection lifetime and adjusted adaptively. In this section, let's denote **EstimatedRTT** the estimated RTT measured from a TCP endpoint.

Let's first discuss how an instantaneous RTT, or a **SampleRTT**, can be measured. An instance of RTT can be measured at a TCP endpoint by setting a timer when the endpoint sends a segment and stopping the timer when the endpoint receives its ACK. This method sounds legit, but it has a flaw, it only works in the absence of segment loss. When a segment is lost, the retransmission of this segment confuses the starting and ending time, making this method of measuring the SampleRTT invalid. However, with a slight modification, we can resolve this issue. If we ignore the retransmitted segments and restrict our measurements on only those segments that are transmitted only once, we can mitigate the mentioned above issue and end up with accurate SampleRTT measurements. As a result, SampleRTT measurement is conducted for all TCP segments but when a segment suffers from retransmission, its SampleRTT measurement ceases.

With SampleRTTs, we can now in the position to compute the Estimated RTT. Since SampleRTT only measures an instance of RTT and may change greatly from one instant to another, a straight forward way to calculate Estimated RTT is by taking a running average of SampleRTT as illustrated in equation (6.1), where $\alpha \leq 1$ is the weight of the running average that governs its dependency on the historical value and the instantaneous value. If α is close to 1, the EstimatedRTT would vary slowly and may slack behind the network changes. On the other hand, if α is close to 0, EstimatedRTT would track the network changes closely but it may fluctuate wildly. A typical value of α is 0.875.

$$EstimatedRTT = \alpha \times EstimatedRTT + (1 - \alpha) \times SampleRTT \tag{6.1}$$

With this running average approach, we can calculate the EstimatedRTT, we are now in the position to calculate the timeout for TCP retransmission. If we set this value too high, TCP would recover from packet loss very slowly. On the other hand, if we set timeout value too low, we would unnecessarily retransmit data segments, wasting communication resources. In chapter 3, timeout was calculated as one round trip time; however, this estimation is only true at the data link level when the transmission between two nodes on the link does not vary much. In the case of TCP, the EstimatedRTT as calculated above should change with time, and using the most recent EstimatedRTT could lead to one of the two extremes mentioned above. A solution for this problem is to incorporate also the variance of the

EstimatedRTT into the timeout. For example, if the EstimatedRTT varies a lot, it is an indication that this value may not be a good indication for RTT. Let's define the deviation of EstimatedRTT as in equation (6.2), where β is a parameter between 0 and 1 (a typical value is $\beta = 0.75$). In this equation, it can be seen that RTTDev is a running average of the difference between the instantaneous and the estimated RTT, hence, it characterises the variation or the devation of EstimatedRTT.

$$RTTDev = \beta \times RTTDev + (1-\beta) \times |SampleRTT - EstimatedRTT| \tag{6.2}$$

Using this deviation of the EstimatedRTT, the TCP timeout can be calculated as the weighted sum between RTTDev and EstimatedRTT as in equation (6.3). The typical values of μ and φ are $\mu = 1$ and $\varphi = 4$.

$$Timeout = \mu \times EstimatedRTT + \varphi \times RTTDev \tag{6.3}$$

Using equation (6.3), it can be seen that when the deviation in EstimatedRTT is small, the TCP timeout is essentially the EstimatedRTT. However, when EstimatedRTT varies a lot, RTTDev is large and hence, it contributes significantly into the calculation of the TCP timeout.

6.3.5 TCP Flow Control

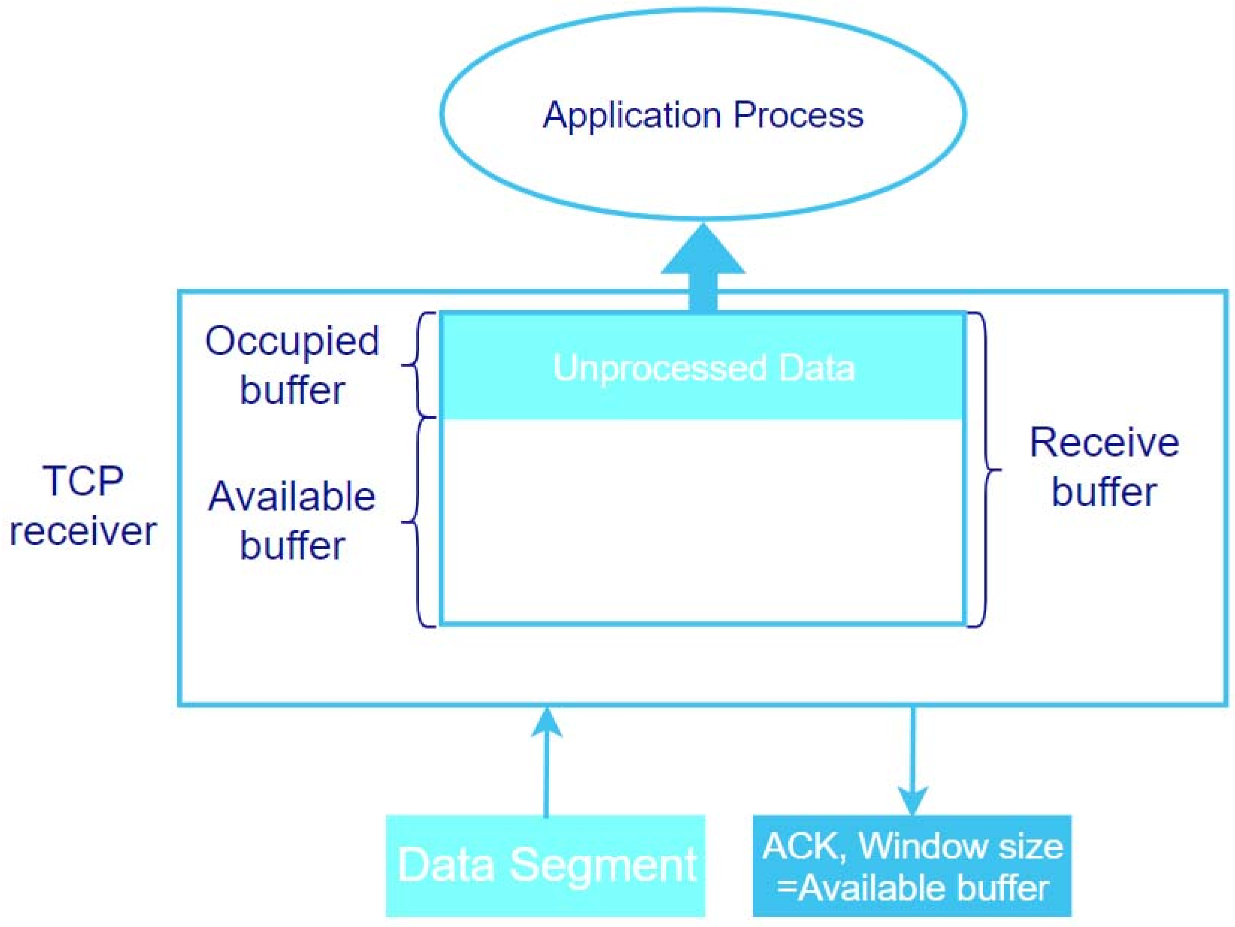

Figure 10: TCP receive buffer and window size.

In TCP, the receiving endpoint is allocated with a receiver buffer that caches the received data segments from the sending endpoint for its connection. The receiver application process will access to this buffer and read data from this buffer at its pace. Depending the type of application process and its priority among other application processes in the receiver, the receiving application process may not read the data immediately as they arrive at the receiver endpoint. As a result, these data segments are stored temporary in this receive buffer as illustrated in Figure 10, waiting for the receiving process to consume. If the reading speed of the receiving application process is slower than the arrival speed of the data segments, the receive buffer will be filled up and then overflowed. Consequently, there is a need for a flow control mechanism that allows the TCP receiver endpoint to feedback about its available buffer so that the sending endpoint can adjust its rate accordingly.

TCP supports flow control by using the Window size field in the TCP segments as a feedback to the sending endpoint. This information is embedded in the segments that travel in the reversed direction from the receiver to the sender such as ACK segments. This field specifies the available buffer at the receiver side so that the sending endpoint can adapt the size of its next segments accordingly. However, this Window size cannot be used directly since at the time the sending endpoint receives this Window size, it may already transmit additional segments which should be also buffered at the receiver. To account for these on-the-fly segments, the TCP sending endpoint computes and maintains a **Usable Window** that yields a constraint on the amount of data it can send to the receiver without overflow its buffer. In particular, let **LastSentByte** be the byte order of the last byte that the sender already sent and **LastACKByte** be the latest byte that the sender already received in the acknowledgement from the receiver, the sender Usable Window can be calculated as in equation (6.4).

$$Usable\ Window = Window\ size - (LastSentByte - LastACKByte) \tag{6.4}$$

To gain the insights behind this equation, let's assume that the data flows only one way and that the receiver application process is extremely slow in comparison to the TCP sending endpoint. To this end, as the sending endpoint transmits more segments, the feedback Window size shrinks smaller and smaller. This value in turn decreases the Usable Windows. When the Usable Window equals to 0, the sender must stop sending any further data segment and must wait for the receiver buffer to free up. In this way, the receiver can throttle the speed of the TCP sending endpoint.

Until this point, the slow receiver succeeds in stopping the fast sender, giving it time to process the received data. However, another question arises, when the receive buffer is freed, how can the sending endpoint know about this buffer availability so that it can feed the receiver with more data? In TCP implementation, the receiver only sends a segment to the sender if it has data to send to the sender or when it responds to a segment from the sender. In the case that the data flows only one way, the sender would not know when the receiver is ready again. To overcome this issue, when the Usable Window equals 0, TCP allows the sender to try to probe the buffer availability at the receiver by periodically sending a segment with the data size of one-byte long. Even if this 1-byte segment is discarded due to occupied buffer, the receiver still has to send acknowledgements, through which the sender is aware of the receiver available buffer. Eventually, the receiver's buffer will become available and one of those 1-byte segment will trigger an acknowledgement that contains a nonzero Window size and data can be resumed from the sender at a fast speed again.

6.3.6 TCP Congestion Control

In the previous subsection, we studied how flow control is realized to adjust the sending rate according to the capability of the receiver. However, flow control alone does not guarantee that the network is able to sustain such a traffic flow. For example, the transmitter node may attach to a 1Gbps Ethernet link while a link on the path to the destination is only a 10Mbps link. In this case, if the transmitter tries to pump traffic as fast as possible into the network with respect to only the flow control mechanism, it will overflow the buffer of the routers on these bottleneck low-speed links. Moreover, on an internetworking environment, there are multiple connections that compete for the bandwidth of one link, as illustrated in Figure 11. In this case, if each connection selfishly keeps transmitting as fast as possible, the output link will not be able to keep up with the aggregated rate and will start dropping packets due to output buffer overflow. In either case, pumping too much traffic into the network will cause congestion.

When congestion occurs, network throughput will be affected severely. As illustrated in Figure 12, at a low input load, the output interface can accommodate the input traffic easily and the output throughput grows linearly with the traffic load. As the input traffic increases further, the output interface will not be able to handle all the incoming packets and those that exceed the output bitrate will be buffered in output buffer. As the buffer grows larger, it takes more time for a packet to be delivered to the output interface and hence, the rate of increase in the outgoing throughput decreases. At a certain point, the output buffer is not big enough to hold all the exceeded packets and the output interface starts dropping packets. Those packets that are dropped must be retransmitted according to TCP reliable data transfer. Those retransmissions come back and cause further congestion. Those retransmissions also stop new packets from being delivered and hence, the outgoing throughput decreases to 0. Sending traffic in an un-corporative way is also the reason behind the congestion collapses at the birth of the Internet, which caused a severe degradation in throughput performance.

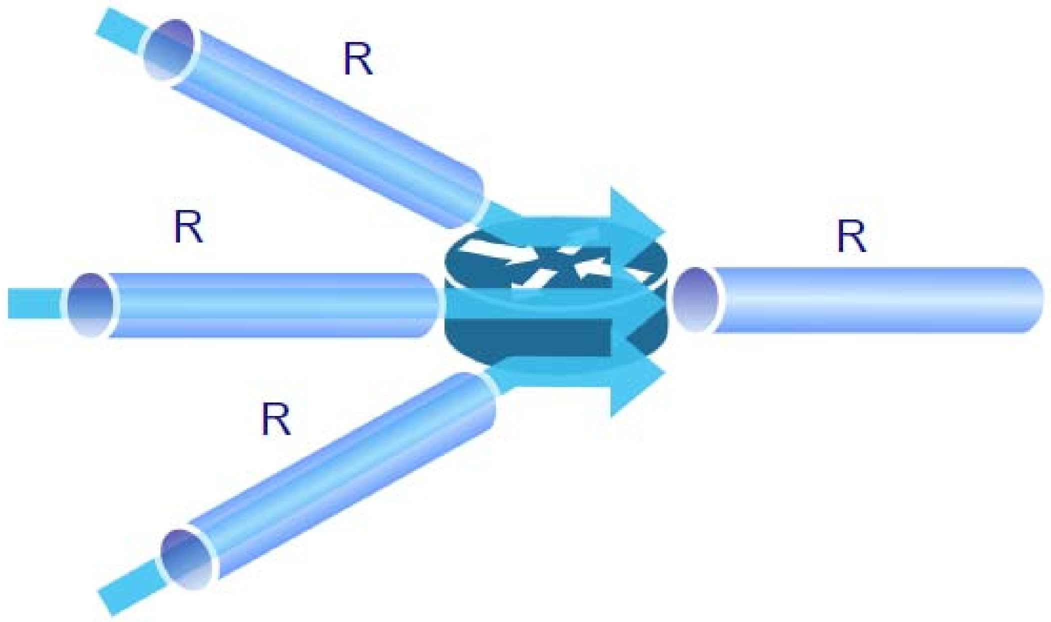

Figure 11: Network congestion due to contention of multiple simultaneous traffic flows.

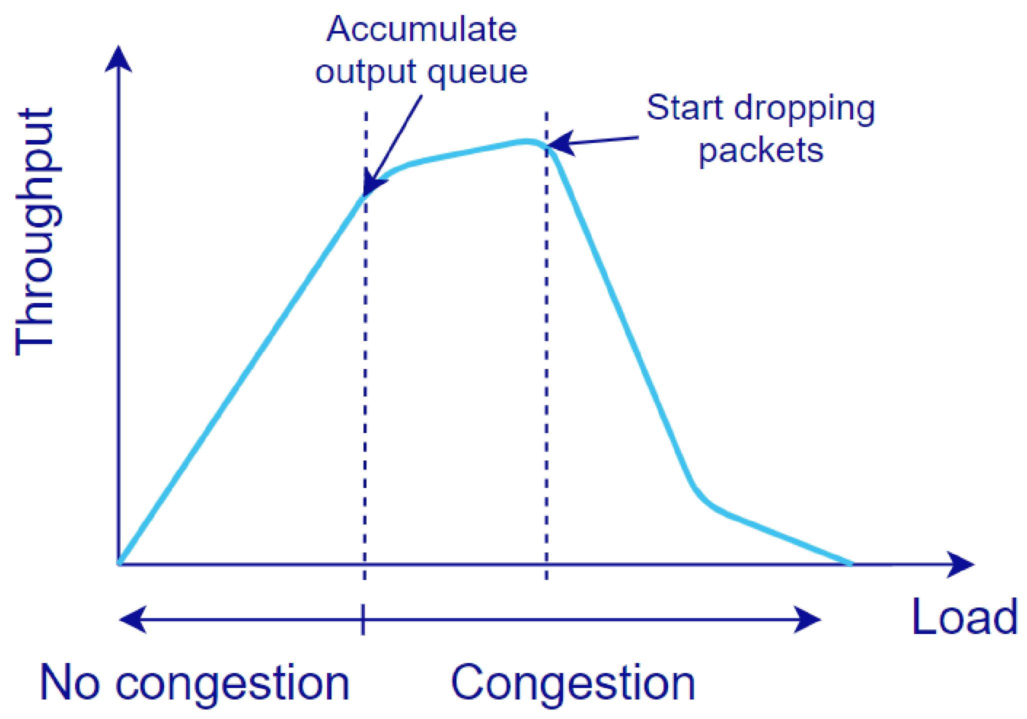

Figure 12: Throughput performance with input load level.

From the above discussion, it is clear that in a shared internetwork environment, using the network aggressively is not a good policy and will penalize the achievable throughput. However, sending traffic in a conservative way is also not good as the available capacity in the network will be underutilized. The best way should be striking the balance between these two extremes and using exactly the available bandwidth in the network for transmission. Since the available bandwidth in the network changes from time to time, the sending endpoint must continuously sense this available capacity to adjust its sending speed accordingly. In TCP, the arrivals of data segments are confirmed by ACKs; hence, TCP can make use of these ACKs to be aware of the network conditions. Motivated by this observation, several congestion control mechanisms were designed and integrated into TCP. This topic is still an ongoing research today and, in this subsection, only the basic and standard congestion control mechanisms in TCP are covered.

To regulate the sending traffic with respect to the network condition, the sending endpoint maintains a **Congestion Window** which limits the amount of data that it can send at a time without overloading the network. With the presence of this Congestion Window, the Usable window which defines the maximum amount of data the sending endpoint could send at any time must be updated according to equation (6.5). In this equation, WS and CW are the abbreviation of Window size and Congestion Window accordingly. Since the focus of this subsection is congestion control, let's ignore the effect of flow control for now by assuming that Window size is greater than Congestion Window.

$$Usable\ Window = \min\{WS, CW\} - (LastSentByte - LastACKByte) \qquad (6.5)$$

In TCP, the Congestion Window is updated dynamically at the sender with respect to the feedback from the network such as the receipt of an ACK or a presence of a packet loss. The principle for this adjustment is that each data segment is considered as a probe for evaluating the network condition. If in response to a sent data segment, the sender receives an ACK, this essentially means that the path along which this segment took was not congested. Hence, the sender can try to send more traffic by enlarging the Congestion Window. On the other hand, if the sender senses a packet loss, it concludes that the network is under congestion and since there will be penalties if it makes the issue worse (as discussed above), it should decrease the Congestion Window and start sensing the network again. The decrease of the Congestion Window throttles the sender's rate, giving the routers in the network a chance to clear their buffers; hence, giving a chance for the network to recover from this congestion. It is noted that the assumption that packet loss is due to congestion is not 100% correct as this phenomenon can be caused by bit errors as well. However, since bit error rate of modern transmission links are normally very low and data link protocols can be integrated with error control mechanisms such as ARQ, it is safe to infer that the major cause of packet loss is due to congestion. The standard congestion control mechanism in TCP is divided into states: Slow Start, Congestion Avoidance and Fast Recovery as will be discussed below.

Slow Start

When the connection is first started, the Congestion Window is initialized at 1 MSS. At this moment, the sending endpoint has no knowledge about the network condition, so it assumes that the network is free. Hence, it will try to probe the network aggressively by opening the Congestion Window by 1 MSS for every received ACK. As illustrated in Figure 13, Slow Start increases the congestion window **exponentially**. After the first successful ACK, the congestion is increased to 2, after receiving the next two ACKs, the Congestion Window increase by another 2, to 4 and hence, TCP doubles the sending rate every RTT. The motivation behind this exponential increase of Congestion Window is for the sender to quickly estimate the available capacity of the network. Since the increase trend is actually exponential, it is confusing for some that this state has the term "*slow*" in its name. Nevertheless, it is actually *slow* if we compare it with the TCP transmission speed without Slow Start where an ending endpoint would send as much traffic as the Window size allows (and cause congestion collapses).

The second case when Slow Start is used is when the sender experiences a segment timeout event. As discussed above, from the sender perspective, this event implies a segment loss in the network. Upon observing this event, the sender assumes that its previous estimation of Congestion Window is not correct, and it has to do this estimation again. As a result, the sender resets its congestion window to 1 MSS and restarts Slow Start again. However, this time, the sender already has an idea about the available network capacity so it should not let Slow Start overloads the network again. To this end, the sender maintains the second variable, namely **Congestion Threshold**, which indicates the maximum window size of the Congestion Window where Slow Start should stop. In particular, in this case, the Congestion Threshold is set to half of the Congestion Window before the timeout event occurs.

With the two use cases of Slow Start, there are two ways that the exponential increase in Congestion Window of Slow Start stops. The first case is when a timeout is observed, in this case, the Congestion Threshold is set to half of the latest Congestion Window while the Congestion Window is reset to 1 MSS. After this event, Slow Start will restart again. The second case is when the Congestion Window approaches Congestion Threshold, in this case, the sender moves to the second phase of TCP congestion control, the Congestion Avoidance.

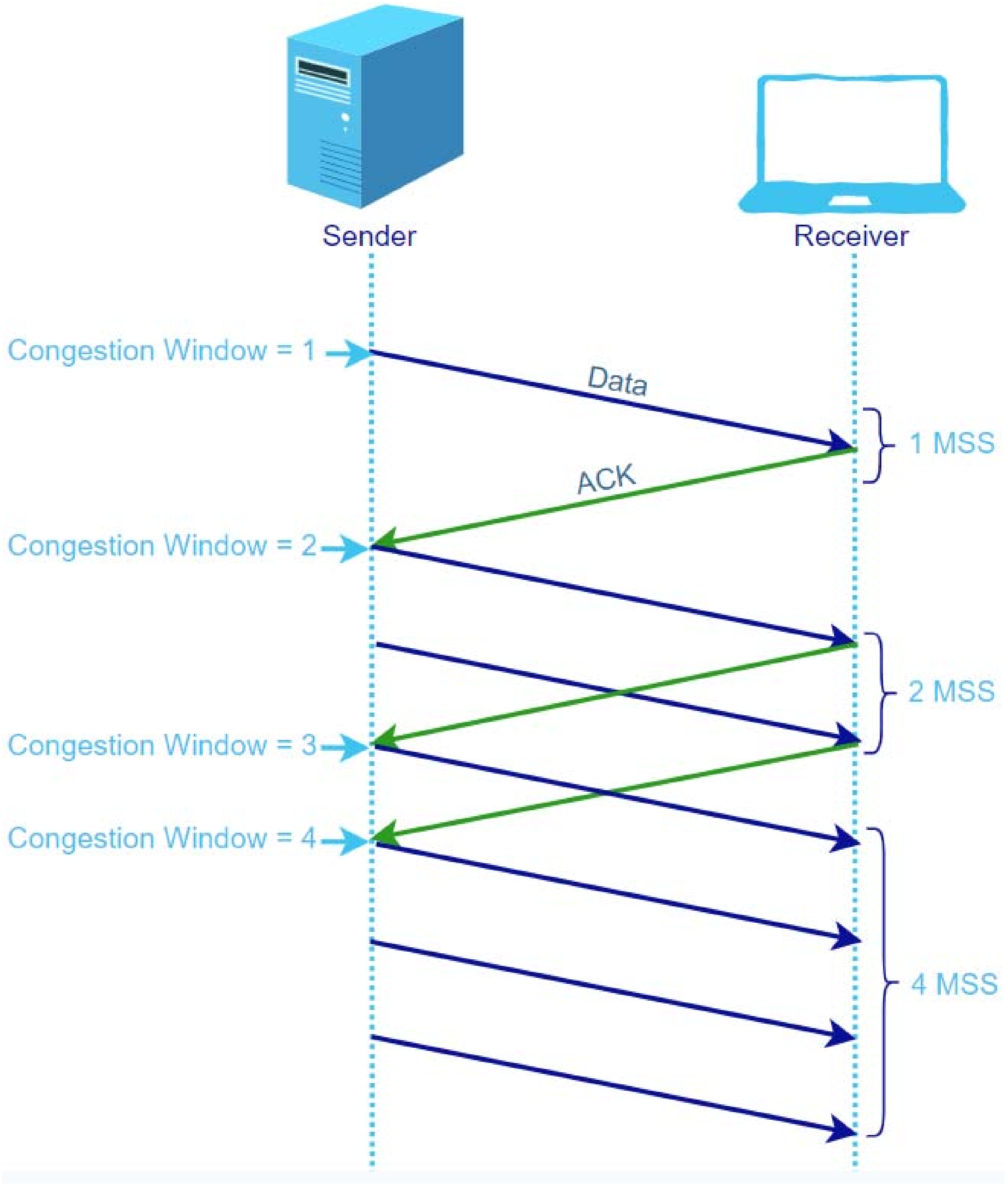

Figure 13: The increase of Congestion Window during TCP Slow Start.

Congestion Avoidance

Upon entering this state, the Congestion Window equals to the Congestion Threshold and the sender knows that it is approaching the network capacity. At this phase, it should not introduce data into the network aggressively as in the Slow Start but rather in a more conservative manner. In particular, in this phase, the sender will increase the Congestion

Window linearly by adding one MSS every RTT. Implementation wise, the Congestion Window is increased by an amount of $MSS \times MSS/Congestion\ Window$ for every ACK. For example, if the initial Congestion Window of Congestion Avoidance is 8, for each ACK received, the Congestion Window will be increased by an amount of 1/8 MSS. Hence, after one RTT, if the sender receives all 8 ACKs, the Congestion Window will be increase by 1MSS to 9MSS. Figure 14 illustrates this increase in Congestion Window and the linear increase in the number of segments sent from the sender. The linear increase of Congestion Window in Congestion Avoidance avoids the sender to cause severe congestion but at the same time, making sure that it can continuously probe the network for more available capacity.

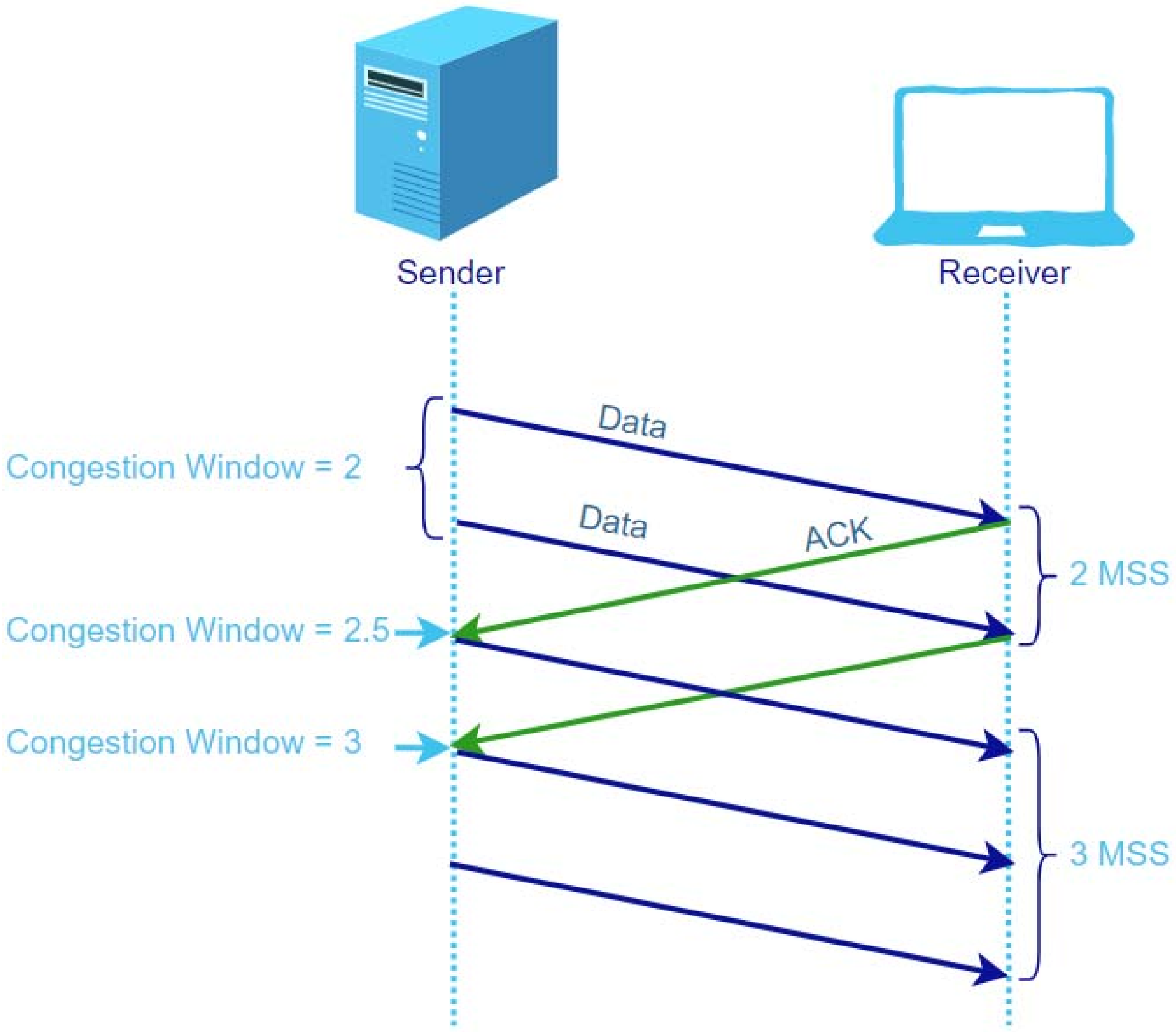

Figure 14: Linear increase of Congestion Window in Congestion Avoidance phase.

The Congestion Avoidance state of TCP ends when the sender detects a packet loss. Recall from the previous discussion that a packet loss can be identified by two ways in TCP. The first case is when the sender experiences a timeout. In this case, the Congestion Avoidance moves back to Slow Start, the Congestion Threshold is set to half of the latest Congestion Window and then, the Congestion Window is reset to 1 MSS. The second case is when the sender receives three duplicate ACKs. In this case, it will move to Fast Recovery state where both the Congestion Window and the Congestion Threshold is set to half of the latest Congestion Window.

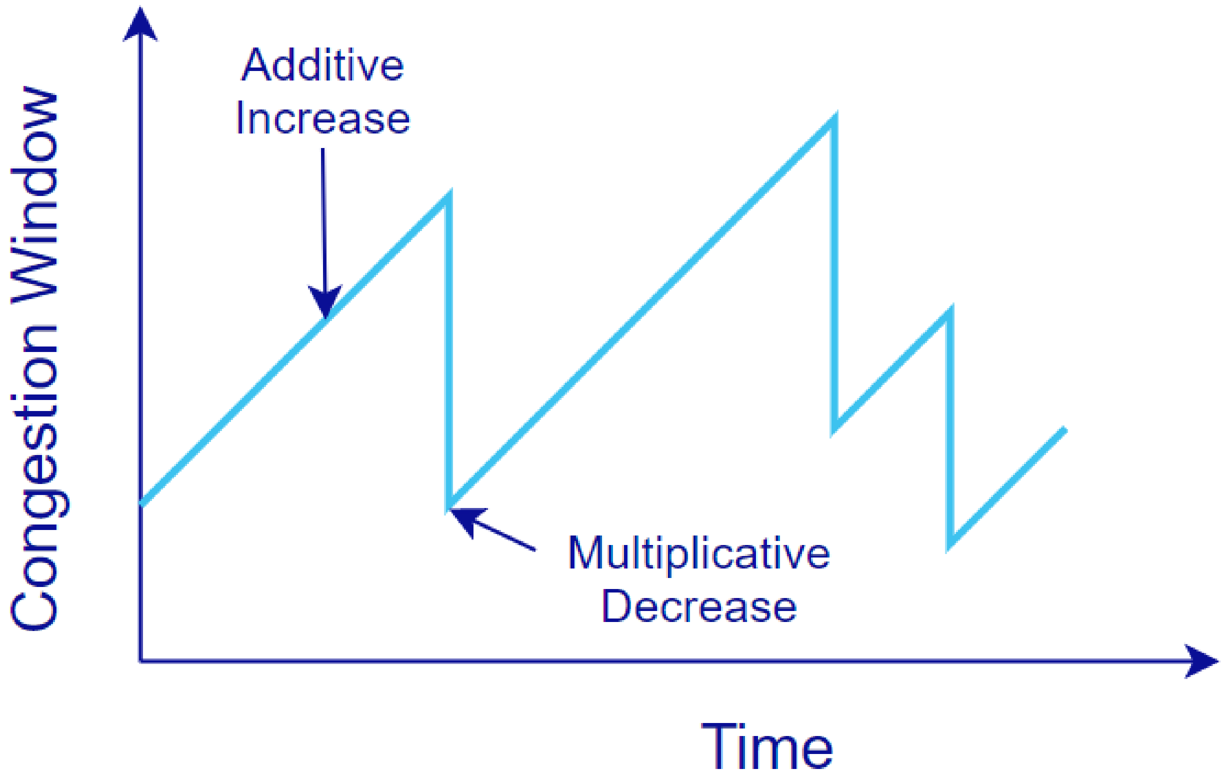

Figure 15: The saw tooth pattern of TCP Additive Increase, Multiplicative Decrease.

Before we move to the discussion on Fast Recovery, there is a last point to elaborate about the congestion control of TCP. It can be seen that in this Congestion Avoidance phase, the reward is only 1 MSS every RTT for the successful ACKs but the penalty is half of the Congestion Window in face of a packet loss. This behaviour is referred to as **Additive Increase, Multiplicative Decrease (AIMD)** and is an important congestion control of TCP. If we ignore the Slow Start phase (by eliminating the extreme congestions in the network that cause timeout), the evolution of the Congestion Window of a TCP connection would exhibit a saw tooth pattern as depicted in Figure 15. The intuition behind this AIMD is to avoid causing congestion in the network. In particular, the sender would carefully increase the speed of pumping traffic into the network while actively dropping its transmission rate drastically when it senses that a congestion is likely to occur. This aggressive back off allows the network to have a chance to resolve the backlogged packets in the output buffer of routers, which would free more capacity for the next iteration of AIMD.

Fast Recovery

It worth noting that Fast Recovery (usually referred to as **TCP Reno**) is an addition to the initial proposal of TCP congestion control (usually referred to as **TCP Tahoe**). The difference between the two schemes is in the way they handle segment loss in the case of three duplicate ACKs. TCP Tahoe does not differentiate this case with the case segment loss caused by timeout, i.e. transiting the connection into Slow Start. However, the authors of TCP

Reno argued that the presence of those duplicate ACKs indicates that despite the loss, some segments actually got through the network, to the destination. Effectively, the network congestion should not be as severe as in the case of timeout when no further segments got through. As a result, the response to segment loss occurs by duplicate ACKs should be treated differently. In particular, they reasoned that instead of dropping the Congestion Window to 1 and restarting with Slow Start, a better solution would be removing the Slow Start phase and using directly to the Congestion Avoidance phase where both the Congestion Window and the Congestion Threshold equal to half the latest value of Congestion Window. In this way, the sending speed is not much compromised while still allowing the network to release previous congestions.

A running example of TCP congestion control mechanisms

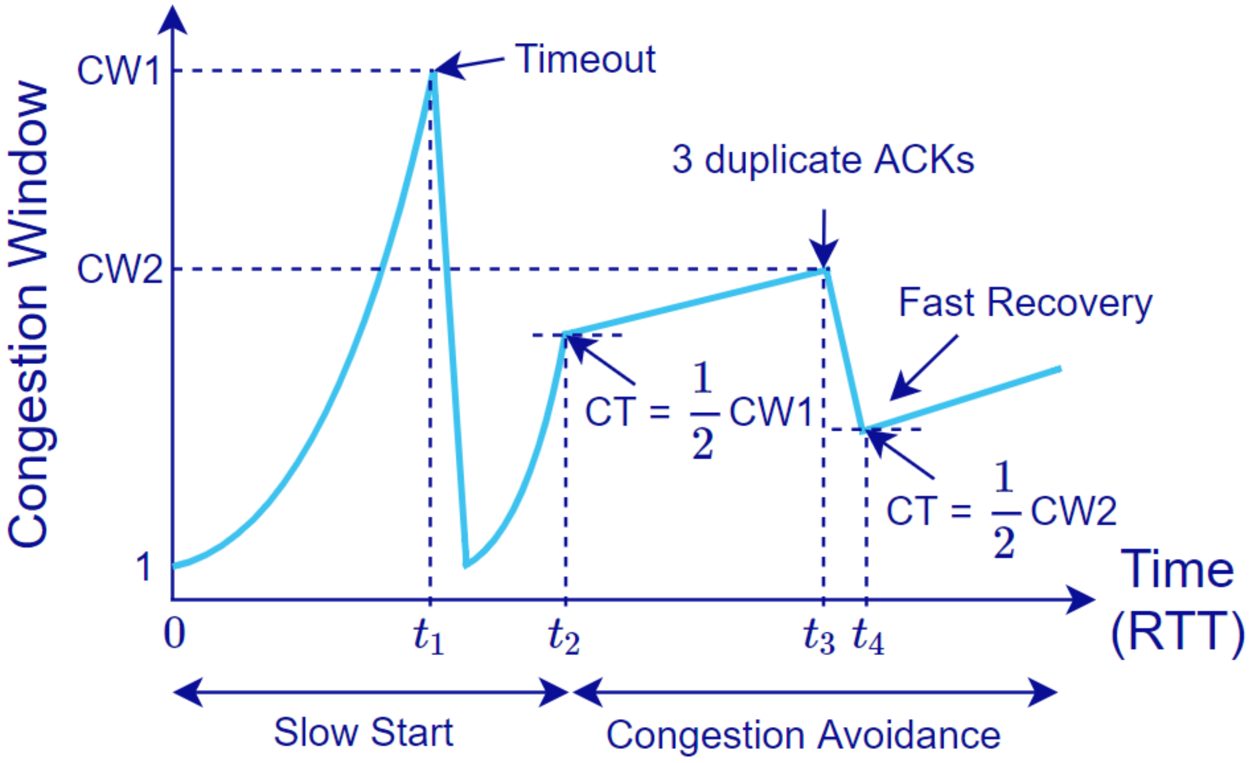

Figure 16: An example evolution of TCP Congestion Window in TCP Reno.

Figure 16 illustrates an evolution of Congestion Window in a TCP Reno connection. At the beginning of the TCP session, the sending endpoint uses Slow Start to probe for the available capacity of the network. In this phase, the Congestion Window increases from 1 to CW1 where the sender detects a timeout after t_1 RTTs. At this time, the sender assumes that it already overloads the network and restart the Slow Start procedure again with Congestion Window of 1. However, this time, it sets the Congestion Threshold (CT) to half of the latest Congestion Window (CW1). The next Slow Start phase increases the sender Congestion Window exponentially to the previously calculated Congestion Threshold. After this time, Slow Start stops and the connection state moves to Congestion Avoidance phase where the Congestion Window increases linearly. After t_3 RTTs, the sender detects 3 duplicate ACKs, Fast Recovery is executed where the Congestion Window is set to half of the latest Congestion Window (CW2) and then increases linearly.

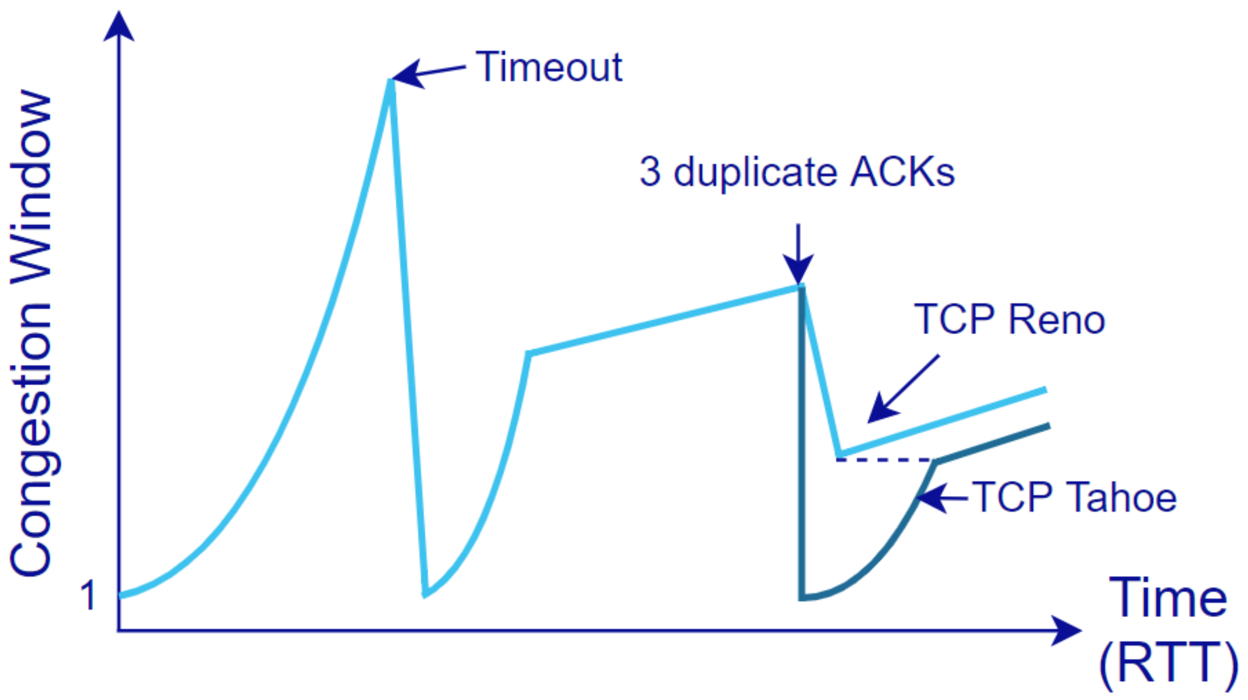

Figure 17: Comparison between TCP Reno and Tahoe.

Figure 17 further illustrates the difference between TCP Tahoe and Reno with the same events as in Figure 16 during the lifetime of the TCP connection. The major difference is found in the way they handle 3 duplicate ACKs. In TCP Tahoe, it does not differentiate the segment loss due to a timeout or 3 duplicate ACKs and force the connection into Slow Start state in both cases. It can be seen from Figure 17 that the lack of capability to differentiate these cases makes the Congestion Window in TCP Tahoe lags behind that of TCP Reno, hence penalizing its throughput performance.

Fairness in TCP congestion control

From the previous discussion, it is demonstrated that TCP congestion control forces a connection to decrease its bandwidth aggressively when it senses that a congestion occurs. However, this AIMD approach also helps to solve the issue of fairness among users in a distributed manner.

To see how this works, consider the case of two users, each with a TCP connection, sharing a common link with normalized bandwidth of 1. Figure 18 illustrates the dynamics of the normalized bandwidth of the two users. The diagonal dotted line that connects the two points (0,1) and (1,0) indicates the points where the sum normalized bandwidth of the two users is 1. In other words, the total bandwidth utilization of the two users equals to the link bandwidth and hence, the link is fully utilized. Since the total bandwidth of the link is 1, all operational combinations of the bandwidth shares of the two users must be below this line. The second dotted line starts from the origin of the plot and tilts 45^o from the horizontal line specifies the operational points where the bandwidth shares of the two users equals to each other, meaning fairness is achieved. The intersection of these two dotted lines specifies the optimal operational point where the bandwidth share of the two users equals to each other and summed to 1, meaning, full usage of the link as well as fairness are achieved.

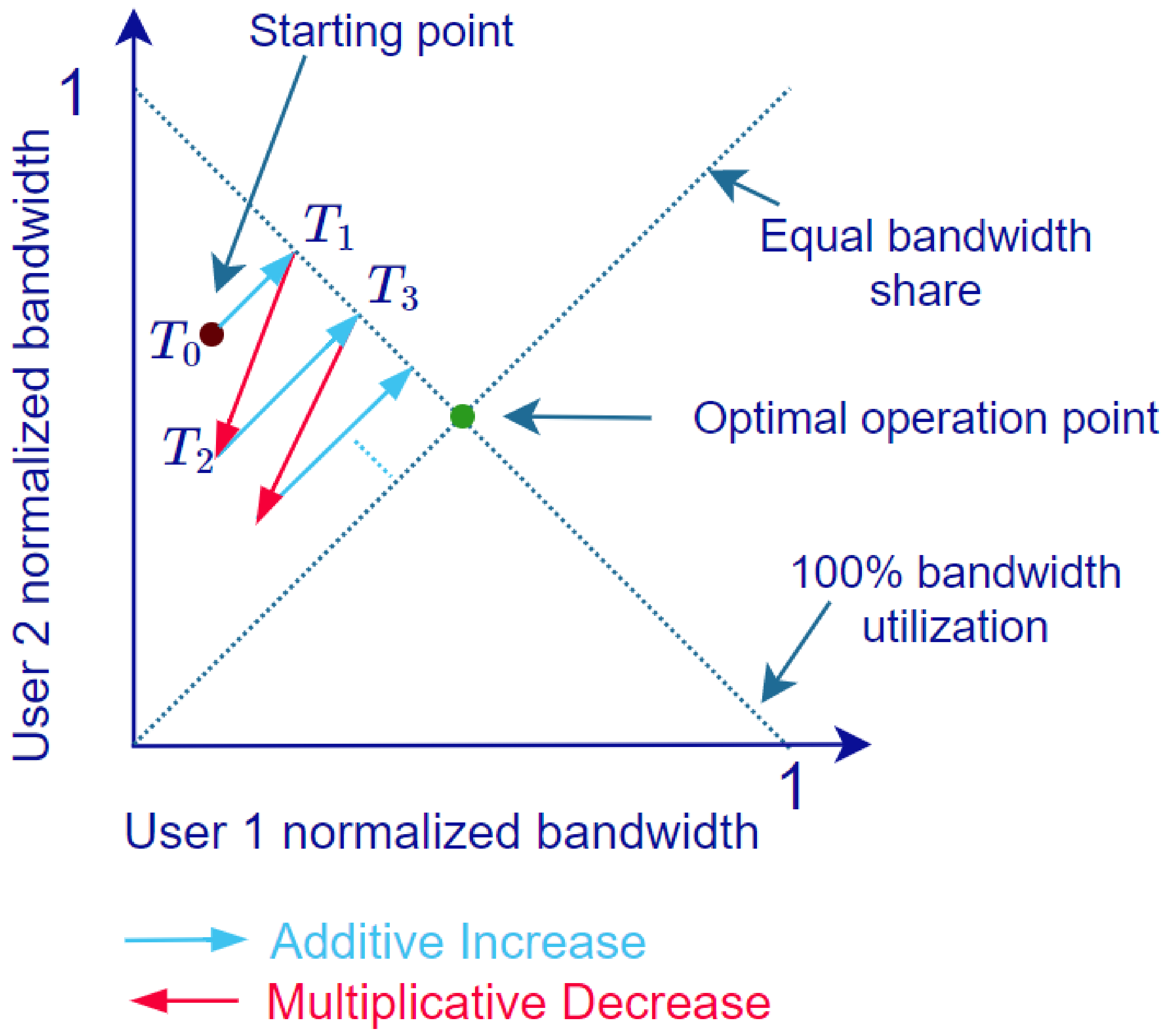

Figure 18: Fairness in TCP with two competing users

From an arbitrary starting point T_0 in the operational bandwidth shares of the two users, let's trace the AIMD operations of the two users to see that these operations help to converge the bandwidth usage to the optimal point. Let $r_1^{T_i}$ and $r_2^{T_i}$ be the bandwidth shares of the two users accordingly ($r_1^{T_i} + r_2^{T_i} \leq 1$) at time T_i. The initial difference in terms of bandwidth of the two users, Δr^{T_0}, can be expressed in equation (6.6).

$$\Delta r^{T_0} = \left| r_1^{T_0} - r_2^{T_0} \right| \tag{6.6}$$

Assume that $r_1^{T_0} + r_2^{T_0} < 1$, the two hosts apply Additive Increase mechanism to slowly increase their Congestion Window. At time T_1, this accumulated bandwidth exceeds the link capacity, packet loss occurs. Let δr^{T_1} be the bandwidth gain that each users gets after this Additive Increase, the bandwidth share of the two users can be expressed in equation (6.7).

$$\begin{aligned} r_1^{T_1} &= r_1^{T_0} + \delta r^{T_1} \\ r_2^{T_1} &= r_2^{T_0} + \delta r^{T_1} \end{aligned} \tag{6.7}$$

Due to the packet loss, Multiplicative Decrease kicks in. At T_2, both users decrease their bandwidth share by half as expressed in equation (6.8).

$$\begin{aligned} r_1^{T_2} &= \frac{r_1^{T_0} + \delta r^{T_1}}{2} \\ r_2^{T_2} &= \frac{r_2^{T_0} + \delta r^{T_1}}{2} \end{aligned} \tag{6.8}$$

As a result, after this Multiplicative Decrease, the difference in bandwidth share between the two users becomes Δr^{T_2}, as expressed in equation (6.9).

$$\Delta r^{T_2} = \left| r_1^{T_2} - r_2^{T_2} \right| = \frac{\left| r_1^{T_0} - r_2^{T_0} \right|}{2} = \frac{1}{2} \Delta r^{T_0} \tag{6.9}$$

It can be seen that $\Delta r^{T_2} = \frac{1}{2} \Delta r^{T_0}$, hence, the bandwidth difference of the two users becomes half of their initial difference, meaning, after the first cycle of AIMD, the bandwidth shares of the two users are closer to each other and the operational point of the two users is closer to the equal bandwidth share line. From the point T_2, the two users will start increasing their Congestion Window gradually and a new cycle of AIMD is executed. Follow the same derivation as above, it can be seen that at the end of this second Additive Increase cycle, the bandwidth difference between the two users is further halved, making them further closer to each other.

As derived above, it is demonstrated that with an arbitrary starting point, the Additive Increase mechanism helps to push the operational point towards the 100% bandwidth utilization line (efficient usage of available bandwidth) while the Multiplicative Decrease helps to bring the operational point towards the equal bandwidth share line (fairness between the sessions) as in equation (6.10). As a result, the repetition of AIMD cycles elegantly helps to gradually converge the operational point to the optimal operational point where both fairness and efficiency are achieved.

$$\begin{aligned} &\text{AI:} \quad r_1^{T_i} + r_2^{T_i} \rightarrow 1 \\ &\text{MD:} \ \Delta r^{T_i} = \left| r_1^{T_i} - r_2^{T_i} \right| \rightarrow 0 \end{aligned} \tag{6.10}$$

It is important to note that AIMD is not perfectly foolproof and there are many cases that can leads to fairness violation. First, in the above derivation, we assumed that the two connections having the same RTT which leads to $\delta r^{T_1} = \delta r_1^{T_1} = \delta r_2^{T_1}$. If the RTTs of the two connections are different from each other, the one with smaller RTT will get a higher bandwidth gain in Additive Increase phase before packet loss kicks in. As a result, this user will be more favorable in terms of bandwidth. In addition, in the case one user establish multiple parallel connections, the bandwidth share will be distributed among the connections, making the user with a smaller number of connections suffers from bandwidth starvation. Using multiple parallel connections is also one of the basic principles of download accelerators that we use on the daily basis.

6.4 Traffic Management

In the previous section, we already learnt how TCP congestion control mechanism helps to regulate traffic flows to allow fairness in the network. However, in the internetworking environment, there are not only TCP-based applications and fairness may not be a good goal to follow in many cases. For example, service providers may want to offer different subscription levels to their users depending on their contracts and payments. As such, it is important for the users to adapt their traffic to these requirements and for the provider to manage the incoming traffic to enforce these policies. In addition, different from circuit-switch traffic, which is normally constant, packet-switch traffic is very bursty. A connection may remain silent for a long period of time then suddenly send/receive a large amount of data. As discussed in the previous section, bursty traffic is generally not desirable as it would overwhelm router's buffer quickly causing packet loss and congestion. Consequently, traffic management is very important in regulating data transport across the Internet.

In order to manage traffic flows, two commonly used mechanisms are utilized, namely traffic policing and shaping as described below.

- **Traffic policing:** traffic policing is a mechanism for monitoring and enforcing policies on traffic flows that are fed into the network. For instance, if a user subscribes for an average bandwidth of 50Mbps, policing should ensure the mean of its long-term traffic output should not exceed this number while accommodating a certain level of traffic bursts from the user. Traffic policing can be implemented at either the user side or at the provider side. If it is implemented at the user side, the traffic that exceeds the policies can be buffered for latter transmission. If it is enforced at the provider side, excess traffic is normally dropped or marked with a lower priority. In the latter case, those lower priority packets will be the first candidates for packet dropping if congestion occurs latter on in the network.
- **Traffic shaping:** traffic shaping is a mechanism that is usually used for regulating bursty traffic, transforming its bursty pattern into a smoother flow. Bursty traffic is known to cause congestion that degrades the network performance by increasing and making delay unpredictable. By altering traffic bursts into a smoother flow, traffic shaping mechanism prevent congestion from happening; hence, optimizing the network performance and helping to guarantee the QoS of traffic flows. Traffic shaping is also used to change the input traffic pattern to meet certain requirements of traffic policing.

There are two major methods in implementing traffic policing and traffic shaping, namely leaky and token bucket as will be discussed below.

6.4.1 Leaky Bucket

As its name indicates, a leaky bucket is analogous to a bucket with a leaky hole at the bottom as illustrated in Figure 19. Water can be poured into the bucket at irregular rates but the hole at the bottom of the bucket only allow a constant rate of water draining. Due to the limited bucket size, if the incoming water flow is greater than the output rate, after a certain time, the bucket is overflowed. However, if the incoming flow burst does not last long, the bucket can hold all of the incoming water and regulate it with a constant output (leaky) rate. In this case, the bucket volume is used to consume the bursty in coming flow and maintain a constant, smooth output rate. Applying into traffic management, the leaky bucket is essentially a single-server queue where the queue length (analogous to the bucket size) is used to absorb the sporadic incoming traffic and to maintain a constant output rate according to the requirements of the policing or traffic shaping mechanism.

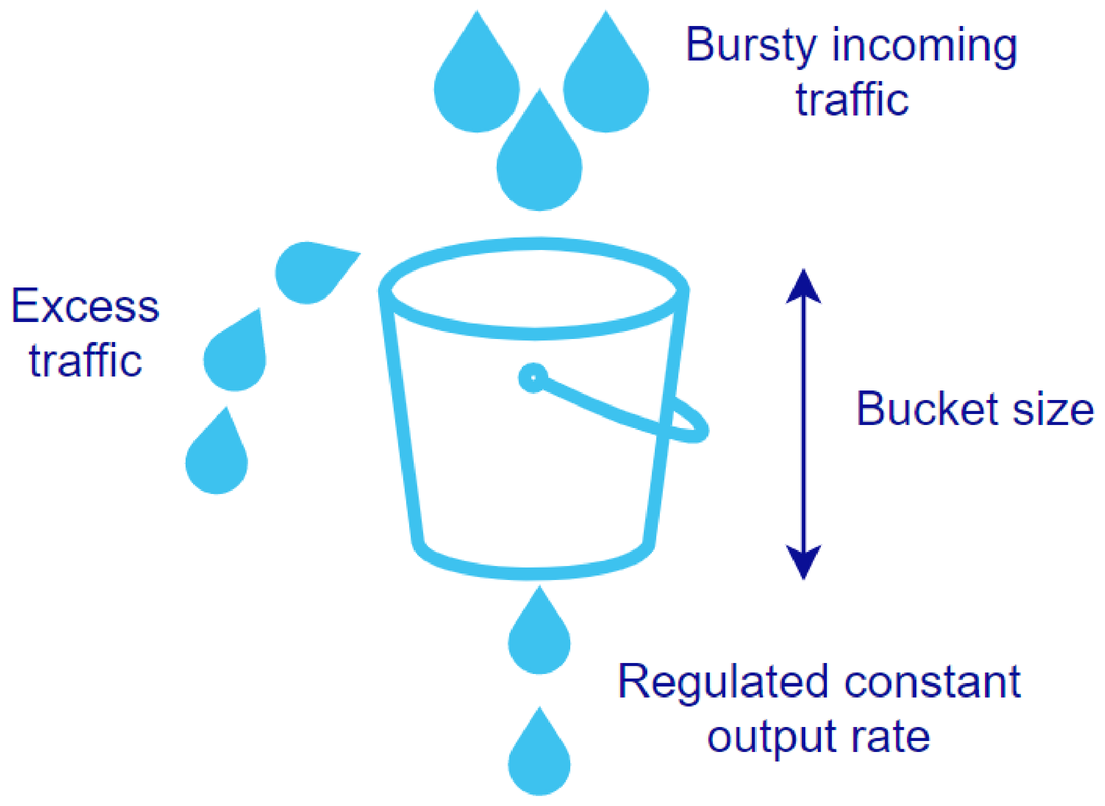

Figure 19: Leaky Bucket mechanism.

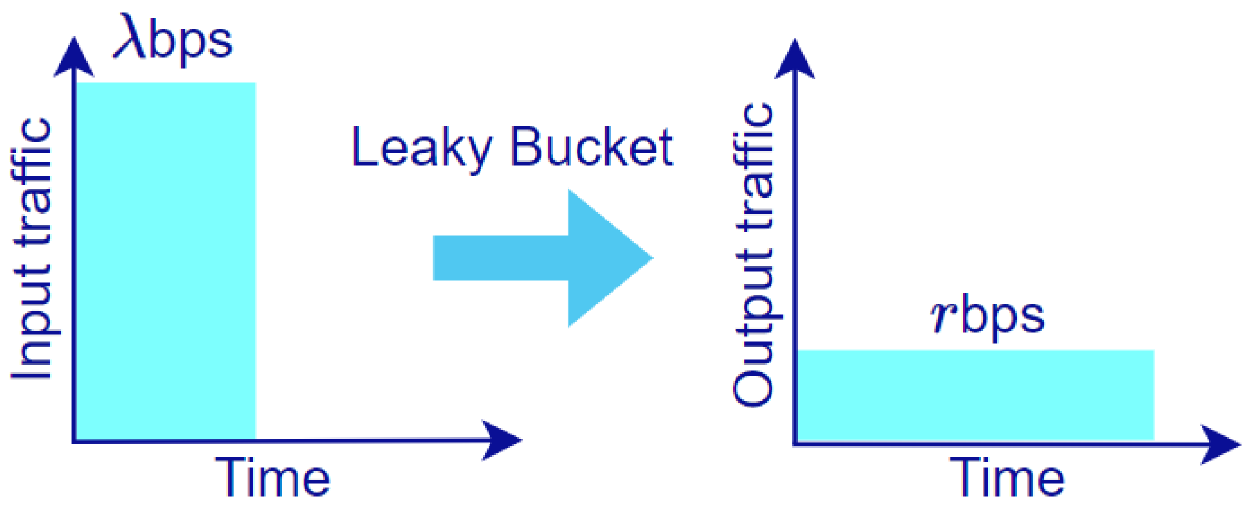

Figure 20: Traffic pattern before and after a leaky bucket.

The Leaky Bucket model as shown in Figure 19 helps to regulate the average output traffic rate as illustrated in Figure 20 where λ is the incoming traffic rate and r is the outgoing traffic rate according to the leaky bucket.

To have a better understanding of the Leaky Bucket, let's consider an example where a Leaky Bucket has a bucket size of $B = 7$ packets and a leaky rate of $r = 5$ packets per second. Suppose that the bucket is initially empty when traffic comes as 3 bursts of 12, 8 and 4 packets at time instances $t(i)$=1, 2, 4s. To evaluate the evolution of this Leaky Bucket, let's further define the following variables:

$a(i)$: the arriving traffic at time instance $t(i)$.

$u(i)$: remaining packets in the bucket at time instance $t(i)$, before receiving $a(i)$.

$b(i)$: current bucket occupancy.

$e(i)$: excess packets due to insufficient bucket size.

$d(i)$: departing traffic from $t(i-1)$ to $t(i)$.

Following these definitions, the relationship between these variables can be expressed as in equation (6.11)

$$\begin{aligned} u(i) &= \max\{b(i-1) - [t(i) - t(i-1)] \times r, 0\} \\ b(i) &= \min\{u(i) + a(i), B\} \\ e(i) &= \max\{u(i) - a(i) - B, 0\} \\ d(i) &= \min\{[t(i) - t(i-1)] \times r, b(i-1)\} \end{aligned} \tag{6.11}$$

Then, the Leaky Bucket evolution in this example can be illustrated as in Table 6.1.

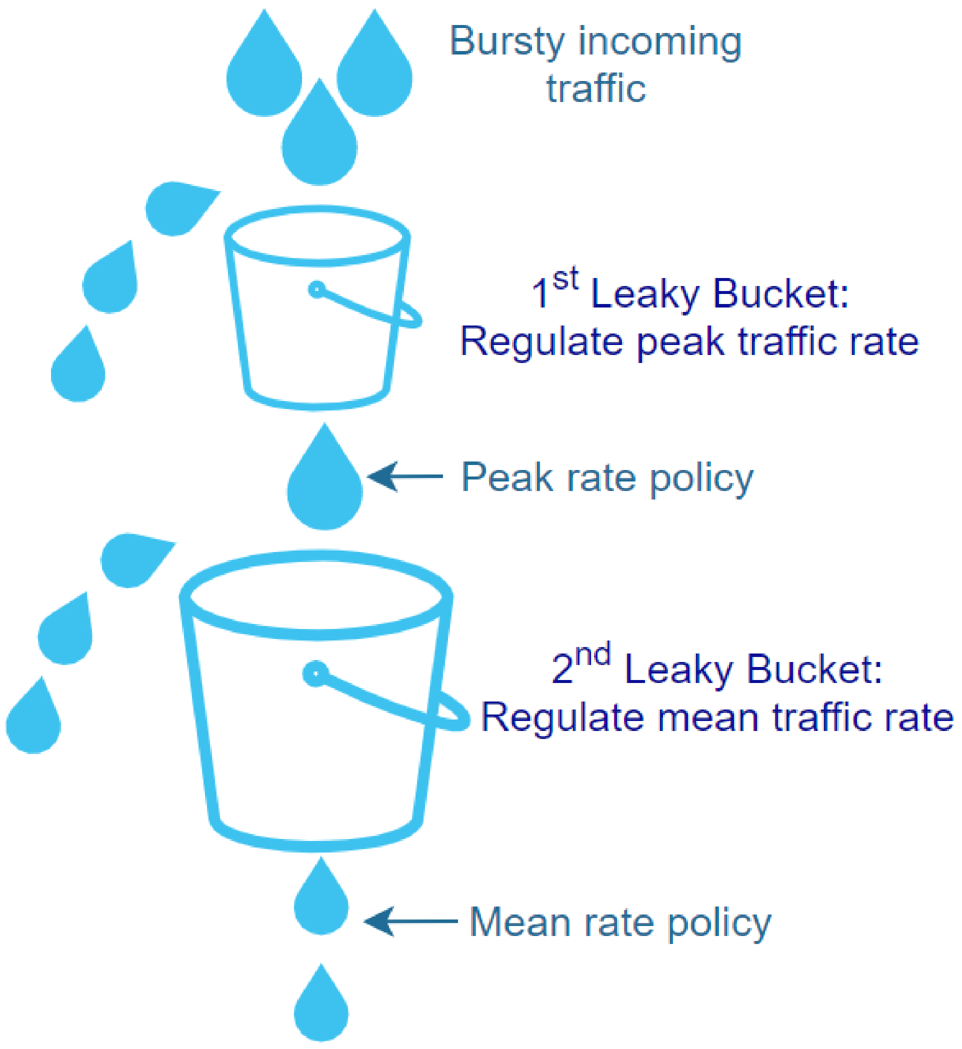
Bursty incoming traffic
1st Leaky Bucket: Regulate peak traffic rate
Peak rate policy
2nd Leaky Bucket: Regulate mean traffic rate
Mean rate policy

Figure 21: Dual Leaky Bucket.

Table 6.1: Example of Leaky Bucket evolution.

$t(i)$	0	1	2	4
$a(i)$	0	12	8	4
$b(i)$	0	7	7	4
$u(i)$	0	0	2	0
$e(i)$	0	5	3	0
$d(i)$	0	0	5	7

This Leaky Bucket model can be modified to regulate more features of the incoming traffic. For example, both the average and the peak output traffic rates can be controlled using a management model with two Leaky Buckets as illustrated in Figure 21. This model is referred to as the Dual Leaky Bucket model. In this model, the first leaky bucket has a high output rate which corresponding to the desired peak traffic rate and a small bucket size. This bucket cuts off any exceed incoming rate that is higher than the designed peak rate. Then, the output of this bucket is the input to the second leaky bucket, which is used to regulate the average traffic rate. This second leaky bucket has the output rate as the desired average rate but is built with a larger bucket size, which can accommodate some degrees of busty traffic. As a result, the output of this Dual Leaky Bucket model is a regulated traffic flow with the designed average rate while being able to withstand excess traffic of up to the designed peak rate for a limited amount of time.

6.4.2 Token Bucket

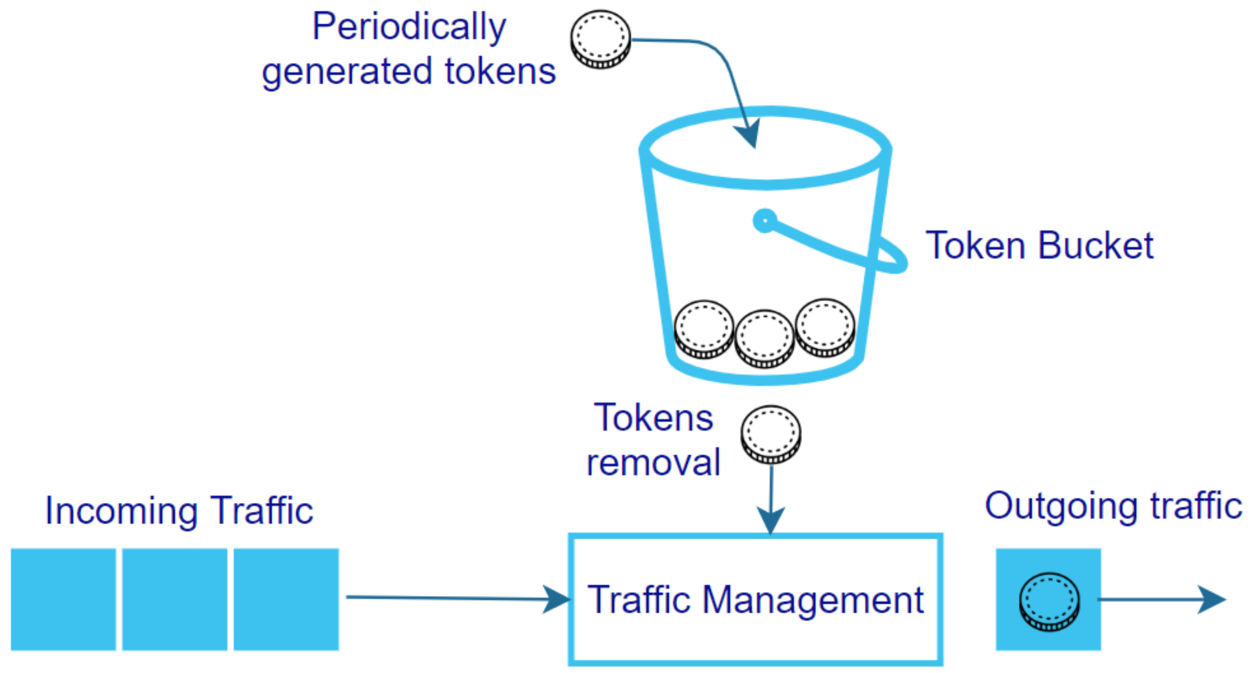

Figure 22: Token Bucket mechanism.

It can be seen in Figure 20 that the Leaky Bucket approach to traffic management imposes a hard constraint on the output rate. Traffic bursts with higher incoming rates have to be buffered into the bucket, slowly waiting for their turn to be transmitted. This rigid constraint causes a significant delay, which may not be acceptable for some applications such as real-time applications. In those applications, traffic is not only bursty in its nature but is also delay sensitive. In this case, it is better to allow some degrees of burstiness in the output traffic to ensure the QoS level required by the transported application. To resolve this issue, a slightly different but more flexible mechanism was introduced, namely Token Bucket.

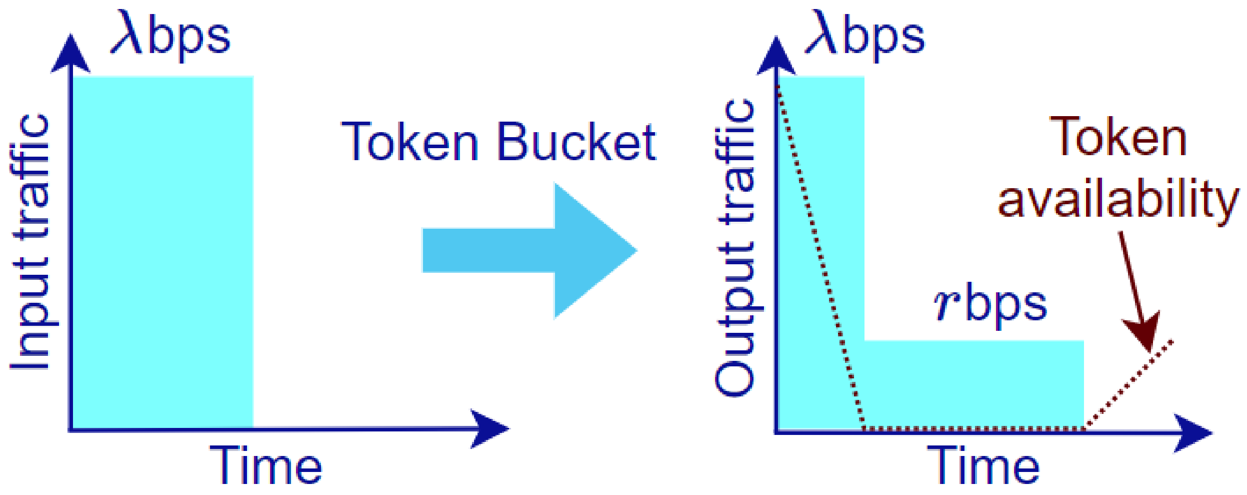

Figure 23: Traffic pattern before and after a token bucket.

In Token bucket mechanism, tokens are periodically generated with a rate of r and is stored in a bucket with a size of B. The available tokens are those that are presented in the bucket as illustrated in Figure 22. If the bucket is full, excess tokens are wasted and those do not contribute to the available token. An incoming packet can be served only if there are enough tokens in the bucket to cover the length of the packet (assuming one token can serve one byte of the packet). In this case, the appropriate number of tokens are removed from the bucket to serve the packet. If the bucket is short of tokens, the packet must be waited in a queue until the number of accumulated tokens is sufficient.

Assume that the maximum output rate is at least as high as the incoming traffic rate. In the case the token bucket is full, incoming traffic can be served as fast as it arrives without the output rate constraint as in the leaky bucket mechanism. This allows the accommodation of a certain degree of traffic burstiness. However, as the burst prolongs, the bucket runs out of tokens eventually and the output rate is throttled to the regenerating rate of the tokens. As a result, the token regenerated rate r helps to **regulate the long-term output traffic rate** to r while the bucket depth allows some bursty traffic to pass through. In a way, this behaviour is analogous to a person with steady income can sometimes afford a luxury item by saving.

Figure 23 illustrates the behavior of token bucket mechanism on the output traffic. Assuming that before the input traffic arrives, the bucket is full of tokens and the incoming traffic rate is higher than the token regenerating rate ($\lambda > r$). When the incoming traffic arrives, the tokens are drawn from the bucket and the output rate equals to the input rate. After the tokens are used up, the burst rate is no longer sustainable and the output rate drops to r, the token regenerating rate. In this case, the excess traffic will have to wait for new tokens to be served and will suffer from queuing delay. Eventually, when all of the incoming burst is served, the bucket starts accumulating tokens again.

To have a better understanding of the Token Bucket, let's consider an example where a Token Bucket has a bucket size of $B = 7$ tokens and a token replenishment rate of $r = 5$ tokens per second (assuming each token can be used to serve 1 packet). Suppose that the bucket is initially full of tokens and the queue size is infinite. Arriving traffic comes as 3 bursts of 12, 8 and 4 packets at time instances $t(i)$=1, 3, 4s. To evaluate the evolution of this Token Bucket, let's further define the following variables:

$a(i)$: the arriving traffic at time instance $t(i)$.

$q(i)$: current queue occupancy.

$b(i)$: current number of tokens left in the bucket.

$d(i)$: departing traffic from $t(i-1)$ to $t(i)$.

Following these definitions, the relationship between these variables can be expressed as in equation (6.12)

$$\begin{aligned} q(i) &= \max\{a(i) + q(i-1) - b(i-1) - [t(i) - t(i-1)] \times r, 0\} \\ d(i) &= \min\{b(i-1) + [t(i) - t(i-1)] \times r, q(i-1)\} \\ b(i) &= \min\{b(i-1) + [t(i) - t(i-1)] \times r - d(i), B\} \end{aligned} \tag{6.12}$$

Then, the Token Bucket evolution in this example can be illustrated as in Table 6.2

Table 6.2: Example of Token Bucket evolution.

$t(i)$	0	1	2	4
$a(i)$	0	12	8	4
$b(i)$	7	7	0	2
$q(i)$	0	12	8	4
$d(i)$	0	0	12	8

It is noted that from $t(1) = 1s$ to $t(2) = 2s$, the Token Bucket can output 12 packets as a result of the 7 initial tokens and the $5 \times 1 = 5$ additional replenished tokens during this period. This behaviour allows some bursty traffic to pass through for a short time. In contrast, the output rate of the Leaky Bucket is always constrained by the leaky rate as shown in Table 6.1.

As bursty traffic can be sustained for a short period of time, it is important to calculate this maximum bursty duration. It is noted that while being drawn from the buckets, the tokens continue to be regenerated. When the bucket is full and the output rate is at least as much as the input rate, the maximum amount of traffic that can be sustained within this bursty duration T is expressed as in equation (6.13).

$$\text{Max Burst size} = \lambda T = B + rT \tag{6.13}$$

From equation (6.13), the maximum bursty duration can be calculated as in equation (6.14).

$$T = \frac{B}{\lambda - r} \tag{6.14}$$

6.5 Conclusions

In this chapter, we have covered the principles for supporting data transport from a process to another process across an internetworking environment. First, an introduction is presented to provide an overview of data transport between application processes through the Internet to highlight the important issues that may occur to them. Then, based on these issues, the set of services and features that should be available to facilitate process-to-process data transfer are presented and discussed. These features include process multiplexing/de-multiplexing, data reordering, connection-oriented data communication, reliable data transfer, timely data delivery, flow control, and congestion control.

Next, we turned our focus on the materialization of these features in commonly used protocols in the Internet. The first protocol that we studied is UDP, a simple and easy protocol that does only data multiplexing/de-multiplexing and error detection. First, we studied the motivations and benefits of UDP despite its lack of features and then examined its header format. Then, we looked at how process multiplexing/de-multiplexing can be facilitated with UDP in specific and at the transport layer in general. The operation of UDP serves as a preamble for us to explore the more complicated and feature-rich TCP. Its offerings include connection-oriented data transfer, reliable and in-order data deliver, flow control and congestion control beside the basic multiplexing/de-multiplexing and error detection services in UDP. Similar to UDP, we started our TCP journey with a tour on its header fields. After understanding the header fields in TCP, we took a deeper dive into the operations of its features including connection-oriented, reliable data transfer and reordering, flow control and congestion control.

Finally, we closed the chapter with a survey on common techniques for traffic management, including traffic policing and shaping. To this end, we explored the two methods to implement traffic management, namely leaky bucket and token bucket. In each approach, its operations, usages and characteristics are described.

References

[1] RFC 3550; H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications". 2003.

[2] IANA Service Name and Transport Protocol Port Number Registry - https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

Review questions

1. IP can transfer packets between any two hosts in the Internet, why IP is not adequate for delivering data between the two application processes?
2. Assume that all the links from the source to the destination hosts are reliable and can guarantee in-order frame transfer, illustrate two cases in which the correctness of end-to-end data delivery cannot be achieved with only the mechanisms at the data link level.
3. What is the size of User Datagram Protocol (UDP) header in bits? Specify all four fields with their sizes inside UDP header format.

4. Compare User Datagram Protocol (UDP) and Transmission Control Protocol (TCP)? Provide at least three properties of each protocol.
5. Illustrate two applications in which it is preferable to use UDP instead of the full-featured TCP?
6. What is the difference between port and socket?
7. Briefly explain all steps for three-way handshake mechanism in TCP for connection setup.
8. According to the TCP header format, what is the size of flags in bits? Explain four fields of flags and their functions.
9. Which transport layer protocol can be used for the following use cases: packet voice, file transfer, remote login?
10. Which additional features are provided in UDP in comparison to IP?
11. Briefly describe fast retransmit mechanism. Explain its effect for reliable data transfer.
12. What is the difference for round trip time (RTT) in data link layer and transport layer?
13. Suppose that instead of using AIMD, an alternative scheme named Additive Increase Additive Decrease (AIAD) is used. In this scheme, when the participate hosts encounter a packet loss, they decrease their window sizes by a fixed number. Suppose that 2 users share the same link, draw the bandwidth evolution of the two users in this case. Is AIAD provide fairness between the two users?
14. What are the names of two commonly used mechanisms for traffic management? Briefly explain them.
15. Explain the leaky bucket model in the transport control to manage the traffic flow.

Problems

1. Considering a 512 kbps link between Canada and France, the link throughput is 282 kbps with a 256ms round-trip time. Moreover, the link throughput decreases to 60 kbps with a 600ms round-trip time, when it is routed over a satellite. For both links, a fixed window size is used and both of the links are error-free.
    a. Find the utilization and the window size of the two links.
    b. What is the largest window size for the first link in bytes?
    c. What is the largest window size for the satellite link in bytes?
    d. Briefly discuss about the window sizes of those two links. What are the advantages or disadvantages of the satellite link?
2. For a given window size of 64kB, the link round-trip time between two hosts is estimated as 250ms.
    a. Calculate the maximum throughput for this scenario in bps.
    b. If the minimum required throughput is 4Mbps, how can it be achieved?
3. In a TCP session, the average round-trip time between two hosts is estimated as 60ms.
    a. Briefly express TCP round-trip time.

   b. For α=0.8, as the weight of the running average, calculate the estimated round-trip time for the following instantaneous round-trip time values as 50ms, 130ms, 20ms.
   c. For α=0.2, recalculate the estimated round-trip time for the given values in question b.
   d. In general, what is the range of α? How does α affect the estimation of round-trip time?
4. In a TCP session, the maximum segment size is 8kB, the round trip time is 10ms and the current congestion window is 512KB.
   a. If a packet loss is encountered and no congestion is experienced hereafter, how long would it take for the congestion window to grow back to 512kB?
   b. If a 3 duplicate ACKs is encountered and no congestion is experienced hereafter, how long would it take for the congestion window to grow back to 512kB when TCP Reno is used? When TCP Tahoe is used?
   c. Suppose that the maximum congestion window of the sending host is 512kB, what is the maximum achievable throughput of this host?
5. The leaky bucket mechanism is employed as a traffic management technique. Assume that maximum outgoing traffic is 2 packets/second. The bucket is empty at t=0. At the time instances t=1,2,4,8,12,17, the bucket receives 4 packets.
   a. Considering the infinite bucket size, plot the number of packets inside the bucket versus time from t=1 to t=20.
   b. How long does it take to send all the packets inside the bucket?
   c. For the current transmission scenario, what is the minimum required bucket size? Why? Please briefly explain it.
6. The leaky bucket mechanism is employed as a traffic management technique. Assume two incoming traffic links to the leaky bucket. The first link has the constant rate of 2 packets/second. Moreover, at the time instances t=1,3,4,5,10, the bucket receives 5 packets from the second link. The maximum output traffic rate is also calculated as 5 packets/second. The bucket is empty at t=0.
   a. Considering the infinite bucket size, plot the number of packets inside the bucket versus time from t=1 to t=15. For the current transmission scenario, what is the minimum required bucket size? Why? Please briefly explain it.
   b. When the incoming traffic rate is increased to 3 packets/second, redo question (a).
   c. If the bucket size is chosen as 13 packets for the question (b), plot the number of packets inside the bucket versus time from t=1 to t=15. Is there any difference in question (b) and question (c)? If yes, please briefly explain it.
7. Considering the leaky bucket mechanism as a traffic management technique, the leaky bucket receives two incoming traffic links. The first link has a constant rate of 3 packets/second. Moreover, the bucket receives 7 packets at t=1,3,5,8,12 from the second link. The bucket size is 12 packets, which is empty at t=0.

   a. When the maximum output rate is 7 packets/second, plot two graphs: (1) the output rate versus time and (2) the number of packets inside the bucket versus time from t=1 and t=20
   b. When the maximum output rate is decreased to 5 packets/second, redo question (a).
   c. In order to prevent any packet loss in question (b), find the maximum number of packets that can be delivered to the bucket at the same time instances. Please verify your answer by re-plotting two graphs expressed in question (a).

8. For the token bucket algorithm, the bucket size is B packet, the periodic token regeneration rate is r packets/second and the traffic rate is λ packets/second. Considering λ>r, the burst duration is denoted as T seconds,
   a. Derive the maximum burst size.
   b. Find the expression of the maximum burst duration in terms of B, r, λ.
9. Suppose the bucket is initially full for the token bucket algorithm. The input traffic between t=1 and t=8 is given in the table below, where the number of packets sent at the corresponding time instances is indicated. The system is designed to avoid any delay for the incoming packets.
   a. What is the minimum bucket size for this transmission, when token rate is 3 packets/second?
   b. What is the minimum bucket size for this transmission, when token rate is 4 packets/second?
   c. Compare the results obtained in question (a) and (b).
10. A bucket is initially full for the token bucket algorithm. The input traffic between t=1 and t=5 is given in the table below, where the number of packets sent at the corresponding time instances is indicated. When the periodic token regeneration rate is r packets/second, find the necessary bucket size B as a function of r. Please note that r can be only positive integer values.

Time (Seconds)	1	2	3	4	5
Number of packets received	3	4	6	0	2

11. Consider a token bucket scheme with the bucket size of 500KB, the token rate of 4MB/s and the incoming traffic rate of λ MB/s. This incoming traffic rate may vary between 5MB/s and 25MB/s. Plot the maximum burst size and maximum burst duration with respect to the varying incoming traffic rates. Briefly discuss the results.
12. At t=0, the average round-trip time in a TCP session is initially estimated is as 200ms between two hosts. The instantaneous round-trip time values are respectively 150ms, 300ms, 120ms, 400ms, 280ms at t=1, 2, 3, 4, 5.
   a. For α=0.9, calculate and plot the estimated round-trip time for t=1, 2, 3, 4, 5.
   b. For α=0.1, calculate and plot the estimated round-trip time for t=1, 2, 3, 4, 5.
   c. Compare the estimated round-trip time values in question (a) and (b).

13. For a token bucket scheme, the maximum burst size is given as 5MB, when the bucket size is 3.2MB and the incoming traffic rate of 56 Mbps. Find the maximum burst duration and token rate in Mbps.
14. Given a constant incoming traffic, compare the following traffic management schemes: (i) the leaky bucket algorithm, (ii) the token bucket algorithm starting with a bucket full of tokens and (iii) the token bucket algorithm starting with empty bucket.
15. Different from UDP, TCP is a connection-oriented protocol. Therefore, the two TCP endpoints must establish a connection before transmitting any data. When data transmission finishes, they must terminate this connection. Draw the connection setup and connection termination in TCP.