T

TRANSPORT LAYER

INTRODUCTION

The Internet has evolved into an extremely large complex system and has changed many important aspects of our lives. Like any complex engineering system, the design of the Internet is carried out in a modular way, where each main functional module is called a "layer." One of the layering structures often used is the five-layer model consisting of the physical layer, the link layer, the network layer, the transport layer, and the application layer.[1] See Fig. 1 for a simple illustration.

The sending and receiving computers each run analogous stacks, with data being passed down the stack from the sending application, and then up the receiver's stack to the receiving application. The physical layer is the part that actually moves information from one place to another, such as by varying voltages on wires or generating electromagnetic waves. The application layer is the part with which users interact, such as the hypertext transport protocol (HTTP) used to browse the Web, or the simple mail transfer protocol (SMTP) used to send e-mail.

Each layer consists of *protocols* to specify such things as the data format, the procedure for exchanging data, the allocation of resources, and the actions that need to be taken in different circumstances. This protocol can be implemented in either software or hardware or both. This article concerns transport layer protocols and their associated algorithms, mainly focusing on the wireline Internet but also discussing some other types of networks such as wireless ones.

The transport layer manages the end-to-end transportation of packets across a network. Its role is to connect application processes running on end hosts as seamlessly as possible, as if the two end applications were connected by a reliable dedicated link, thus making the network "invisible." To do this, it must manage several nonidealities of real networks: shared links, data loss and duplication, contention for resources, and variability of delay. By examining these functionalities in turn, we will provide a brief introduction to this important layer, including its functions and implementation, with an emphasis on the underlying ideas and fundamentals. We will also discuss possible directions for the future evolution of the transport layer and suggest some further reading.

MULTIPLEXING: MANAGING LINK SHARING

One basic function of the transport layer is *multiplexing* and *demultiplexing*. Usually there are multiple application processes running on one host. For example, a computer may be sending several files generated by filling in web forms, while at the same time sending e-mails. The network layer only cares about sending a stream of data out of the computer. Therefore, the transport layer needs to aggregate data from different applications into a single stream before passing it to the network layer. This is called multiplexing. Similarly, when the computer receives data from the outside, the transport layer is again responsible for distributing that data to different applications—such as a web browser or e-mail client—in a process called demultiplexing. Figure 1 also shows the data directions for multiplexing (sending) and demultiplexing (receiving).

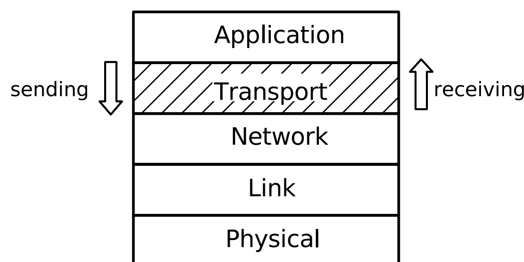

Figure 1. Internet protocol stack.

Multiplexing is achieved by dividing flows of data from the application into (one or more) short *packets*, also called *segments*. Packets from different flows can then be interleaved as they are sent to the network layer. Demultiplexing is achieved by allocating each communication flow a unique identifier. The sender marks each packet with its flow's identifier, and the receiver separates incoming packets into flows based on their identifiers. In the Internet, these identifiers consist of transport layer *port* numbers, and additionally the network layer addresses of the sender and receiver and a number identifying the transport layer protocol being used. Even before a flow is started, a "well-known" port number can be used to identify which process on the receiver the sender is attempting to contact; for example, Web servers are "well known" to use port 80.

Multiplexing and demultiplexing is one of the most fundamental tasks of the transport layer. Other functions are required by some applications but not by all, and so different transport layer protocols provide different subsets of the possible services. However, essentially all transport layer protocols perform at least multiplexing and demultiplexing.

There are two dominant types of transport layer protocol used in the Internet. One is UDP (User Datagram Protocol), and the other is TCP (Transmission Control Protocol). The former provides unreliable and connectionless service to the upper layers, whereas the latter generates reliable and connection-based service.

[1]There are also some other ways of defining these layers; e.g., the standard OSI (open systems interconnection) reference model defines seven layers with the session layer and the presentation layer added.

Wiley Encyclopedia of Computer Science and Engineering, edited by Benjamin Wah.
Copyright © 2008 John Wiley & Sons, Inc.

UDP: Multiplexing Only

UDP is a very simple protocol. It is called connectionless because all UDP packets are treated independently by the transport layer, rather than being part of an ongoing flow.[2] Besides minor error checking, UDP essentially only does multiplexing and demultiplexing. It does not provide any guarantee that packets will be received in the order they are sent, or even that they will be received at all. It also does not control its transmission rate. In fact, the rationale behind the design of UDP is to let applications have more control over the data sending process and to reduce the delay associated with setting up a connection. These features are desirable for certain delay-sensitive applications such as streaming video and Internet telephony. In general, applications that can tolerate certain data loss/corruption but are sensitive to delay would choose to use UDP.

TCP: Reliable Connection-Oriented Transport

In contrast to UDP, TCP provides a connection-oriented service, which means that it sends data as a stream of related packets, making concepts such as the order of packets meaningful. In particular, TCP provides reliable service to upper layer applications, ensuring that the packets are correctly received and in the order in which they are sent.

At the start of a connection, TCP uses a *three-way handshake* to establish a connection between sender and receiver, in which they agree on what protocol parameters to use. This process takes 1.5 round-trip times (one side sends a SYNchronize packet, the other replies with a SYN and an ACKnowledge packet, and the first confirms with an ACK), which is an overhead avoided by UDP.

TCP receives data from the application as a single stream, e.g., a large file, and segments it into a sequence of packets. It tries to use large packets to minimize overhead, but there is a maximum size that the network can carry efficiently, called the MTU (maximum transfer unit). TCP is responsible for choosing the correct size, in a process called *path MTU discovery*. In contrast, UDP is given data already segmented into packets, and so it is the application's responsibility to observe MTU restrictions.

TCP is the most common transport protocol in the Internet; measurements show that it accounds for about 80% of the traffic (1). Applications such as file transmission, Web browsing, and e-mail use it partly because of its ability to transfer continuous streams of data reliably, and partly because many firewalls do not correctly pass other protocols. The following two sections, on *reliable transmission* and *congestion control*, describe in greater detail the main features of TCP.

RELIABLE TRANSMISSION: MANAGING LOSS, DUPLICATION, AND REORDERING

When the underlying network layer does not guarantee to deliver all packets, achieving reliable transmission on top of this unreliable service becomes an important task. Reasons for packet loss include transient routing loops, congestion of a resource, or physical errors that were not successfully corrected by the physical or link layer.

This problem is very similar to that faced by the link layer. The difference is that the link layer operates over a single unreliable physical link to make it appear reliable to the network layer, whereas the transport layer operates over an entire unreliable network to make it appear reliable to the application. For this reason, the algorithms employed at the transport layer are very similar to those employed at the link layer, and they will be reviewed briefly here.

ARQ (Automatic Repeat-reQuest) is the basic mechanism to deal with data corruption. When the receiver receives a correct data packet, it will send out a positive acknowledgment (ACK); when it detects an error, it will send out a negative acknowledgment (NAK).

Because physically corrupted packets are usually discarded by the link layer, the transport layer will not directly observe that a packet has been lost, and hence, many transport layer protocols do not explicitly implement NAKs. A notable exception is multicast transport protocols. Multicast protocols allow the sender to send a single packet, which is replicated inside the network to reach multiple receivers, possibly numbering in the thousands. If each sent an ACK for every packet, the receiver would be flooded. If instead receivers send NAKs only when they believe packets are missing, the network load is greatly reduced. TCP implicitly regards ACKs for three or more out-of-order packets (resulting in "duplicate ACKs") as forming a NAK.

An interesting problem immediately develops. If the sender only sends the next packet when it is sure the first one is received correctly, the system will be very inefficient. In that case, the sender would only be able to send one packet every round-trip time (the time it takes for a packet to reach the destination and the ACK to return), whereas the time taken to send a packet is usually much smaller than the round-trip time. This is especially true for today's high-speed networks. For example, consider a 1 Gbit/s (1,000,000,000 bits per second) connection between two hosts that are separated by 1500 km and therefore have a round-trip distance of 3000 km. Sending a packet of size 10 kbit takes only 10 μs (ten microseconds), whereas the round-trip time cannot be smaller than the physical distance divided by the light of speed (300,000 km per second), which in this case is 10 ms. In other words, in this case, the utilization of the sender is about 0.1%, which is clearly not acceptable. This is the motivation for the general sliding window algorithm that will be discussed in the following subsection.

Sliding Window Transmission Control

The basic idea here is again simple. The sender sends more packets into the network while it is waiting for the acknowledgment of the first packet. This certainly increases the utilization. On the other hand, it cannot send too much before receiving ACKs as that may heavily congest the network or overflow the receiver. In general, the sender is allowed to send no more than W packets into the network before receiving an acknowledgment. Here W is called the *window* size. If $W = 1$, it reverts to the inefficient transmission previously discussed. Since the sender can send W

[2] It is common for the application layer to implement flows on top of UDP, but that is not provided by UDP itself.

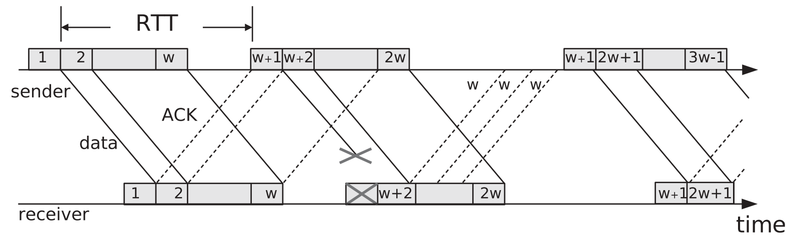

Figure 2. Sliding window flow control, with packet $W + 1$ being lost.

packets every round-trip time, the utilization is increased by a factor of W if the network and receiver can handle the packets. If not, then there will be congestion and the value of W must be reduced, which is the topic of the next section. The left of Fig. 2 shows the case when packets are successfully received. It shows that using $W > 1$ allows more than one packet to be sent per round-trip time (RTT), increasing the throughput of the connection.

However, things become a little more complex when $W > 1$ as now the sender needs to keep track of acknowledgments from more than one packet in one round-trip time. This is done by giving every packet a *sequence number*. When a packet is received, the receiver sends back an ACK carrying the appropriate sequence number.

When packets are received out of order, the receiver has three options, depending on the protocol. It may simply discard packets that are received out of order, it may forward them immediately to the application, or it may store them so that it can deliver them in the correct order once the missing packet has been received, possibly after being resent. Finally, if the sender has not received an ACK for a particular packet for a certain amount of time, a *timeout* event occurs. After that, the sender resends all packets that are sent out but have not yet been acknowledged.

Realization

The following example presents a simplified example of how the above ideas are realized by TCP, which is currently the most widespread example of sliding window transmission control.

Example 1. Rather than acknowledging each packet, TCP ACKs cumulatively acknowledge all data up until the specified packet. This increases the robustness to the loss of ACKs. Figure. 2 shows the operation of TCP when packet $W + 1$ is lost. Initially, the window spans from 1 to W, allowing the first W packets to be sent. The sender then waits for the first packet to be acknowledged, causing the window to slide to span packets 2 to $W + 1$, allowing the $W + 1$st packet to be sent. This continues until the second window is sent, and after sending the $2W$th packet, the sender again must pause for an ACK to slide the window along. However, this time, the $W + 1$st packet was lost, and so no ACK is received. When packet $W + 2$ is received, the receiver cannot acknowledge it, since that would implicitly acknowledge packet $W + 1$, which has not yet arrived. Instead, it sends another ACK for the most recently received packet, W. This is repeated for all subsequent arrivals, until $W + 1$ is received. When the sender receives the third duplicate ACK for W, it assumes that $W + 1$ was lost, and retransmits it. It then continues transmitting from where it left off, with packet $2W + 1$.

The precise response to packet loss of current TCP is more complex than in this example, because packet loss is treated as a signal that a link in the network is overloaded, which triggers a congestion control response, as described in the following section.

CONGESTION CONTROL: MANAGING RESOURCE CONTENTION

When transport layer protocols were first designed, they were intended to operate as fast as the receiver could process the data. The transport layer provided "flow control" to slow the sender down when the receiver could not keep up. However, in the 1980s, the Internet suffered from several famous *congestion collapses*, in which the sliding window mechanism was resending so many packets that the network itself because overloaded to the point of inoperability, even when the receivers were not overloaded.

Recall from the previous section that senders use $W > 1$ to increase the utilization of the network. Congestion occurred because flows sought to use more than 100% of the network capacity. As a result, a set of rules were proposed (2) for how senders should set their windows to limit their aggregate sending rate while maintaining an approximately fair allocation of rates.

Congestion control considers two important topics: what rates would we ideally like to allocate to each flow in a given network, and how can we achieve that in practice using only distributed control. The latter is made difficult because of the decentralized nature of the Internet: senders do not know the capacity of the links they are using, how many other flows share them, or how long those flows will last; links do not know what other links are being used by the flows they are carrying; and nobody knows when a new flow will arrive. Figure 3 shows an example in which two flows each use three links, of which they share one.

Let us now consider the current solutions to the problem of implementing congestion control in a scalable way, and then examine the other problem of deciding what rate allocation is more desirable.

Existing Algorithms

There are two main phases of a congestion control algorithm: *slow start* and *congestion avoidance*, punctuated by

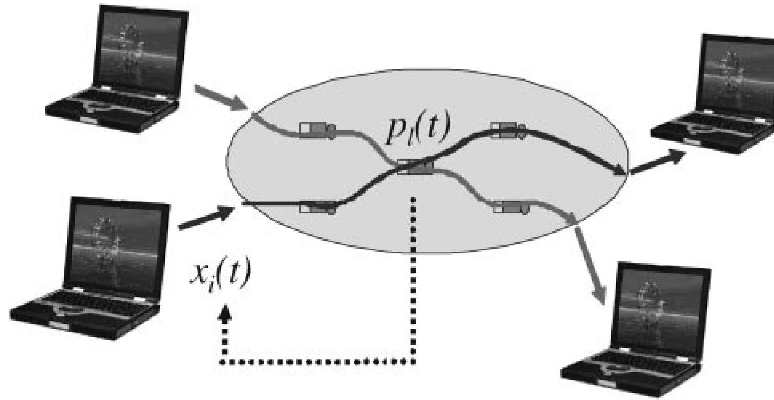

Figure 3. Two flows sharing a link, and also using nonshared links.

short periods of retransmission and loss recovery. We now introduce both using the standard TCP congestion control algorithm, commonly called TCP Reno.[3]

When a TCP connection begins, it starts in the slow start phase with an initial window size of two packets. This results in a slow initial transmission, giving rise to the name. It then rapidly increases its sending rate. It doubles its window every round-trip time until it observes a packet loss, or the window reaches a threshold called the "slow start threshold." If a loss occurs, the window is then halved, and in either case, the system enters the congestion avoidance phase. Note that the sender increases its transmission rate exponentially during the slow start.

In the congestion avoidance phase, the sender does what is known as Additive Increase Multiplicative Decrease (AIMD) adjustment. This was first proposed by Chiu and Jain (3) as a means to obtain fair allocation, and implemented in the Internet by Jacobson (2). Every round-trip time, if all packets are successfully received, the window is increased by one packet. However, when there is a loss event, then the sender will halve its window. Because large windows are reduced by more than small windows, AIMD tends to equalize the size of windows of flows sharing a congested link (3). Finally, if a timeout occurs, the sender will start from slow start again. Figure 4 shows how the window evolves along time in TCP Reno. Importantly, TCP Reno uses packet loss as congestion indication.

In summary, the basic engineering intuition behind most congestion control protocols is to start probing the network with a low transmission rate, quickly ramp up initially, then slow down the pace of increase, until an indicator of congestion occurs and transmission rate is reduced. Often packet loss or queueing delay (4) are used as congestion indicators, and packet loss events are in turn inferred from local measurements such as three duplicated acknowledgments or timeout. These design choices are clearly influenced by the views of wireline packet-switched networks, in which congestion is the dominant cause of packet loss. The choice of the ramp-up speed and congestion

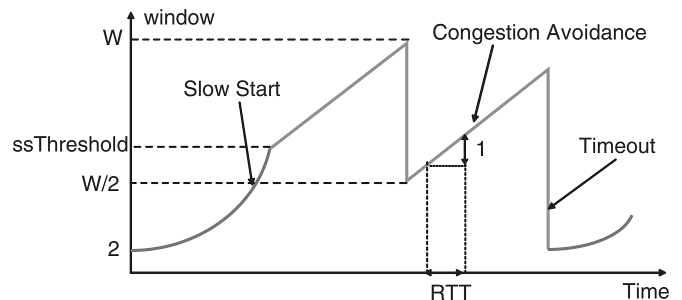

Figure 4. TCP Reno Window Trajectory.

indicators have mostly been based on engineering intuition until recent developments in predictive models of congestion control have helped with a more systematic design and tuning of the protocols.

This window adaptation algorithm is combined with the sliding window transmission control, to form the whole window-based congestion control mechanism, as illustrated in Fig. 5. The transmission control takes two inputs, the window size and the acknowledgments from the network. The window size is controlled by the congestion control algorithm such as TCP Reno, which updates the window based on the estimated congestion level in the network. In summary, with window-based algorithms, each sender controls its window size—an upper bound on the number of packets that have been sent but not acknowledged. As pointed out by Jacobson (2), the actual rate of transmission is controlled or "clocked" by the stream of received acknowledgments (ACKs). A new packet is

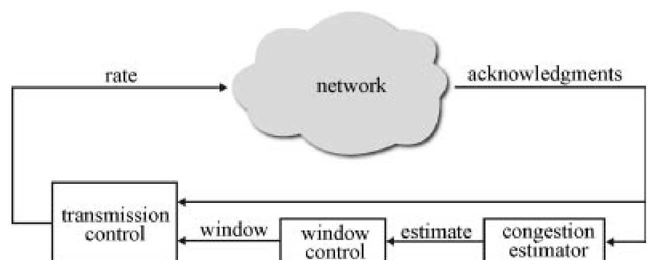

Figure 5. Window-based congestion control.

[3]Most systems actually implement a variant of Reno, typically NewReno, since Reno performs poorly when two packets are lost in a single round trip. However, the differences do not affect the descriptions in this section, and so we use the term "Reno."

transmitted only when an ACK is received, thereby ideally keeping the number of outstanding packets constant and equal to the window size.

Theoretical Foundation

As mentioned, congestion control is essentially a resource allocation scheme that allocates the capacities of links to TCP flows. It is desirable to be able to calculate the share of the capacity and discuss its properties, such as the fairness of the allocation.

Many valuable models of TCP have been proposed. Since Kelly's work in the late 1990s, generic congestion control protocols have been modeled as distributed algorithms maximizing the total benefit obtained by the applications (5–8). This "reverse-engineering" approach shows that existing TCP congestion control mechanisms are implicitly solving an underlying global optimization, with an interpretation of link-price-based balancing of bandwidth demand by the end users. Following economic terminology, the user objective being maximized is called the *utility*, and the utility that the ith flow obtains by sending at a rate x_i is denoted $U_i(x_i)$. If each flow i uses a set of links $L(i)$ and link $l \in L(i)$ has capacity c_l, then the problem of maximizing the utility can be expressed as follows:

$$\max_{x \geq 0} \sum_i U_i(x_i)$$

$$\text{subject to} \sum_{i:l \in L(i)} x_i \leq c_l$$

This is a convex optimization problem provided that the utility functions follow the usual "law of diminishing returns," that is, the utility increases as the rate received increases, but the incremental benefit becomes smaller. Such problems have a very rich mathematical structure.

The theory of Lagrange duality for convex optimization allows the problem to be decomposed into subproblems in which each flow *independently* chooses its rate based on congestion signals from the links, such as packet loss or queueing delay, which are computed based only on local information. Again following economic terminology, these congestion signals are sometimes referred to as *prices*.

The strict convexity structure also implies that the optimal rates are unique, and that those rates are independent of many properties of the links, such as their buffer sizes. In particular, as long as the congestion signal is zero when the sum of the rates through the link is less than its capacity, it does not matter how the congestion signals are calculated; the equilibrium rates will depend only on the utility functions, which are in turn determined by the TCP algorithm at the sender.

The choice of utility function determines the notion of fairness implemented by the network (9). If the utility function is almost linear, it reflects only slightly diminishing returns as the transmission rate is increased, and the network will seek to maximize the sum of the rates of all flows, with no regard to fairness. At the opposite extreme, if the incremental benefit decreases rapidly, the utility function will be very concave and *max–min* sharing is achieved. The max–min rate allocation is the one in which no flow can increase its rate, except by reducing the rate of a flow that already has a lower rate. This is often seen as the fairest way to allocate rates.

A logarithmic utility function results in a compromise between fairness and throughput known as proportional fairness. Similarly, to a first approximation, the utility of the AIMD algorithm used by TCP Reno is as follows:

$$U_i(x_i) = -\frac{1}{x_i \tau_i^2}$$

where τ_i is the round-trip time of the flow i. This is similar to proportional fairness, but it tends slightly toward improving fairness at the expense of throughput, as will be seen in the following example.

Example 2. Consider a network with two congested links, each with a capacity of c. One flow uses both links, and each link also carries a single-link flow, as shown in Fig. 6.

The maximum sum of rates is achieved when $x_1 = x_2 = c$ and $x_3 = 0$, which maximizes the sum of utilities if $u_i(x_i)$ is approximately proportional to x_i. This is clearly unfair since the two-link flow cannot transmit at all. In contrast, the max–min rates are $x_1 = x_2 = x_3 = c/2$, which maximizes the sum of utilities if $u_i(x_i)$ rises very sharply for $x_i < c/2$, and rises only very slightly for $x_i > c/2$. This is completely fair in that all flows receive the same rate, but it is unfair in the sense that the long flow causes twice as much congestion but still achieves the same rate. In this case, the total rate has reduced from $c + c = 2c$ to $c/2 + c/2 + c/2 = 1.5c$.

The proportional-fair rates, which maximize logarithmic utilities, are $x_1 = x_2 = 2c/3$ and $x_3 = c/3$. These rates are in the ratio 2:1 because the resources consumed are in the ratio 1:2, and they give a total rate of around $1.67c$. If all flows have equal round-trip times τ_i, TCP Reno will give average rates in ratio $1 : \sqrt{2}$, namely $x_1 = x_2 = \sqrt{2}c/(1 + \sqrt{2})$ and $x_3 = c/(1 + \sqrt{2})$, with a total throughput of $1.59c$. The fact that the rates are more similar for Reno than for proportional fairness, but the sum of rates is lower, supports the statement that Reno is a compromise between proportional fairness and max–min fairness.

In concluding this subsection, we mention that there has been a lot of recent research on both reverse-engineering and forward-engineering congestion control protocols, based on the above mathematical model and its variants. Some of the ongoing research issues will be briefly presented toward the end of this entry.

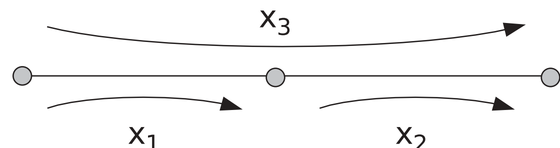

Figure 6. A two-link network shared by three flows.

TIMING RESTORATION: MANAGING DELAY VARIATION

In most cases, it is desirable for the transport layer to pass data to the receiving application as soon as possible. The notable exception to this is streaming audio and video. For these applications, the temporal spacing between packets is important; if audio packets are processed too early, the sound becomes distorted. However, the spacing between packets gets modified when packets encounter network queueing, which fluctuates in time. In its role of hiding lower layer imperfections from the upper layers, it is up to the transport layer to reestablish the timing relations between packets before sending them to the application.

Specialist transport protocols such as the Real Time Protocol (RTP) are used by flows requiring such timing information. RTP operates on top of traditional transport layer protocols such as UDP and provides each packet with a timestamp. At the receiver, packets are inserted as soon as they arrive into a special buffer known as a jitter buffer, or playout buffer. They are then extracted from the jitter buffer in the order in which they were sent and at intervals exactly equal to the interval between their timestamps. Jitter buffers can only add delay to packets, not remove delay; if a packet is received with excessive delay, it must simply be discarded by the jitter buffer. The size of the jitter buffer determines the tradeoff between the delay and packet loss experienced by the application.

TCP can itself cause delay fluctuation, both through ACK-clocking and the fluctuation in rate induced by Reno-like congestion control. When transmitting video and other streaming data, it is sometimes desirable to have packets sent with more uniform spacing. The burstiness caused to ACK-clocking can be avoided by *paced* TCP. Rather than sending packets exactly when acknowledgments are received, paced TCP sends one window's worth of packets uniformly spaced throughout a round-trip time. Many congestion control algorithms have been proposed that dispense with Reno's AIMD, reducing burstiness on longer timescales; notable examples include TCP Vegas and TCP Friendly Rate Control (TFRC).

RECENT AND FUTURE EVOLUTION

With the Internet expanding to global scale and becoming ubiquitous, it is encountering more and more new environments. On the one hand, the TCP/IP "hourglass model"[4] has been very successful at separating applications from the underlying physical networks and enabling the Internet's rapid growth. On the other hand, some basic assumptions are becoming inaccurate or totally invalid, which therefore imposes new challenges. This section describes some of the hot issues in both the Internet Engineering Task Force (IETF, the primary Internet standards body) and the broad research community. Many topics touch on both implementation issues and fundamental questions. We will start with the most implementation related ones and then progress to the more theoretical ones. It is certainly clear that the list below cannot be exhaustive and instead reflects the authors' taste and expertise. For example, many more variants of TCP congestion control are proposed in the last few years than can be surveyed within an encycopedia. In addition to the rest of this section, there are many other exciting developments in the theory and practice of transport layer design for future networks.

Protocol Enhancement

1) Datagram Congestion Control Protocol: Although TCP provides reliable in-order data transfer and congestion control, UDP provides neither. However, applications such as video transmission should implement congestion control, but they do not need guaranteed transmission. Moreover, they cannot tolerate the delay caused by retransmission and in-order delivery. Consequently, the IETF has developed a new protocol called DCCP (Datagram Congestion Control Protocol), which can be viewed either as UDP with congestion control or as TCP without the reliability guarantees. Because many firewalls block unknown protocols, DCCP has not yet been widely used, although it is implemented in many operating systems.

2) Multiple indicators of congestion: The current TCP NewReno relies primarily on detection of packet loss to determine its window size. Other proposals have been made that rely primarily on estimates of the queueing delay. The utility maximization theory applies to networks in which all flows are of the same "family." For example, all flows in the network may respond solely to loss; different flows may respond differently to loss, provided that loss is the only congestion signal they use. However, when a single network carries flows from both the "loss" and "delay" families, or flows responding to other "price" signals such as explicit congestion signals, the standard theory fails to predict how the network behaves.

Unlike networks carrying a single family of algorithms, the equilibrium rates now depend on router parameters, such as buffer sizes, and flow arrival patterns. The equilibrium can be nonunique, inefficient, and unfair. The situation is even more complicated when some individual flows respond to multiple congestion signals, such as adjusting AIMD parameters based on estimates of queueing delay. This has motivated recent efforts to construct a more general framework, which includes as a special case the theory for networks using congestion signals from a single family (10).

Applications

1) Delay-tolerant networks: Sliding window protocols rely on feedback from the receiver to the sender. When communicating with spacecraft, the delay between sending and receiving may be minutes or hours, rather than milliseconds, and sliding windows become infeasible. This has lead to research into "interplanetary TCP." Technology called DTN (Delay-Tolerant Networking) is being developed for this, and also for more mundane situations in which messages suffer long delays. One example is in vehicular networks, in which messages are exchanged over short-range links as vehicles pass one another, and

[4]It is called an hourglass because a small number of simple network and transport layer protocols connect a large variety of complex application layer protocols above with a large variety of link and physical layer protocols below.

are physically carried by the motion of the vehicles around a city. In such networks, reliability must typically be achieved by combinations of error correcting codes and multipath delivery (e.g., through flooding).

2) Large bandwidth delay product networks: In the late 1990s, it became clear that TCP NewReno had problems in high speed transcontinental networks, commonly called "large bandwidth-delay product" or "large BDP" networks. The problem is especially severe when a large BDP link carries only a few flows, such as those connecting supercomputer clusters. In these networks, an individual flow must have a window W of many thousands of packets. Because AIMD increases the window by a single packet per round trip, the sending rate on a transatlantic link will increase by around 100 kbit/s. It would thus take almost three hours for a single connection to start to use fully a 1 Gbit/s link.

Many solutions have been proposed, typically involving increasing the rate at which the window is increased, or decreasing the amount by which it is decreased. However, these both make the algorithm more "aggressive," which could lead to allocating too much rate to flows using these solutions and not enough to flows using the existing TCP algorithm. As a result, most solutions try to detect whether the network actually is a "large BDP" network, and adjust their aggressiveness accordingly. Another possibility is to avoid dealing with packet loss in large BDP networks. Researchers have developed various congestion control algorithms that use congestion signals other than packet loss, e.g., queueing delay. Many proposals also seek to combine timing information with loss detection. This leads to the complications of multiple indicators of congestion described previously.

An alternative that is often proposed is for the routers on the congested links to send explicit messages indicating the level of congestion. This was an important part of the available bit-rate (ABR) service of asynchronous transport mode (ATM) networks. It may allow more rapid and precise control of rate allocation, such as the elimination of TCP's time-consuming slow start phase. However, it presents significant difficulties for incremental deployment in the current Internet.

3) Wireless networks: Wireless links are less ideal than wired links in many ways. Most importantly, they corrupt packets because of fading and interference, either causing long delays as lower layers try to recover the packets, or causing packets to be lost. The first of these results in unnecessary timeouts, forcing TCP to undergo slow start, where as the latter is mistaken for congestion and causes TCP NewReno to reduce its window. Again, many solutions have been proposed. Some mask the existence of loss, where as others attempt to distinguish wireless loss from congestion loss based on estimates of queueing delay or explicit congestion indication.

The fundamental task of resource allocation is also more challenging in wireless networks, partly because resources are more scarce and users may move, but more importantly because of the interaction between nearby wireless links. Because the capacity of a wireless link depends on the strength of its signal and that of interfering links, it is possible to optimize resource allocation over multiple layers in the protocol stack. This cross-layer optimization generalizes the utility maximization. It provides challenges as well as opportunities to achieve even greater performance, which requires a careful balance between reducing complexity and seeking optimality.

Research Challenges

1) Impact of network topology: Transport layer congestion control and rate allocation algorithms are often studied in very simple settings. Two common test networks are *dumbbell* networks in which many flows share a single congested link, and *parking lot* networks, consisting of several congested links in series with one flow traversing all links, and each link also being the sole congested link for another short flow. Figure 6 shows a two-link parking lot network. These are used partly because they occur frequently in the Internet (such as when a flow is bottlenecked at the ingress and egress access links), and partly because there are intuitive notions of how algorithms "should" behave in these settings. However, these simple topologies often give a misleading sense of confidence in our intuition. For example, in parking lot topologies, algorithms that give a high rate to the single link flows at the expense of the multilink flow achieve higher total throughput, and thus it is widely believed that there is a universal tradeoff between fairness and efficiency. However, networks exist in which increasing the fairness actually increases the efficiency (11). This and other interesting and counter-intuitive phenomena develop only in a network setting where sources interact through shared links in intricate and surprising ways.

2) Stochastic network dynamics: The number of flows sharing a network is continually changing, as new application sessions start, and others finish. Furthermore, packet accumulations at each router is shaped by events in all upstream routers and links, and packet arrivals in each session are shaped by the application layer protocols, including those in emerging multimedia and content distribution protocols. Although it is easy to study the effects of this variation by measuring either real or simulated networks, it is much harder to capture these effects in theoretical models. Although the deterministic models studied to date have been very fruitful in providing fundamental understanding of issues such as fairness, there is an increasing interest in extending the theoretical models to capture the stochastic dynamics occurring in real networks.

As an example of one type of these dynamics, consider a simple case of one long flow using the entire capacity of a given link, and another short flow that starts up using the same link. If the short flow finishes before the long flow does, then the finish time of the long flow will be delayed by the size of the short flow divided by the link capacity, *independent* of the rate allocated to the short flow, provided that the sum of their rates is always the link capacity. In this case, it would be optimal to process the flows in "shortest remaining processing time first" (SRPT) order; that is, to allocate all rate to the short flow and meanwhile totally

suspend the long flow. However, as the network does not know in advance that the short flow will finish first, it will instead seek to allocate rates fairly between the two flows. This can cause the number of simultaneous flows to be much larger than the minimum possible, resulting in each flow getting a lower average rate than necessary. The fundamental difficulty is that the optimal strategy is no longer to allocate instantaneous rates fairly based on the existing flows.

FURTHER READING

The transport layer is a main topic in many textbooks on computer networks, which is now a standard course in most universities. This article only seeks to provide a basic understanding of the transport layer. For those who are interested in digging into details and working in related areas, the following references are a useful starting point. For a complete introduction to computer networks including the transport layer, see any of the major networking textbooks such as Ref. 12. The Internet's main transport layer protocol, TCP, is described in detail in Ref. 13, although several details have evolved since that was written. For a general mathematical approach to understanding network layering, see a recent survey (14). Samples of early TCP congestion control analysis include Refs. 15–18. A survey on the mathematical treatment of Internet congestion control can be found in Ref. 19. Enduring issues are also well described in Ref. 20.

BIBLIOGRAPHY

1. M. Fomenkov, K. Keys, D. Moore, and K. Claffy, Longitudinal study of Internet traffic in 1998-2003, *WISICT '04: Proc. Winter Int. Symp. Info. Commun. Technol*, 2004.
2. V. Jacobson, Congestion avoidance and control, *Proc. ACM SIGCOMM*, 1988.
3. D. M. Chiu and R. Jain, Analysis of the increase and decrease algorithms for congestion avoidance in computer networks, *Computer Networks ISDN Sys.*, **17**(1): 1–14, 1989.
4. L. Brakmo and L. Peterson, TCP Vegas: End-to-end congestion avoidance on a global Internet, *IEEE J. Selected Areas Communi.***13**(8): 1465–80, 1995.
5. F. Kelly, A. Maoulloo, and D. Tan, Rate control for communication networks: Shadow prices, proportional fairness and stability. *J. Operational Research Society*, **49**: 237–252, 1998.
6. S. Kunniyur and R. Srikant, End-to-end congestion control schemes: Utility functions, random losses and ECN marks, *IEEE/ACM Trans. Networking*, **11**(5): 689–702, 2003.
7. S. Low, A duality model of TCP and queue management algorithms. *IEEE/ACM Trans. Networking*, **11**(4): 525–536, 2003.
8. S. Low and D. Lapsley, Optimization flow control—I: Basic algorithm and convergence, *IEEE/ACM Trans. Networking*, **7**(6): 861–874, 1999.
9. J. Mo and J. Walrand, Fair end-to-end window-based congestion control, *IEEE/ACM Trans. Networking*, **8**(5): 556–567, 2000.
10. A. Tang, J. Wang, S. Low and M. Chiang, Equilibrium of heterogeneous congestion control: Existence and uniqueness, *IEEE/ACM Trans. Networking*, **15**(4): 824–837, 2007.
11. A. Tang, J. Wang and S. H. Low, Counter intuitive throughput behaviors in networks under end-to-end control, *IEEE/ACM Trans. Networking*, **14**(2): 355–368, 2006.
12. J. Kurose and K. Ross, *Computer Networking*. Fourth edition, Addison Wesley, 2007.
13. W. R. Stevens, *TCP/IP Illustrated, Volume 1, The Protocols*. Upper Saddle River, NJ: Addison-Wesley, 1994.
14. M. Chiang, S. Low, A. Calderbank and J. Doyle, Layering as optimization decomposition: A mathematical theory of network architectures, *Proc. of the IEEE*, **95**(1): 255–312, 2007.
15. T. Lakshman and U. Madhow, The performance of TCP/IP for networks with high bandwidth-delay products and random loss, *IEEE/ACM Trans. on Networking*, **5**(3): 336–350, 1997.
16. M. Mathis, J. Semke, J. Mahdavi and T. Ott, The macroscopic behavior of the TCP congestion avoidance algorithm, *ACM Computer Communication Review*, **27**(3): 67–82, 1997.
17. J. Padhye, V. Firoiu, D. Towsley and J. Kurose, Modeling TCP throughput: A simple model and its empirical validation, *ACM Computer Communication Review*, **28**(4): 303–314, 1998.
18. K. Ramakrishnan and R. Jain, A Binary feedback scheme for congestion avoidance in computer networks with connectionless network layer, *Proc. ACM SIGCOMM*, 1988.
19. R. Srikant, *The Mathematics of Internet Congestion Control*, Cambridge, MA: Birkhauser, 2003.
20. J. Walrand and P. Varaiya, *High Performance Communication Networks*, San Francisco: CA, Morgan Kaufmann, 2000.

AO TANG
Cornell University
Ithaca New York

LACHLAN L. H. ANDREW
California Institute of Technology
Pasadena, California

MUNG CHIANG
Princeton University
Princeton New Jersey

STEVEN H. LOW
California Institute of Technology
Pasadena, California