Applied Sciences Faculty

Management Information Systems

Dr. Yakup BAKIŞ

Good morning everyone.
Last week, we talked about how the Internet works in general.

Today, we move one layer higher.
We will begin to understand the **Application Layer**,
which is the layer closest to the user.

When you open a website, send an email, or use Instagram, you are using the Application Layer.

This is the part of networking that you actually see and use every day.

Chapter 2

Application Layer

12.02.2026 2

Today I am going to present Chapter 2, which is the Application Layer. In computer networking, the application layer is the part closest to the user, because this is where network applications such as the Web, e-mail, messaging, and video services work. In this presentation, I will explain the main ideas of this chapter in a simple and organized way. I will talk about how applications communicate, what kinds of architectures they use, and what transport services they need. I will also introduce important protocols such as HTTP, SMTP, IMAP, and DNS. My goal is to show that network applications are the reason why computer networks are useful in daily life.

Application layer: overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

In this slide, I want to give a quick overview of the chapter. The chapter begins with the principles of network applications, so first we learn the general ideas behind how applications are designed and how they communicate over a network. After that, we study some of the most important Internet applications and protocols. These include the Web and HTTP, e-mail with SMTP and IMAP, and the Domain Name System, or DNS. The chapter also introduces peer-to-peer applications, video streaming, content distribution networks, and socket programming with both UDP and TCP. So this chapter is not only theoretical. It also connects concepts with real applications and implementation.

Application layer: overview

Our goals:

- conceptual *and* implementation aspects of application-layer protocols
  - transport-layer service models
  - client-server paradigm
  - peer-to-peer paradigm
- learn about protocols by examining popular application-layer protocols
  - HTTP
  - SMTP, IMAP
  - DNS
- programming network applications
  - socket API

This slide explains the main goals of the chapter. First, we want to understand the conceptual side of application-layer protocols. That means we study what applications need from the network and how protocols organize communication between processes. Second, we look at implementation aspects, especially transport-layer service models and the difference between the client-server paradigm and the peer-to-peer paradigm. Third, we learn important protocols by examining real examples such as HTTP, SMTP, IMAP, and DNS. Finally, we move from theory to practice by learning how to program network applications with sockets. So, the chapter helps us understand both how application protocols work and how we can build applications ourselves.

Some network apps

- social networking
- Web
- text messaging
- e-mail
- multi-user network games
- streaming stored video (YouTube, Hulu, Netflix)
- P2P file sharing
- voice over IP (e.g., Skype)
- real-time video conferencing
- Internet search
- remote login
- …

Q: *your* favorites?

Here we can see some common network applications. These examples show how the application layer is part of our daily lives. Social networking, the Web, text messaging, and e-mail are very familiar examples. We also have multi-user games, streaming video platforms such as YouTube or Netflix, peer-to-peer file sharing, voice over IP services like Skype, real-time video conferencing, Internet search, and remote login. The important point is that all of these applications run on top of the network and use communication protocols to exchange data. In other words, networks become valuable because they support useful applications. At the end, the slide asks about our favorites, because almost every user depends on application-layer services every day.

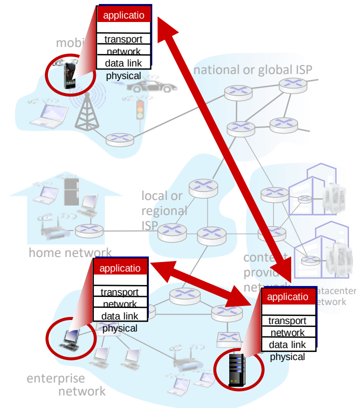
Creating a network app

write programs that:

- run on (different) end systems
- communicate over network
- e.g., web server software communicates with browser software

no need to write software for network-core devices

- network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation



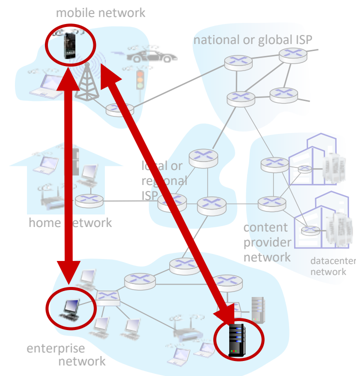
This slide explains what it means to create a network application. When we develop a network application, we write programs that run on different end systems and communicate with each other over the network. A simple example is a web browser communicating with a web server. Another example is a video application running on a user device and a server program running in a data center. A very important idea from the book is that application software is written for end systems, not for network-core devices such as routers or switches. Core devices work at lower layers, so they do not run user applications. This design makes application development faster and more flexible, because new services can be deployed by updating software at the edges of the network.

Client-server paradigm

server:
- always-on host
- permanent IP address
- often in data centers, for scaling

clients:
- contact, communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do *not* communicate directly with each other

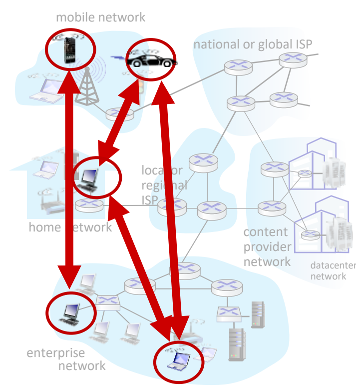
- examples: HTTP, IMAP, FTP



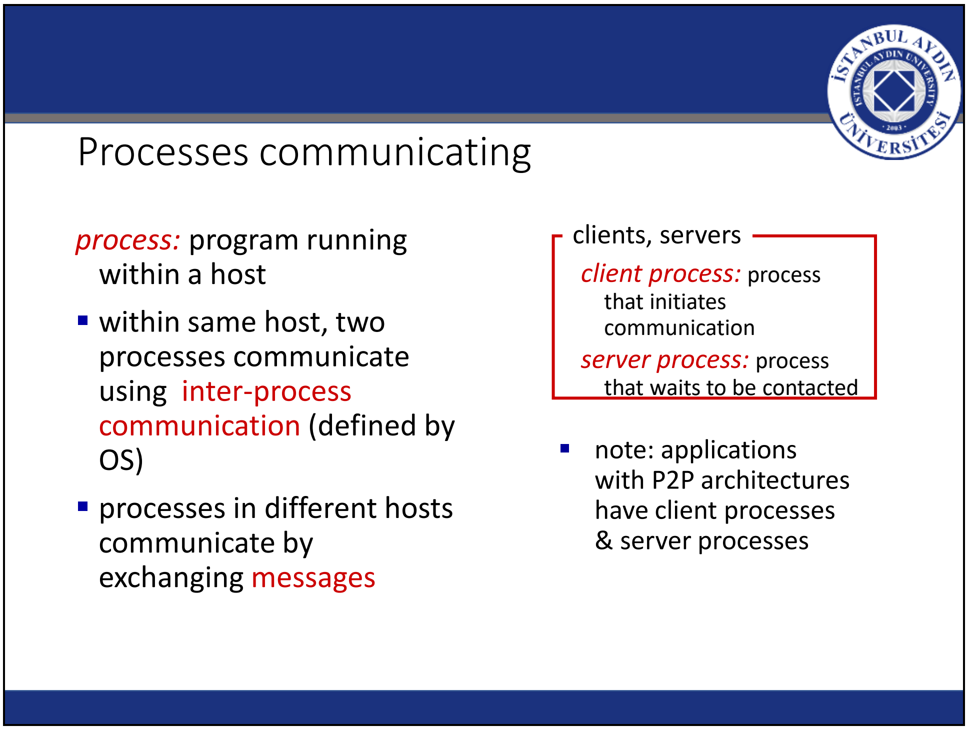
This slide introduces the client-server paradigm. In this model, there is a server that is usually always on, has a permanent IP address, and is often located in a data center for scalability. Clients contact the server to request services or data. Clients may connect only when needed, and they often use dynamic IP addresses, especially in home or mobile networks. Another key point is that clients usually do not communicate directly with each other in this architecture. Instead, they communicate through the server. Common examples are HTTP for the Web, IMAP for e-mail access, and FTP for file transfer. So, the client-server model is centralized and is very common on the Internet because it is simple to manage and easy for users to access.

Peer-peer architecture

- *no* always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, as well as new service demands
- peers are intermittently connected and change IP addresses
  - complex management
- example: P2P file sharing



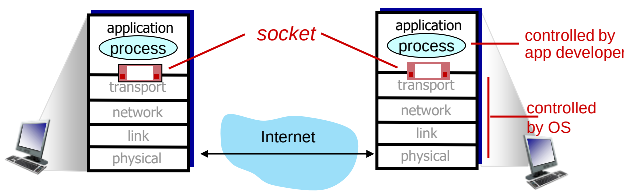
This slide presents the peer-to-peer, or P2P, architecture. Unlike the client-server model, P2P does not depend on one always-on central server. Instead, end systems called peers communicate directly with each other. Each peer can request service from other peers and can also provide service in return. This creates what the book calls self-scalability, because every new peer brings not only more demand but also more service capacity. However, there are also challenges. Peers may connect and disconnect frequently, and their IP addresses may change. Because of this, management, security, performance, and reliability become more complex. A classic example of this architecture is P2P file sharing. So, P2P can be efficient and cost-effective, but it is harder to control than the client-server model.



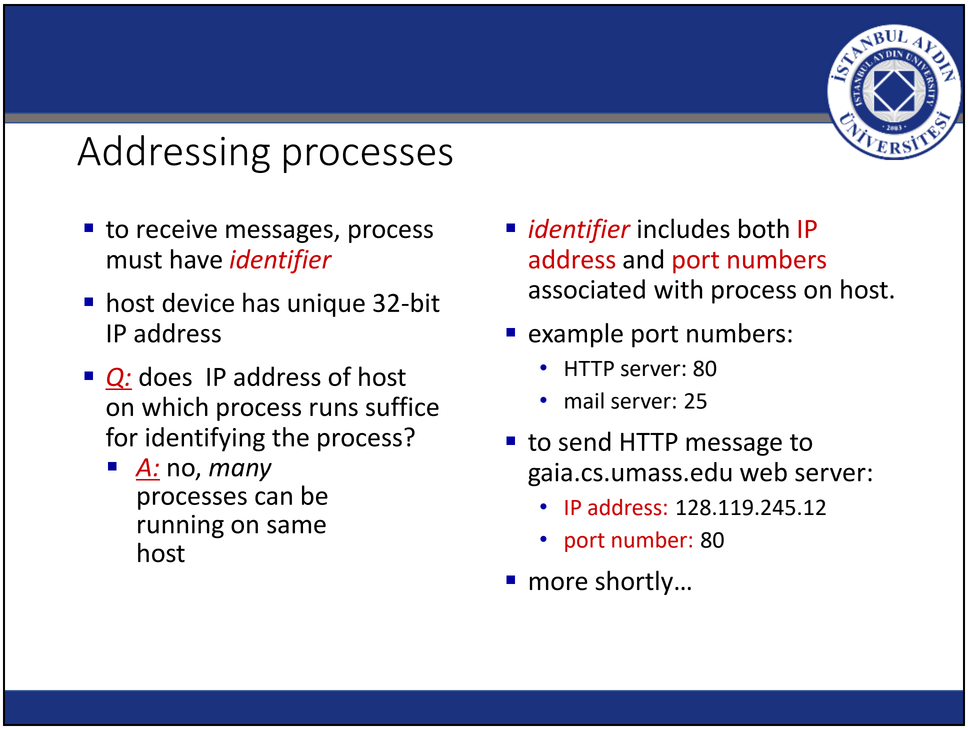
In this slide, we focus on processes communicating with each other. A process is simply a program that is running within a host. If two processes are on the same host, they can communicate through inter-process communication provided by the operating system. But in networking, we are mainly interested in processes running on different hosts. These processes communicate by exchanging messages over the network. For every communication session, we usually describe one side as the client process and the other side as the server process. The client process is the one that initiates communication, and the server process is the one that waits to be contacted. Even in peer-to-peer systems, a process can act as a client in one session and as a server in another session.

Sockets

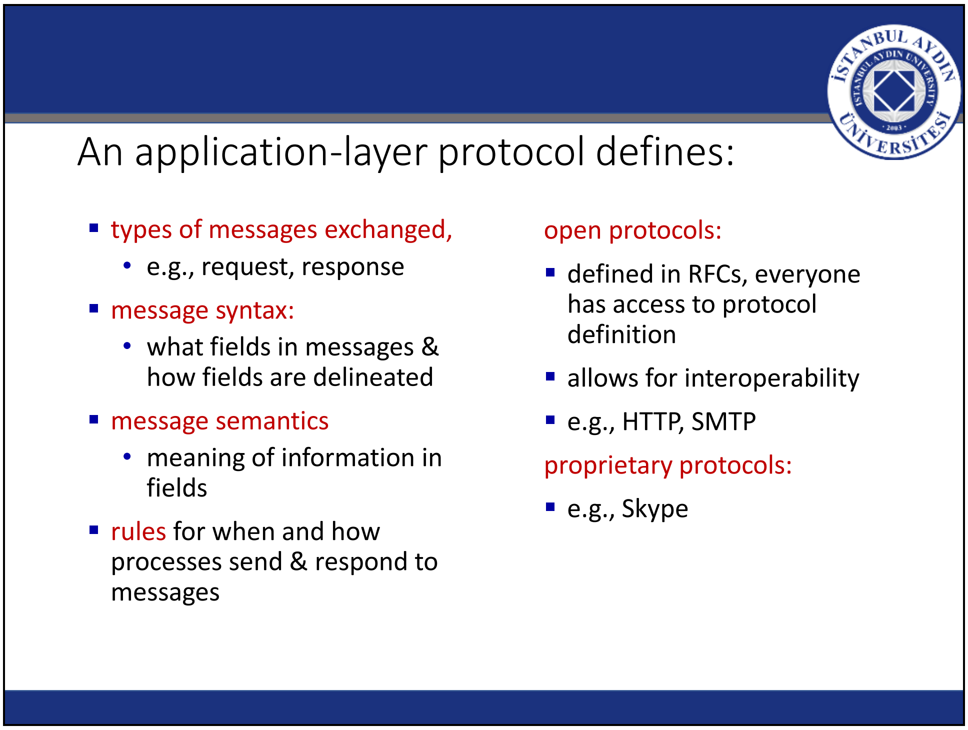
- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process
  - two sockets involved: one on each side



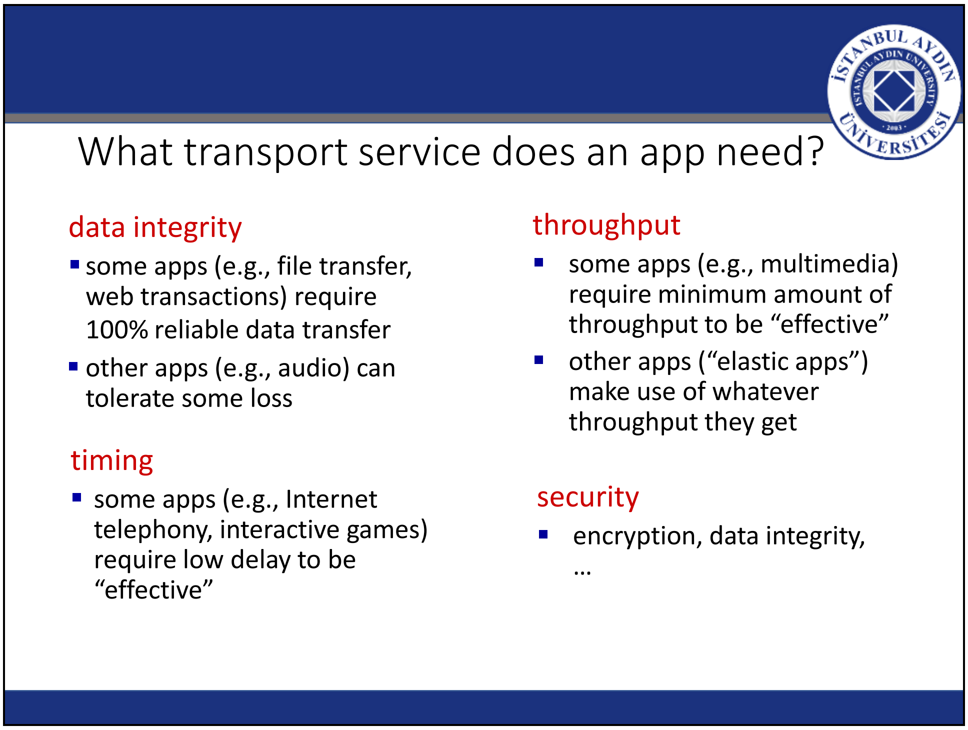
This slide explains the idea of a socket. A socket is the software interface between an application and the transport layer. The book uses a very helpful analogy: a socket is like a door. A process sends messages out through its socket, and the message enters the transport infrastructure on the other side. At the receiving host, the message arrives at another socket and is delivered to the receiving process. So, two sockets are involved in communication, one at each side. The application developer controls the application side of the socket, but the operating system controls most of the transport-layer details. In practice, sockets are very important because they are the programming interface used to build network applications.



This slide is about addressing processes. To receive messages, a process must have an identifier. At first, we may think that the IP address of the host is enough. But the answer is no, because many different processes can run on the same host at the same time. For this reason, a process is identified by both the host IP address and a port number. The IP address identifies the host, and the port number identifies the specific process, or more precisely, the destination socket. For example, a web server commonly uses port 80, and a mail server using SMTP commonly uses port 25. So, when a message is sent to a service, the network needs both pieces of information to deliver the data to the correct application.



This slide defines what an application-layer protocol is. An application-layer protocol specifies the types of messages that are exchanged, such as request and response messages. It also defines the syntax of these messages, meaning which fields they contain and how those fields are separated. In addition, it defines the semantics of the fields, or in other words, what the information in those fields means. Finally, it gives rules for when and how processes send messages and how they respond. The slide also distinguishes between open protocols and proprietary protocols. Open protocols are defined in public RFC documents, so everyone can use them and build interoperable systems. HTTP and SMTP are examples. Proprietary protocols, such as Skype in the book's example, are controlled by a company and are not fully open.



This slide asks an important design question: what transport service does an application need? Different applications have different requirements. The first requirement is data integrity. Some applications, such as file transfer or web transactions, need reliable delivery with no loss. Others, especially audio or video applications, can tolerate a small amount of loss. The second requirement is timing. Interactive applications such as Internet telephony or online games need low delay to work effectively. The third requirement is throughput. Multimedia applications may need a minimum throughput, while elastic applications simply use whatever bandwidth is available. The fourth requirement is security, including encryption and data integrity. So, the transport protocol should be chosen according to the needs of the application.

Transport service requirements: common apps

application	data loss	throughput	time sensitive?
file transfer/download	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5Kbps-1Mbps video:10Kbps-5Mbps	yes, 10's msec
streaming audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	Kbps+	yes, 10's msec
text messaging	no loss	elastic	yes and no

This final slide summarizes transport service requirements for common applications. File transfer, e-mail, and web documents require no data loss, but they are elastic, which means they can use whatever throughput is available and are not usually time sensitive. Real-time audio and video are different. They can tolerate some loss, but they need low delay, often in the range of tens of milliseconds, to remain effective. Streaming stored audio and video can also tolerate some loss, but they are less sensitive to delay because buffering can help. Interactive games are both delay sensitive and somewhat loss tolerant. Text messaging usually requires no loss, but its time sensitivity can vary depending on the application. This comparison helps us understand why different Internet applications may choose different transport protocols and design strategies.

Internet transport protocols services

TCP service:

- *reliable transport* between sending and receiving process
- *flow control:* sender won't overwhelm receiver
- *congestion control:* throttle sender when network overloaded
- *does not provide:* timing, minimum throughput guarantee, security
- *connection-oriented:* setup required between client and server processes

UDP service:

- *unreliable data transfer* between sending and receiving process
- *does not provide:* reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup.

Q: why bother? *Why* is there a UDP?

Internet transport protocols services

application	application layer protocol	transport protocol
file transfer/download	FTP [RFC 959]	TCP
e-mail	SMTP [RFC 5321]	TCP
Web documents	HTTP 1.1 [RFC 7320]	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary	TCP or UDP
streaming audio/video	HTTP [RFC 7320], DASH	TCP
interactive games	WOW, FPS (proprietary)	UDP or TCP

Securing TCP

Vanilla TCP & UDP sockets:
- no encryption
- cleartext passwords sent into socket traverse Internet in cleartext (!)

Transport Layer Security (TLS)
- provides encrypted TCP connections
- data integrity
- end-point authentication

TSL implemented in application layer
- apps use TSL libraries, that use TCP in turn

TLS socket API
- cleartext sent into socket traverse Internet *encrypted*
- see Chapter 8