Unit 22: Above the transport Layer...
The Internet *Hourglass*

There is just one network-layer protocol, **IP**. The narrownarrow waist facilitates interoperability.

![Diagram of the Hourglass Model]

**Applications**

**Transport**

**Data Link**

**Physical**

- **IP**
  - **TCP**
  - **UDP**
  - **SMTP**
  - **HTTP**
  - **DNS**
  - **NTP**

- **Ethernet**
  - **SONET**
  - **802.11**

- **Copper**
  - **Fiber**
  - **Radio**
End-to-End Layering View
Process A sends a packet to process B

**IP Address:**
A four-part number used by *Network Layer* to route a packet from one computer to another
Process Address

• to receive messages, process must have *identifier*

*identifier* includes both IP address and port numbers associated with process on host.

Example port numbers:
  - HTTP server: 80
  - Mail server: 25

To send HTTP message to gaia.cs.umass.edu web server:
  - IP address: 128.119.245.12
  - Port number: 80
Creating a network app

write programs that:
Årun on (different) end systems
Åcommunicate over network

no need to write software for network-core devices
Ånetwork-core devices do not run user applications
Åapplications on end systems allows for rapid app development, propagation
Streams of Bits/bytes can be transmitted: so what?

How do we know what is the INFORMATION inside?
Simple example

Representation of base types

- floating point: IEEE 754 versus non-standard
- integer: big-endian versus little-endian (e.g., 34,677,374)

<table>
<thead>
<tr>
<th>Big-endian</th>
<th>Little-endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2)</td>
<td>(126)</td>
</tr>
<tr>
<td>00000010</td>
<td>01111111</td>
</tr>
<tr>
<td>(17)</td>
<td>(34)</td>
</tr>
<tr>
<td>00010001</td>
<td>00100010</td>
</tr>
<tr>
<td>(34)</td>
<td>(17)</td>
</tr>
<tr>
<td>00100010</td>
<td>00010001</td>
</tr>
<tr>
<td>(126)</td>
<td>(2)</td>
</tr>
<tr>
<td>01111110</td>
<td>00000010</td>
</tr>
</tbody>
</table>

- on a 680x0 CPU, the 32 bit integer number 255 is stored as:
  00000000 00000000 00000000 11111111
- but an Intel 80x86-CPU stores this as:
  11111111 00000000 00000000 00000000
Taxonomy

Data types
- base types (e.g., ints, floats); must convert
- flat types (e.g., structures, arrays); must pack
- complex types (e.g., pointers);

Conversion Strategy
- canonical intermediate form
  - receiver-makes-right (an $N \times N$ solution)
Data Conversion

Two different types of rules are needed:

- Abstract syntax: a station must define what datatypes are to be transmitted
- Transfer syntax: it must be defined how these datatypes are transmitted, i.e. which representation has to be used.

Tagged versus untagged data

```
type = INT
len = 4
value = 417892
```
Abstract Syntax Notation.1 - ASN.1

Each transmitted data value belongs to an associated data type.

For the lower layers of the OSI-RM, only a fixed set of data types is needed (frame formats), for applications with their complex data types ASN.1 provides rules for the definition and usage of data types.

ASN.1 distinguishes between a data type (as the set of all possible values of this type) and values of this type (e.g. 1 is a value of data type Integer).

Basic ideas of ASN.1:

- Every data type has a globally unique name (type identifier)
- Every data type is stored in a library with its name and a description of its structure (written in ASN.1)
- A value is transmitted with its type identifier and some additional information (e.g. length of a string).
Definition of Datatypes using ASN.1 (1)

A data type definition is called “abstract syntax” it uses a Pascal-like syntax.

Lexical rules:
- Lowercase letters and uppercase letters are different
- A type identifier must start with an uppercase letter
- Keywords are written in uppercase letters

ASN.1 offers some predefined simple types:
- BOOLEAN (Values: True, False)
- INTEGER (natural numbers without upper bound)
- ENUMERATED (association between identifier and Integer value)
- REAL (floating point values without upper or lower bound)
- BIT STRING (unbounded sequence of bits)
- OCTET STRING (unbounded sequence of bytes/ octets)
- NULL (special value denoting absence of a value)
- OBJECT IDENTIFIER (denoting type names or other ASN.1-objects)
Definition of Datatypes using ASN.1 (2)

Examples:
- MonthsPerYear ::= INTEGER
  MonthsPerYear ::= INTEGER (1..12)
  Answer ::= ENUMERATED (correct(0), wrong(1), noAnswer(3))

With the following type constructors new types can be built from existing ones:
- SET: the order of transmission of the elements of a set is not specified. The number of elements is unbounded, their types can differ
- SET OF: like SET, but all elements are of the same type.
- SEQUENCE: the elements of a sequence are transmitted in the defined order. They can be of different types. The number of elements is unbounded.
- SEQUENCE OF: like SEQUENCE, but all elements are of the same type
- CHOICE: the type of a given value is chosen from a list of types (like a Pascal variant record)
- ANY: unspecified type
ASN.1 Transfer Syntax (1)

Some coding rules (the "transfer syntax") specifies how a value of a given type is transmitted. A value to be transmitted is coded in four parts:

- identification (type field or tags)
- length of data field in bytes
- data field
- termination flag, if length is unknown.

The coding of data depends on their type:

- integer numbers are transmitted in High-Endian Two's complement representation, using the minimal number of bytes: numbers smaller 128 are encoded in one byte, numbers smaller than 32767 are encoded in two bytes, ...
- Booleans: 0 is false, every value not equal 0 is true.
- for a sequence type first a type identification of the sequence itself is transmitted, followed by each member of the sequence.
- Similar rules apply to the transfer of set types
The Client-Server approach

Is the messagepassing/stream interface enough? What about more complex cooperation patterns?...

The client / server approach:

- A client wants to perform a specific action, e.g. to print a file. The client itself is not able or willing to do that (he has no printer), but he knows someone (another computer), who could do it (the printer is connected to that other computer). The other one is the server.

- The client transmits a request message to the server (including the file to be printed), asking the server to perform the service.

- The server receives this message and performs (probably) the appropriate action (i.e. prints the file).

- The results are sent back to the client via a reply message.
Service

Service: Any act or performance that one party can offer to another that is essentially intangible and does not result in the ownership of anything. Its production may or may not be tied to a physical product.

* D. Jobber, Principles and Practice of Marketing

Focus is on the *output*, the *result* of the service

*NOT* the means to achieve it
Client-Server Model

server:
- always-on host
- permanent IP address
- server farms for scaling

clients:
- communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do not communicate directly with each other
Berkeley Sockets for Client-Server

Server

- socket
- bind
- listen
- accept
- read
- write
- close

Synchronization point

Communication

Client

- socket
- connect
- write
- read
- close
An Alternative P2P 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
A useful tool: Remote Procedure Calls (RPC)

Remote Procedural Calls are the preferred tool to implement the client-server model.

In classical procedure calls the code of the procedure is located on the same computer (in the same address space) as the calling program, in an RPC the code is located on another computer.

One major design goal of an RPC system is transparency: ideally the caller should not know if the callee is located locally or remotely. So in RPC we have to consider the following topics:

- Parameter handling and marshalling
- Semantics
- Addressing

An RPC system is attractive for the users because automatic support for the conversion from local to remote procedural call can be supported (see below).
Local vs. Remote Procedural Call *(SLIDES 21-28 are a supplement!)*
Marshaling

Marshalling: taking parameters/results of a procedure call and prepare them for transmission over a network

- To ensure, e.g., transparency between different hardware, operating systems, programming languages
- Handled by client stub & server stub/skeleton

---

**Diagram:**

1. Client call to procedure
2. Stub builds message
3. Message is sent across the network
4. Server OS hands message to server stub
5. Stub unpacks message
6. Stub makes local call to "add"
RPC Parameter Passing

Procedures in common programming languages have different types of parameters and calling conventions, which have to be treated in a RPC:

- Simple call-by-value parameters are passed as is (e.g. simple integer values).
- Call-by-reference parameters are pointers; since different address spaces are used by sender and receiver, the denoted value (e.g. a buffer) has to be completely transmitted (so its length and its type must be known in advance). If the server changes some buffer values, the buffer must be retransmitted.
- Complex data types using pointers (e.g. graphs, trees or lists) cannot (or only with difficulty) be transmitted.

The stub procedures must use a common encoding convention for different parameter types.
Finding an RPC server (Addressing)

A client can use fixed, hard coded addresses for finding the appropriate server station. This approach is simple but not flexible.

A dynamic binding approach can be used:

- A server stub transmits at its initialization a message containing its name (procedure name), its version number, its address and a unique (within the server station) identification to a special station, the bindery station, which maintains a database of all available services.

- A client stub, if operating the first time, queries the bindery station for an appropriate server providing the requested service (i.e. service name, version number). If no server exists, the client stub fails. Otherwise, the bindery returns the address and the unique identification to the client stub.
RPC Semantics (1)

In normal operation the RPC should behave exactly as the corresponding local procedure call (LPC). In the local case it is assumed that a procedure call returns correctly (unless the system fails). This assumption is not valid for RPC systems. Several problems can arise:

- Addressing (the client could not find an appropriate service)
- The client or the server can fail
- Message loss

If the client could not find an appropriate server, a kind of exception handling is needed, thus violating the transparency requirement.

The server can fail before executing the requested action or while executing it or immediately before returning an answer; another possibility is the loss of either an answer of a request message. The client is not able to distinguish these cases.
RPC Semantics (2)

The client stub has three possibilities for further behavior if the result is missing:

- He can retransmit his request until he receives a correct message (the server can be restarted or another server was found). In case that the server crashed after execution of the command, it could be executed twice. This is called "at least once semantics".
- He can stop after transmitting one message and report an error to the client. This is called "at most once semantics".
- He can do anything else (e.g. make exactly 37 attempts), thus failing to give any guarantees to the client.

An action is called idempotent if multiple executions do not change the result or the state of the system (e.g. reading from a file does not change its state - a second read operation yields the same result - but this is not true for writing to a file). There are also semi-idempotent actions.

If all RPC actions are idempotent, the RPC semantics does not matter, since every request can be repeated without harm.

If there are non-idempotent actions, "exactly once" semantics is required.
Idempotent operations

An operation is idempotent if

1. Doing it twice has the same effect as doing it once
2. Doing it partially (several times, possibly) and then doing it whole has the same effect as doing it once

Example: writing a block to disk

Doing it partially, results in a bad checksum for the block (so the block becomes unreadable)

Doing it whole makes the block readable

Doing it again doesn't matter (it's the same block again)
Making an operation idempotent

Let's say you want to transfer $1000 to my (remote) bank account. Commands sent to the bank account are in blue.

Here's the non-idempotent way:

Add $1000 to my balance

This is non-idempotent, because doing it twice (which I can only encourage) will give me $2000.

The idempotent way:

Keep trying to read my balance (idempotent) until that succeeds, call it x
Add $1000 to it (a local operation)
Keep trying to write x+$1000 to my balance (idempotent) until that succeeds

Now we have a new problem to deal with: concurrency
Internet End-to-End View

- Process A sends a packet to process B

**IP Address:**
- A four-part number used by *Network Layer* to route a packet from one computer to another
Do I have to memorize the IP Address?

Host names depict machines in the organizations

Eg. robotics.eecs.berkeley.edu

This conveys more information to humans than 128.32.48.234

Why IP address?

- The network needs an address to route

Host names yield information to people and IP addresses yield information to routers
DNS: Domain Name System

people: many identifiers:
   - SSN, name, passport #

Internet hosts, routers:
   - IP address (32 bit) - used for addressing datagrams
   - "name" e.g., www.yahoo.com - used by humans

Q: map between IP address and name, and vice versa?

Domain Name System:
   - *distributed database* implemented in hierarchy of many *name servers*
   - *application-layer protocol* host, routers, name servers to communicate to *resolve* names (address/name translation)
      - note: core Internet function, implemented as application-layer protocol
      - complexity at network's *edge*
DNS Features

- Hierarchical Namespace
  - Name servers assigned zones of the hierarchical namespace
  - Backup servers available for redundancy

- Distributed architecture for storing names

- Administration divided along the same hierarchy

- Client server interaction on UDP Port 53
Host names are organized hierarchically

- The first level names are called "Top Level Domains".
- Depth of tree is arbitrary (limit 128).
- Domains are subtrees.
  - E.g. berkeley.edu and eecs.berkeley.edu.
DNS: a distributed, hierarchical database

Client wants IP for www.amazon.com; 1st approx:
- Client queries root server to find com DNS server
- Client queries .com DNS server to get amazon.com DNS server
- Client queries amazon.com DNS server to get IP address for www.amazon.com
DNS: Root name servers

- contacted by local name server that can not resolve name
- root name server:
  - contacts authoritative name server if name mapping not known
  - gets mapping
  - returns mapping to local name server

13 root name servers worldwide:

- a Verisign, Dulles, VA
- b USC-ISI Marina del Rey, CA
- c Cogent, Herndon, VA (also LA)
- d U Maryland College Park, MD
- e NASA Mt View, CA
- f Internet Software C. Palo Alto, CA (and 36 other locations)
- g US DoD Vienna, VA
- h ARL Aberdeen, MD
- i Autonomica, Stockholm (plus 28 other locations)
- j Verisign, (21 locations)
- k RIPE London (also 16 other locations)
- m WIDE Tokyo (also Seoul, Paris, SF)
- i ICANN Los Angeles, CA
TLD and Authoritative Servers

Top-level domain (TLD) servers:
- responsible for com, org, net, edu, aero, jobs, museums, and all top-level country domains, e.g.: uk, fr, ca, jp
- Network Solutions maintains servers for com TLD
- Educause for edu TLD

Authoritative DNS servers:
- organization’s DNS servers, providing authoritative hostname to IP mappings for organization’s servers (e.g., Web, mail).
- can be maintained by organization or service provider

Local Name Server
- each ISP (residential ISP, company, university) has one
- when host makes DNS query, query is sent to its local DNS server
DNS name resolution

- host at cis.poly.edu wants IP address for gaia.cs.umass.edu

**iterated query:**
- contacted server replies with name of server to contact
- “I don’t know this name, but ask this server”
Alternatively: Recursive Query Resolution

**recursive query:**
- puts burden of name resolution on contacted name server
- heavy load?

![Diagram of recursive query resolution]

1. Requesting host `cis.poly.edu`
2. Local DNS server `dns.poly.edu`
3. Root DNS server
4. TLD DNS server `gaia.cs.umass.edu`
5. Authoritative DNS server `dns.cs.umass.edu`
6. Recursive query
7. Heavy load?
DNS Caching

- Performing all these queries takes time
  - And all this before actual communication takes place
  - E.g., 1-second latency before starting Web download

- Caching can greatly reduce overhead
  - The top-level servers very rarely change
  - Popular sites (e.g., www.cnn.com) visited often
  - Local DNS server often has the information cached

- How DNS caching works
  - DNS servers cache responses to queries
  - Responses include a time to live (TTL) field
  - Server deletes cached entry after TTL expires
Separating Naming and Addressing

Names are easier to remember
- www.cnn.com vs. 64.236.16.20 (but not tiny urls)

Addresses can change underneath
- Move www.cnn.com to 4.125.91.21
- E.g., renumbering when changing providers

Name could map to multiple IP addresses
- www.cnn.com to multiple (8) replicas of the Web site
- Enables
  - Load-balancing
  - Reducing latency by picking nearby servers
  - Tailoring content based on requester’s location/identity

Multiple names for the same address
- E.g., aliases like www.cnn.com and cnn.com
DNS records

**DNS**: distributed db storing resource records (RR)

RR format: \((\text{name}, \text{value}, \text{type}, \text{ttl})\)

- **type=A**
  - \text{name} is hostname
  - \text{value} is IP address

- **type=NS**
  - \text{name} is domain (e.g., foo.com)
  - \text{value} is hostname of authoritative name server for this domain

- **type=CNAME**
  - \text{name} is alias name for some “canonical” (the real) name
  - \text{www.ibm.com} is really servereast.backup2.ibm.com
  - \text{value} is canonical name

- **type=MX**
  - \text{value} is name of mailserver associated with \text{name}
The WWW

*Content:* A distributed database of URLs

*Client-Server Principle:*
- Servers which store files and execute remote commands
- Clients retrieve and display "pages" of content linked by hypertext

*The basic aspects:*
- Need a language to define the objects and the layout: HTML, XML
- Need the way to identify the resource: URL
- Need a protocol to transfer information between clients and servers: HTTP

*Note: the idea is old!!!
Architectural Overview
The parts of the Web model.

[Tanenbaum]
HTML

A Web page has several components
- Base HTML file
- Referenced objects (e.g., images)

HyperText Markup Language (HTML)
- Representation of hypertext documents in ASCII format
- Web browsers interpret HTML when rendering a page
- Several functions:
  - Format text, reference images, embed hyperlinks (HREF)

Straight-forward to learn
- Syntax easy to understand
- Authoring programs can auto-generate HTML
- Source almost always available
Uniform Record Locator

- protocol://host-name:port/directory-path/resource

- Extend the idea of hierarchical namespaces to include anything in a file system

- Extend to program executions as well
  - Server side processing can be incorporated in the name
HTTP Overview [this and following EECS122, Abhay Parekh]

HTTP: hypertext transfer protocol
- Web’s application layer protocol
- client/server model
  - client: browser that requests, receives, “displays” Web objects
  - server: Web server sends objects in response to requests
- HTTP 1.0: RFC 1945
- HTTP 1.1: RFC 2068
HTTP overview (cont)

Uses DNS to obtain the IP address

Uses TCP:
- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- TCP connection closed

FTP

HTTP is “stateless”
- server maintains no information about past client requests

Protocols that maintain “state” are complex!
- past history (state) must be maintained
- if server/client crashes, their views of “state” may be inconsistent, must be reconciled
HTTP request message

GET — transfer resource from given URL
HEAD — GET resource metadata (headers) only
PUT — store/modify resource under the given URL
DELETE — remove resource
POST — provide input for a process identified by the given URL (usually used to post CGI parameters)

HTTP request message:
- ASCII (human-readable format)

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
User-agent: Mozilla/4.0
Connection: close
Accept-language: fr

(extra carriage return, line feed)
```
How does it work - Example

1. After finding out the IP address of the hosté (DNS)
2. Client sends the get request via socket established in 1
3. Server sends the html file, which is encapsulated in its response
4. http server tells tcp to terminate connection
5. http client receives the file and the browser parses ité contains ten jpeg images
6. Client repeats steps 1-4
Persistency of TCP usage

- A web page typically contains many objects
  - E.g. Images
  - Each object must be requested with a separate http “Get” command
  - Non Persistent Connection:
    - Different TCP connection for each object request.
    - HTTP 1.0
  - Persistent Connection
    - Reuse the same TCP connection for each object request
    - HTTP 1.1
HTTP 1.0 Performance

- Create a new TCP connection for each resource
  - Large number of embedded objects in a web page
  - Many short lived connections
- Requires 2 RTTs per object
- TCP transfer
  - Too slow for small object
  - May never exit slow-start phase
- Connections may be set up in parallel (5 is default in most browsers)
- OS overhead for each TCP connection
Internal organization of HTTP

**Persistent HTTP**
- server leaves connection open after sending response
  - TCP overhead minimized
- subsequent HTTP messages between same client/server sent over open connection

**No pipelining:**
- client issues new request only when previous response has been received
- one RTT for each referenced object

**Pipelining:**
- client sends requests as soon as it encounters a referenced object
- as little as one RTT for all the referenced objects
- default in HTTP/1.1
The advantage of pipelining

Comment: remember we have already discussed pipelining in another context different level...
Caching

- Store frequently referenced objects closer to the clients
  - Saves Time: No need to go all the way to the server (access could look “instantaneous”)
  - Saves Access Bandwidth
  - Saves Web Server Resources

- Limitations?
  - Frequently changing objects
  - Hit counts
  - Privacy

Where? Possibly everywhere: at the servers, at your network, at the client!!!
Performance and Reliability...

- Problem: You are a web content provider
  - How do you handle millions of web clients?
  - How do you ensure that all clients experience good performance?
  - How do you maintain availability in the presence of server and network failures?

- Solutions:
  - Add more servers at different locations → If you are CNN this might work!
  - Caching
  - Content Distribution Networks (Replication)
Server side caching

- Cache documents close to server → decrease server load
- Typically done by content providers
... and forward caching

- Cache documents close to clients → reduce network traffic and decrease latency
- Typically done by ISPs or corporate LANs
HTTP is Stateless

A Stateless protocol (e.g. GET, PUT, DELETE)

- Each request-response exchange treated independently
- Servers *not* required to retain state

This is **good** - Improves scalability on the server-side

- Don’t have to retain info across requests
- Can handle higher rate of requests
- Order of requests doesn’t matter

This is also **bad** - Some applications need persistent state

- Need to uniquely identify user or store temporary info
- *e.g.*, Shopping cart, user preferences/profiles, usage tracking, etc.
State in a Stateless Protocol: Cookies

Client-side state maintenance
- Client stores small state on behalf of server
- Client sends state in future requests to the server

Can provide authentication

Request

Response
Set-Cookie: XYZ

Request
Cookie: XYZ
Notion of Fate-Sharing

Àldea: when storing state in a distributed system, keep it co-located with the entities that ultimately rely on the state

À Fate-sharing is a technique for dealing with failure
  À Only way that failure can cause loss of the critical state is if the entity that cares about it also fails ...
  À in which case it doesn’t matter

À Often argues for keeping network state at end hosts rather than inside routers
  À In keeping with End-to-End principle
  À E.g., packet-switching rather than circuit-switching
  À E.g., HTTP “cookies”