Gabor Paller 2003.11.05

Network Computing

Course material

Gabor Paller

1. INTERNET-BASED PROTOCOLS

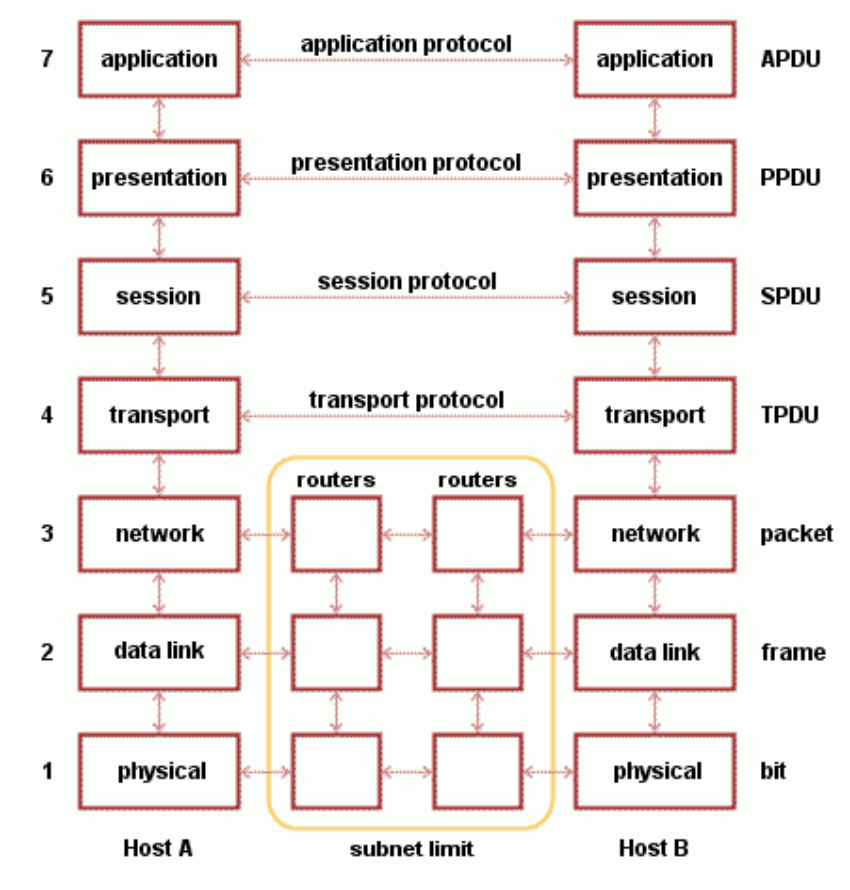
1.1 Short review of OSI layers

In the early day of network-connected computer architectures proprietary communication protocols flourished like SNA from IBM, DecNet from DEC. in order to avoid the interconnection problem of these incompatible architectures, the ISO (International Standards Organisation), body that depends on the UNO and composed of 140 national normalisation bodies, has developed a reference model called the OSI model (Open Systems Interconnection model). This model describes the fundamental concepts and the approach used to normalize the interconnection of open systems.

The OSI model is not real network architecture, because it does not really specify the services that the layers must provide and protocols each layer should use. It rather describes the main functionality of the layers. ISO has standard services and protocols for each layer but this is out of scope of the OSI model.

The first works related to the OSI model date from 1977. They were based on the experience gained in the area of wide area networks and local private networks; the OSI model was indeed supposed to be valid for any type of network. In 1978, the ISO proposed this model as the standard ISO IS7498. In 1984, 12 European manufacturers, joined in 1985 by the main American manufacturers, adopted this standard.

The OSI model is composed of 7 layers.



Principles of the OSI architecture are the following:

- A new layer is created for a new abstraction and every layer has a well-defined function
- There is a virtual communication between each layer meaning that e.g. session protocol in Host A is able to communicate with session protocol in Host B.
- A layer communicates only with the layers directly above and under the layer. This is called *stack* architecture.

Physical layer	This layer is in charge of the raw transmission of bits over a communication channel.
Data link layer	Splits the input data of the sender into frames, sends these frames in sequence and manages the acknowledgement frames sent back by the receiver.
Network layer	This layer is in charge of the sub-network, i.e. the routing packets over the sub-networks and the interconnection of the various sub-networks.
Transport layer	This layer is responsible for the good delivery of messages to the recipient.
Session layer	This layer sets up and synchronizes the exchanges between distant processes.
Presentation layer	This layer deals with the syntax and semantics of the transmitted data: it processes the data so as to make it compatible between

	communicating tasks.
Application layer	This layer is the point of contact between the user and the network.

The OSI model was not adopted widely and popular protocols are not based on this model. Still, the impact of the model is big and we will discover several elements of the model in the popular protocols. The OSI model problems are the following:

- Too complex, implementations are large
- Redundant, flow control appears in multiple layers
- Security requirements are not addressed (although can be introduced as separate layer(s))
- Standardization process is too bureaucratic

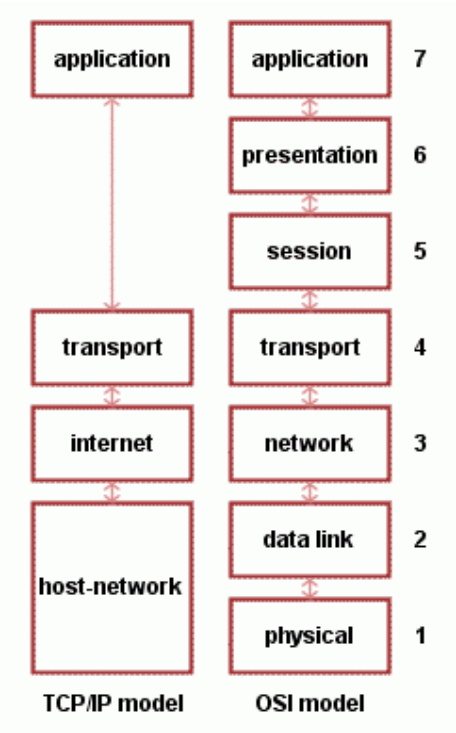
1.2 The Internet Protocol (IP)

There exists extremely wide variety of data link and physical protocols. Here are some motivation examples:

- Ethernet is a bus-like system that relies on collision detection. Ethernet specifies data link and physical layers.
- Frame Relay is a frame protocol used to connect Local Area Networks (LANs) over Wide Area Networks (WANs). Frame Relay specifies data link layer.
- Integrated Services Digital Network (ISDN) is a phone network technology that relies on digital transmission of data frames. ISDN specifies data link and physical layers
- Asynchronous Transfer Mode (ATM) is a frame protocol that supports services with different quality of service requirements (constant bit rate, variable bit rate, connection-oriented, connectionless) ATM specifies data link layer.

Integrating these protocols into one virtual network requires a single protocol in the network layer. It is not possible to put the common ground to higher layers because inter-network routing is generally a requirement. The Internet Protocol (IP) has become the de-facto worldwide standard to be this single protocol.

The IP protocol family uses a different network layer model than the OSI. The IP stack has only 4 layers.



IP sits on top of the host network layers (physical and data links) and provides uniform services to higher-level layers. The main tasks of IP are the following:

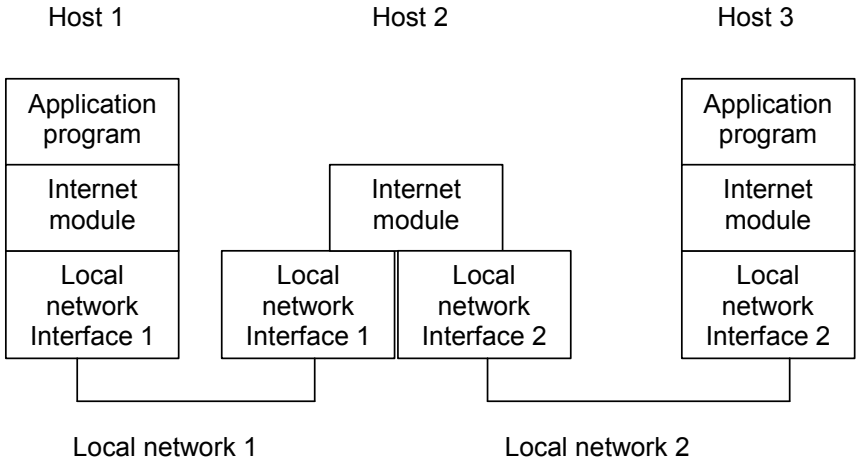
- Addressing. The internet module maps internet addresses to local net addresses. Addresses are fixed length of four octets in case of widely deployed IPv4. An address begins with a network number, followed by local address (called the "rest" field)
- Fragmentation. Fragmentation of an internet datagram is necessary when it originates in a local net that allows a large packet size and must traverse a local net that limits packets to a smaller size to reach its destination.

The following scenario illustrates the model of operation for transmitting a datagram from one application program to another:

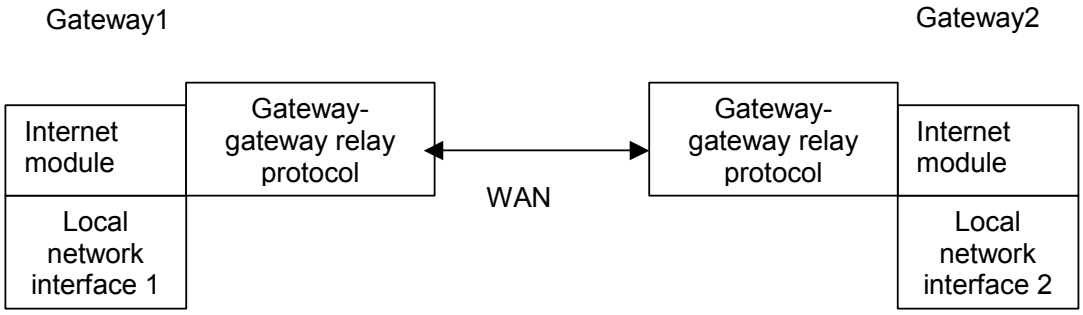
We suppose that this transmission will involve one intermediate gateway.

- The sending application program prepares its data and calls on its local internet module to send that data as a datagram and passes the destination address and other parameters as arguments of the call.
- The internet module prepares a datagram header and attaches the data to it. The internet module determines a local network address for this internet address, in this case it is the address of a gateway.
- It sends this datagram and the local network address to the local network interface.
- The local network interface creates a local network header, and attaches the datagram to it, then sends the result via the local network.

- The datagram arrives at a gateway host wrapped in the local network header, the local network interface strips off this header, and turns the datagram over to the internet module.
- The internet module determines from the internet address that the datagram is to be forwarded to another host in a second network. The internet module determines a local net address for the destination host. It calls on the local network interface for that network to send the datagram.
- This local network interface creates a local network header and attaches the datagram sending the result to the destination host.
- At this destination host the datagram is stripped of the local net header by the local network interface and handed to the internet module.
- The internet module determines that the datagram is for an application program in this host. It passes the data to the application program in response to a system call, passing the source address and other parameters as results of the call.



If the local networks are separated by WAN connections, the gateways speak a WAN protocol among each other to forward IP frames. This can be for example the Frame Relay protocol. Also, gateways talk to each other e.g. to update routing tables. Early, already obsolete example of such a routing table update protocol is Gateway-to-Gateway protocol, GGP.



The Internet Protocol family is managed by the Internet Engineering Task Force standardization body. (IETF, www.ietf.org). IETF issues Request For Comments (RFCs), papers that describe a particular technical solution. Some RFCs become widely deployed solutions; some are forgotten and become irrelevant. Take care when you pick an RFC that you always check its status. The Internet Protocol is defined by RFC791.

Each IP packet is composed of a header and the payload that follows the header. The IP packet itself is packetized and possibly fragmented by the host network layers.

The format of the IP header is the following:

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The meaning of the fields is the following.

Name	Length (bits)	Description
Version	4	The Version field indicates the format of the internet header. Its value is 4 for IPv4
IHL	4	Internet Header Length is the length of the internet header in 32 bit words, and thus points to the beginning of the data. Note that the minimum value for a correct header is 5.
Type of Service	8	The Type of Service provides an indication of the abstract parameters of the quality of service desired. These parameters are to be used to guide the selection of the actual service parameters when transmitting a datagram through a particular network. Several networks offer service precedence, which somehow treats high

		precedence traffic as more important than other traffic (generally by accepting only traffic above a certain precedence at time of high load). The major choice is a three way tradeoff between low-delay, high-reliability, and high-throughput. Bits 0-2: Precedence. 111 - Network Control 110 - Internetwork Control 101 - CRITIC/ECP 100 - Flash Override 011 - Flash 010 - Immediate 001 - Priority 000 - Routine Bit 3: 0 = Normal Delay, 1 = Low Delay. Bits 4: 0 = Normal Throughput, 1 = High Throughput. Bits 5: 0 = Normal Relibility, 1 = High Relibility. Bit 6-7: Reserved for Future Use. The use of the Delay, Throughput, and Reliability indications may increase the cost (in some sense) of the service. In many networks better performance for one of these parameters is coupled with worse performance on another. Except for very unusual cases at most two of these three indications should be set.
Total Length	16	Total Length is the length of the datagram, measured in octets, including internet header and data. This field allows the length of a datagram to be up to 65,535 octets. Such long datagrams are impractical for most hosts and networks. All hosts must be prepared to accept datagrams of up to 576 octets (whether they arrive whole or in fragments). It is recommended that hosts only send datagrams larger than 576 octets if they have assurance that the destination is prepared to accept the larger datagrams.
Identification	16	An identifying value assigned by the sender to aid in assembling the fragments of a datagram.
Flags	3	Bit 1: (DF) 0 = May Fragment, 1 = Don't Fragment. Bit 2: (MF) 0 = Last Fragment, 1 = More Fragments.
Fragment Offset	13	This field indicates where in the datagram this fragment belongs. The fragment offset is measured in units of 8 octets (64 bits). The first fragment has offset zero.
Time to Live (TTL)	8	This field indicates the maximum time the datagram is allowed to remain in the internet system. If this field contains the value zero, then the datagram must be destroyed. This field is modified in internet header processing. The time is measured in units of seconds, but since every module that processes a datagram must

		decrease the TTL by at least one even if it process the datagram in less than a second, the TTL must be thought of only as an upper bound on the time a datagram may exist. The intention is to cause undeliverable datagrams to be discarded, and to bound the maximum datagram lifetime.
Protocol	8	This field indicates the next level protocol used in the data portion of the internet datagram (like UDP or TCP).
Header checksum	16	The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header. For purposes of computing the checksum, the value of the checksum field is zero.
Source address	32	IP address of the sender
Destination address	32	IP address of the receiver

On top of the mandatory header field, the IP header may contain optional headers. The optional header fields can take up to 40 bytes. The identification of optional headers depend on the IHL mandatory header field, if IHL is larger than 5, optional headers are present. Examples of IP options include; timestamps, record route, loose source route, and strict source route. Source routing (loose or strict with record route option) means that the sender specifies the routing when it sends the IP packet and the routers are not allowed to choose a different route (IP packet must be dropped if the route is not feasible).

The general option field looks like the following:

Single-byte option:

```
0 1 2 3 4 5 6 7 8
+-+-+-+-+-+-+-+-+
|  Option code  |
+-+-+-+-+-+-+-+-+
```

Variable-length option:

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Option type   | Option length |          Option bytes …       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The Option bytes part contains as many bytes as the option length field indicates. The option headers always fit into 32 bit words, if there are less bytes, padding is used to fill up to the next 32-bit word boundary.

1.3 Controlling IP packet flow: ICMP

Occasionally a gateway or destination host will communicate with a source host, for example, to report an error in datagram processing. For such purposes this protocol, the Internet Control Message Protocol (ICMP), is used. ICMP, uses the basic support of IP as if

it were a higher level protocol, however, ICMP is actually an integral part of IP, andmust be implemented by every IP module.

ICMP messages are sent in several situations: for example, when a datagram cannot reach its destination, when the gateway does not have the buffering capacity to forward a datagram, and when the gateway can direct the host to send traffic on a shorter route.

The Internet Protocol is not designed to be absolutely reliable. The purpose of these control messages is to provide feedback about problems in the communication environment, not to make IP reliable. There are still no guarantees that a datagram will be delivered or a control message will be returned. Some datagrams may still be undelivered without any report of their loss. The higher level protocols that use IP must implement their own reliability procedures if reliable communication is required.

The ICMP messages typically report errors in the processing of datagrams. To avoid the infinite regress of messages about messages etc., no ICMP messages are sent about ICMP messages. Also ICMP messages are only sent about errors in handling fragment zero of fragemented datagrams. (Fragment zero has the fragment offeset equal to zero).

ICMP messages are sent using the basic IP header. The first octet of the data portion of the datagram is an ICMP type field; the value of this field determines the format of the remaining data. The Protocol field in the IP header is set to 1, this characterizes ICMP.

ICMP is specified by RCF792.

Basic ICMP messages are the following:

- Destination unreachable
- Time exceeded (TTL expired)
- Parameter problem
- Source quench (sent when a gateway has to discard an IP packet due to buffer overflow)
- Redirect (redirects the sender to another gateway which is closer to the destination but still reachable on the sender's local network)
- Timestamp (the sender sends a timestamp, the receiver returns it together with another timestamp. Used to measure network delay)
- Information request and reply (used to find out IP addresses on the local network)
- Echo request and reply (used to check the accessibility of another IP node. Used by the ping command)

1.4 IPv6

IP version 6 (IPv6) is a new version of the Internet Protocol, designed as the successor to IP version 4 (IPv4) [RFC-791]. The changes from IPv4 to IPv6 fall primarily into the following categories:

- Expanded Addressing Capabilities. IPv6 increases the IP address size from 32 bits to 128 bits, to support more levels of addressing hierarchy, a much greater number

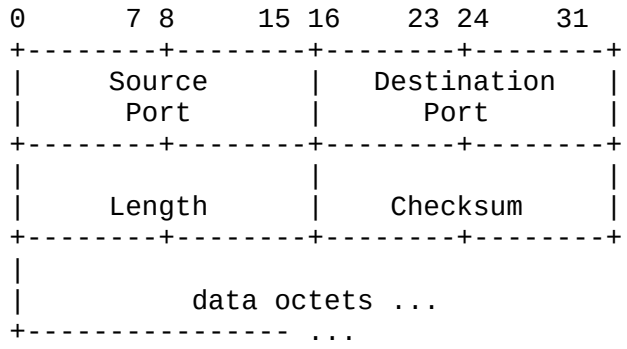
of addressable nodes, and simpler auto-configuration of addresses. The scalability of multicast routing is improved by adding a "scope" field to multicast addresses. And a new type of address called an "anycast address" is defined, used to send a packet to any one of a group of nodes.

- Header Format Simplification. Some IPv4 header fields have been dropped or made optional, to reduce the common-case processing cost of packet handling and to limit the bandwidth cost of the IPv6 header.
- Improved Support for Extensions and Options. Changes in the way IP header options are encoded allows for more efficient forwarding, less stringent limits on the length of options, and greater flexibility for introducing new options in the future.
- Flow Labeling Capability. A new capability is added to enable the labeling of packets belonging to particular traffic "flows" for which the sender requests special handling, such as non-default quality of service or "real-time" service.
- Authentication and Privacy Capabilities. Extensions to support authentication, data integrity, and (optional) data confidentiality are specified for IPv6.

IPv6 is specified by RFC2460.

1.5 The UDP protocol

IP provides the functionality of the network layer. IP services are normally not directly accessible for the protocol user, higher-level protocols cover it. The User Datagram Protocol (UDP) provides access to non-reliable, datagram services. UDP defines the notion of the *port* in addition to the IP addresses in the IP layer. Port identifies an application on the IP host. A certain application on a certain host is thus identified by the IP address (for the host) and the port (application). This allows multiple applications on one host to use IP services. The UDP packet is transferred on top of IP (sent in the data part of the IP packet) so the UDP packet is prefixed by the IP headers. The protocol number associated with UDP is 17 for IP header purposes. The UDP packet looks like the following:



Length is the total length of UDP data and headers (so it's minimum value is 8). Checksum is computed similarly to the checksum in the IP header.

UDP is specified by RFC768.

We will see later how to send and receive UDP packets from applications.

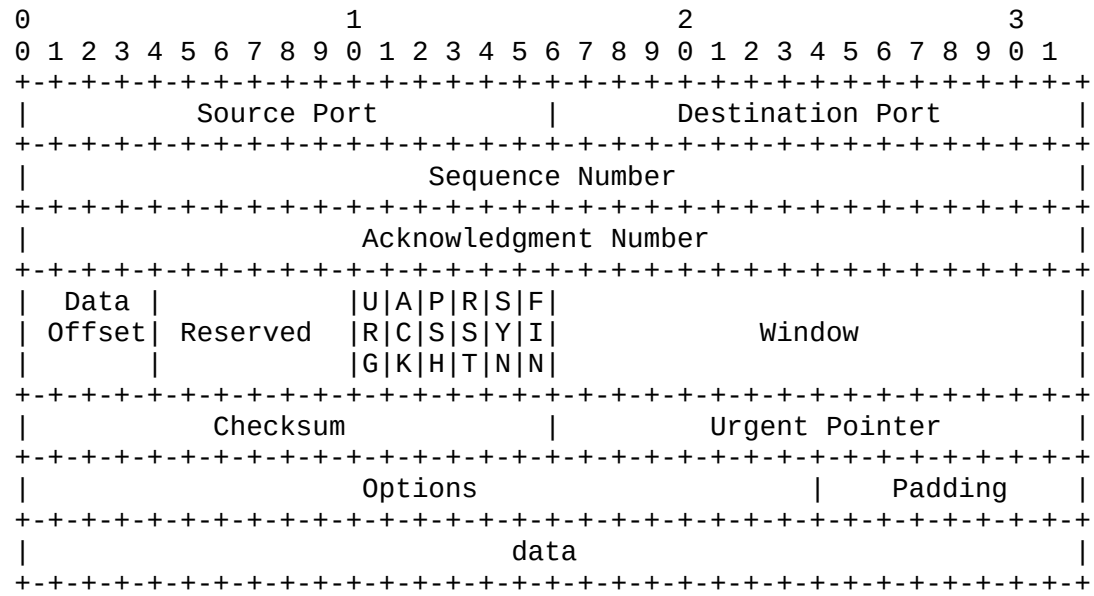
1.6 The TCP protocol

UDP (as IP) is provides unreliable transfer. This means that the delivery of the packets depends on the traffic situation transmission quality of the network. If the packet is lost when transmitted neither the sender, nor the receiver will know about it. Packets can be lost in many ways on IP networks; for example noise on the transmission line can distort some bits in the packet making it unusable or routers can drop packets when they receive more packets than they can transmit (because the incoming line's bandwidth is larger than the bandwidth of the outgoing line).

Other important source of faults that TCP handles is packet reordering or packet duplication. A packet sent later may arrive earlier to the receiver because some router on the way has chosen a different, faster route for the second packet. Lower level (physical or data link) protocols may generate duplicate packets due to race conditions.

The majority of the use cases require reliable transmission. Reliable transmission means that the sender notices if a packet is lost and it retransmits lost packets. If the error is fatal, certain number of retransmissions doesn't help, the user of the protocol is notified.

TCP is specified by RFC793. TCP segments are transmitted in the data part of IP packets. The TCP packet looks like the following:



The meaning of the fields is the following.

Name	Length (bits)	Description
Source Port	16	Source port number
Destination Port	16	Destination port number
Sequence Number	32	The sequence number of the first data octet in this segment (except when SYN is present). If SYN is present the sequence number is the initial sequence number (ISN) and the first data octet is ISN+1.
Acknowledgement Number	32	If the ACK control bit is set this field contains the value of the next sequence number the sender of the segment is

		expecting to receive. Once a connection is established this is always sent.
Data offset	4	The number of 32 bit words in the TCP Header. This indicates where the data begins. The TCP header (even one including options) is an integral number of 32 bits long.
Control Bits	6	URG: Urgent Pointer field significant ACK: Acknowledgment field significant PSH: Push Function RST: Reset the connection SYN: Synchronize sequence numbers FIN: No more data from sender
Window	16	The number of data octets beginning with the one indicated in the acknowledgment field that the sender of this segment is willing to accept.
Checksum	16	The checksum field is the 16 bit one's complement of the one's complement sum of all 16-bit words in the header and text.
Urgent pointer	16	This field communicates the current value of the urgent pointer as a positive offset from the sequence number in this segment. The urgent pointer points to the sequence number of the octet following the urgent data. This field is only be interpreted in segments with the URG control bit set.

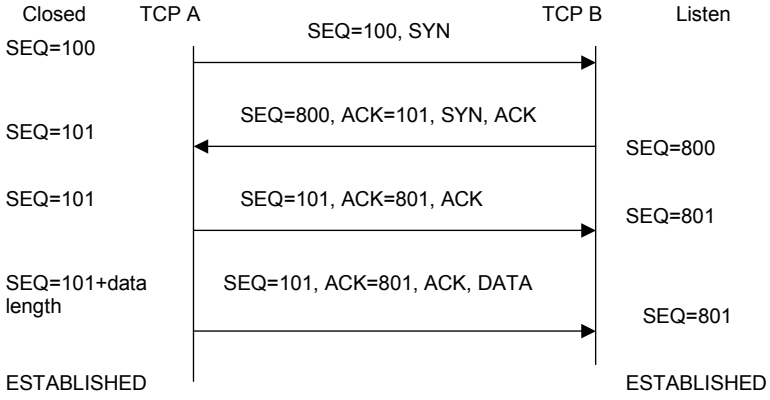
A fundamental notion in the design is that every octet of data sent over a TCP connection has a sequence number. Since every octet is sequenced, each of them can be acknowledged. The acknowledgment mechanism employed is cumulative so that an acknowledgment of sequence number X indicates that all octets up to but not including X have been received. It is essential to remember that the actual sequence number space is finite, though very large. This space ranges from 0 to 2**32 - 1. Since the space is finite, all arithmetic dealing with sequence numbers must be performed modulo 2**32.

The sequence number width (32 bits) and the transmission speed determine the efficiency of the dupplicate detection. TCP can be confused if the sequence number cycles through (we reach the same or overlapping sequence number after many modulo 2**32 increments). TCP uses the assumption that it takes a reasonably long time to cycle through the sequence number space and during that time duplicate packages disappear from the network.

The protocol is initialized with the so-called 3-way handshake.

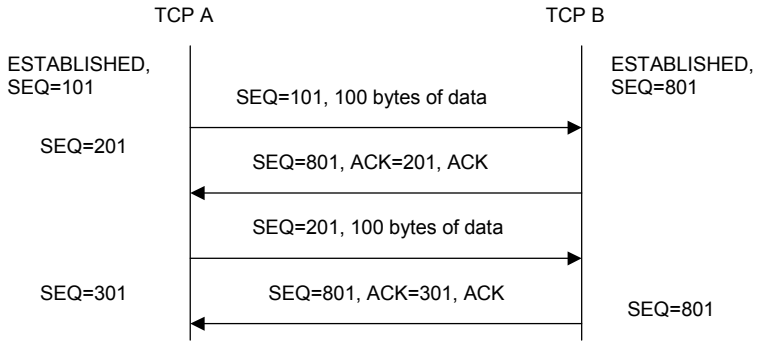
- A sends to B: SYN my sequence number is X
- B sends to A: ACK your sequence number is X
- B sends to A: SYN my sequence number is Y
- A sends to B: ACK your sequence number is Y

As the second and the third steps can be combined into one packet, the handshake requires 3 packets. Example:

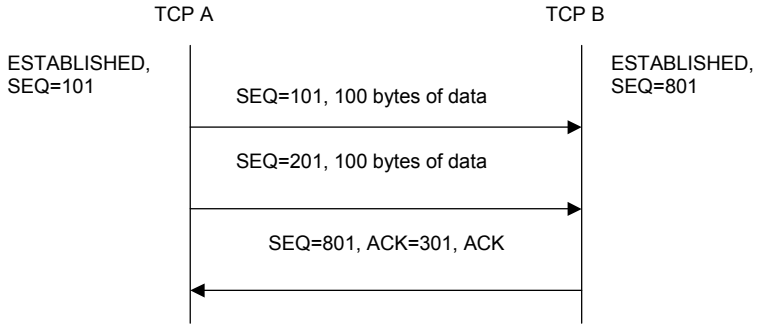


By convention, each control packet (SYN, FIN) counts 1 byte (although the packet itself doesn't contain data part) so that it is possible to acknowledge it. In any other case the SEQ is increased by the number of bytes in the data part.

The normal flow of operation is depicted in the following picture.



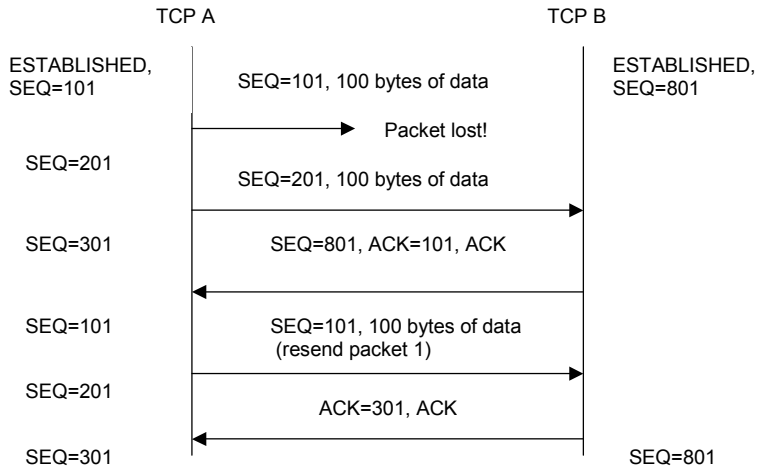
This situation is the so-called delayed ACK. It is called delayed because TCP B has nothing to send except ACKs and it has time to respond in a lock-step fashion. The normal delay for TCPs to send delayed ACKs is 500msec. It is possible to acknowledge multiple packets with one ACK if they come quickly enough.



Note that the Window field limits the amount of data that can be sent without ACK. The receiver always updates its Window value. Normally TCPs have a window of around 10 segments (around 15Kbytes of window).

The window can be imagined as a sliding region on the TCP data stream. If the window size is for example 10 packets, a sender can send up to 10 packets without receiving acknowledgement. It can also receive up to 10 packets without the protocol user reading those packets from the TCP buffer. When reading, it can continue receiving up to 10 packets after a packet is lost. We will see this in mechanism in action in the next example.

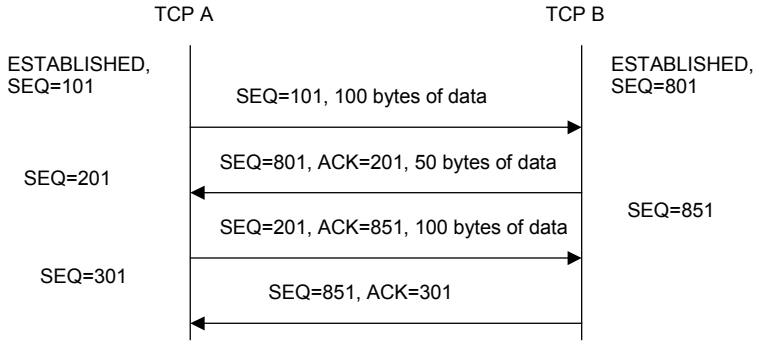
ACK can also be used as negative ACK (NACK). In this case the receiver simply communicates the last sequence number in the stream that the receiver expects.



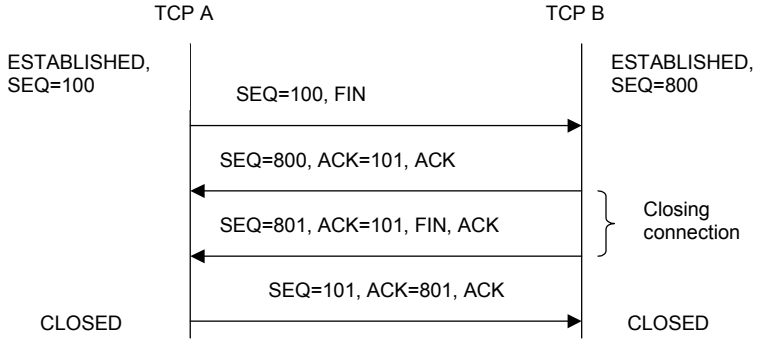
The picture shows a case when TCP B's receive window was large enough to hold packet 2 till the retransmitted packet 1 arrives. Although TCP B didn't acknowledge the lost first packet (SEQ=101), it buffered the second packet (SEQ=201) in its window. When TCP A resent the lost first packet (packet 4), TCP B already had the second packet in its window buffer and acknowledged both packets (ACK=301).

If the receive window is not large enough, packet 2 is dropped and ACK=202 is sent after the retransmitted packet 1.

Each TCP segment has piggybacked ACKs. This means that ACKs can be transmitted on top of packets carrying data segments. In the next example both sides transmits continuously.



The protocol is closed with FIN segments.



Note that TCP A goes to CLOSED state only a certain time after it sent its last packet.

There are many more possible dataflows using sequence numbers and flags in creative ways to handle race conditions and introduce optimizations.

1.7 Services on top of TCP/IP

Internet services are run on top of the IP stack. As the applications can access UDP or TCP, application protocols that these services use are based on UDP and TCP and not on IP. There exist low-level services using ICMP (like ping or traceroute). Some IP-based services are listed here.

- Domain Name Service (DNS). DNS allows using names instead of IP addresses when locating IP nodes. DNS uses hierarchical set of servers (called name servers) that are able to resolve Internet names to IP addresses. DNS is based on UDP.
- E-mail system. The e-mail system uses multiple protocols (SMTP to send and forward messages between Message Transfer Agents (MTAs), POP or IMAP to access received mails by the mail clients). These protocols are based on TCP.
- File transfer service. File transfer service is based on FTP protocol. FTP protocol is spoken between FTP clients and FTP servers. FTP is based on TCP.
- World Wide Web. WWW uses Hypertext Transfer Protocol (HTTP) to transfer hypertex web material like HTML pages. We will see HTTP more in detail during this course. HTTP is based on TCP.

2. THE SOCKETS PARADIGM

We have talked about network protocols so far. At this point you may get interested, how to access the network protocol engines from your applications. The most popular solution is the sockets paradigm.

The sockets model was first introduced in the Berkeley UNIX Software Distribution in the early 1980s, the concept of sockets was originally designed as a method of interprocess communication. Soon, however, it grew to encompass network communications as well. The Berkeley sockets model conceptualizes network communications as taking place between two endpoints, or sockets. Analogies have been drawn that compare plugging an application into a network to plugging a handset into the telephone system, or an appliance into an electrical system.

Most sockets programs — under Unix or Windows — utilize a client/server approach to communications. Rather than trying to start two network applications simultaneously, one application is theoretically always available (the server) and another requests services as needed (the client). The server creates a socket, "names" it so that it can be identified and found by a client, and then listens for service requests. A client application creates a socket, finds a server socket by name or address, and then "plugs in" to initiate a conversation. Once a conversation is initiated, data can be sent in either direction. The client can send data to and receive data from the server, and the server can send data to and receive data from the client.

It is important to note that the concept of a socket is purely an abstraction used in the socket API. It's not necessary that client and server applications both be written in the socket model in order to communicate. Socket interface is a source code interface, a way for programmers to envision network connections. It's not a protocol or network type. A program written in the socket model can communicate with many different types of systems, as long as they use the same protocols.

The socket model can be bound to wide variety of network protocols. Although we will use the Internet protocols in our examples, bear in mind that any network protocol can be mapped to the socket programming model.

A socket can be mapped to network protocols in two ways:

- Filters can be used to select a protocol from the available protocol suites according to certain criterias. For example I can say that I want a datagram protocol from the IP protocol family. This will select UDP (if available)
- It can be a raw socket where the protocol is selected explicitly during socket creation. Users of raw sockets have more extensive access to the protocol's features; there is no additional layer of abstraction between the socket user and the protocol stack. This also introduces larger dependency on the protocol's availability, version or implementation details.

The usage of sockets is demonstrated by a set of programs in the sockets/ subdirectory. These C and Java programs implement simple TCP and UDP client-server pairs.

The socket programming model differentiates between a client and server socket. The difference is meaningful only in connection-oriented protocols like TCP where the client socket represents the initiator of the connection while the server socket represents the responder of the connection request. When a server socket is created, it is *bound* to a

certain port number then the server process *listens* on this port and *accepts* requests. When a connection request is received (for example TCP stack accomplishes the 3-way handshake) the server socket spawns a new client socket. The real communication happens between the client socket on the initiator machine and the client socket created by the server socket.

The mechanism described above is only meaningful for connection-oriented protocols. There is no reason for connection-less protocols like UDP to create an additional client socket for communication because the received or sent data is not part of streams, they are all independent packets.

Client socket is created like the following according to the BSD socket interface:

- The socket handle is created with the *socket* call. The socket function receives the protocol family, the protocol filter and – in case of raw sockets – the identification number of the protocol itself. The result is an unconnected socket that knows which protocol it is bound to.

- The socket is connected with the *connect* call. This is the time when the socket is bound to the communicating party's address and port. It depends on the socket's protocol what happens during the connection phase. Connecting a socket bound to TCP triggers TCP 3-way handshake and fails if nobody is listening on the server end. Connecting an UDP-bound socket has no effect on the protocol layer. In both cases client sockets are assigned port numbers. The client port number is assigned by a mechanism defined by the operating environment so the application programmer does not determine the client port.

- At this stage the socket can be used for communication. Normal *read* and *write* calls can be used on connection-oriented sockets in the same way as these calls are used on files. There is a separate set of functions (*recv, send, recvfrom, sendto*) that work only on sockets. These calls must be used for connection-less protocols.

- The socket is terminated with the *close* method. This frees operating system resources and also accomplishes session termination when connection-oriented protocol is used. Note that normally operating systems close all the sockets when a process is terminated.

The process is slightly different with server sockets.

- The socket is created as with client sockets.

- The socket is bound to a port with the *bind* call. This guarantees that the server socket is assigned the port number that we want. This makes possible for the clients to find the server.

- The server process starts waiting for requests with the *listen* call. There can be many clients trying to connect at the same time. It is possible to specify with the listen call, how many client connection request can queue up for service before the server starts to rejects new connection requests. As you could guess, listen is meaningful only for connection-oriented protocols.

- The server accepts connection requests with the *accept* call. The accept call creates the server-side client socket that can be used to communicate with one particular client. Again, accept can be used only with connection-oriented protocols.

- The same set of functions (*read, write, recv, send, recvfrom, sendto*) is used to accomplish the actual data communication. When connection-oriented protocol is used, these functions receive the server-side client socket created by the accept call. In case of connection-less protocols recv, send, recvfrom, sendto are used directly on the server socket.

- The server-side client socket must be closed when the communication with that particular client finishes. This is relevant only for server sockets bound to connection-oriented protocols.

- At the end of the server process lifetime, the server socket is closed.

Different programming environments like object-oriented class libraries cover the BSD application programming interface with their own abstraction. These abstractions are still based on the socket paradigm, however. Let's see how Java allows access to the BSD API from its class library.

- Sockets can be created with the java.net.Socket class. This is analogous with the *socket* and *connect* calls.

- Communication can be accomplished using the read and write methods of the Java stream classes. This allows easier access than *read* and *write* but is still analogous.

- The socket is at last closed with the close method of the java.net.Socket class

In case of server sockets the socket is created through the java.net.ServerSocket class whose constructor also incorporates the *listen* call. New connection requests are received with the ServerSocket's accept method that maps directly to the *accept* call.

Java introduces a new abstraction in case of sockets mapped to UDP. The java.net.DatagramSocket class represents the UDP socket while the java.net.DatagramPacket class represents the data needed for a *recvfrom* or *sendto* call (IP address, port (target or sender), data to send). The usage of these classes maps exactly to the BSD socket API.

With the socket API we are able to reach the protocol stack functionality up to the transport level. TCP/IP ends here, the higher-level protocols according to the TCP/IP model belong to the application domain. We will now go to higher level to see what kind of communication models are used in the application domain.

3. HTTP AND HTTP-BASED ARCHITECTURES

3.1 Basics of HTTP 1.1

Hypertext Transfer Protocol (HTTP) is one of the success stories of the Internet. The very simple 0.9 (never standardized) version designed by Tim Berners-Lee, the father of the World Wide Web evolved into the quite complex HTTP 1.1 protocol described in RFC2616 and RFC2617.

HTTP is used between the web client (browser) and the web server. HTTP is able to carry any media format (like HTML pages, images) and is able to inform the client and the server about the media type in the request or response. Be careful therefore not to confuse HTTP with HTML: HTTP is a session-layer protocol used to transmit any media content between

the web client and the web server while HTML is a page description language used to represent web pages. HTML content is often transferred on top of HTTP.

HTTP is a session-layer protocol although it supports very simple sessions (one request-response). Additional application logic is needed to create sessions that span multiple request-responses.

HTTP requires a reliable transport. It is most often used on top of TCP but the standard explicitly states that HTTP doesn't mandate the usage of TCP. As it is based on a reliable transport, HTTP just assumes bi-directional, half-duplex bytestream capability and doesn't provide any capabilities in the transport area.

The basics of HTTP are very simple. HTTP is a request-response protocol. The request has the following format:

- HTTP command and resource identification
- HTTP request headers
- Request content

The response is slightly different

- HTTP command status
- HTTP response headers
- Response content

The HTTP command with resource identification and the HTTP command status are always present. There are also some HTTP response headers that are almost always present. All the rest is optional. A typical HTTP transaction consists of HTTP command with resource identification, HTTP request headers, HTTP command status, HTTP response headers and response content.

3.2 URIs, URLs and URNs

HTTP (and a lot of other Internet protocols) use the Uniform Resource Identifier (URI) mechanism to refer to the resource we want to access. URI is composed of the protocol scheme and a resource identifier valid for that protocol scheme. Examples:

http://www.math.uio.no/faq/compression-faq/part1.html
mailto:mduerst@ifi.unizh.ch
news:comp.infosystems.www.servers.unix

If the URI locates the resource by its primary access mechanism (like all the previous examples), it is called URL (Uniform Resource Locator). An URL is an URI therefore. On the other hand, there are URIs that are not URLs like the following:

http://www.google.com/search?hl=en&ie=UTF-8&oe=UTF-8&q=URI&btnG=Google+Search

This is not URL because the resource is not addressed directly, rather than by some associative addressing. Also URIs are the URLs that are used only for identification purposes without having real resource behind the address. For example if you try to reach the popular http://www.w3.org/1999/XSL/Transform URI (namespace URI of XML Schema

Transformations language), you will find that there is no meaningful resource behind the URI. Still, this URI is used to identify XSLT documents, whenever the schema processor sees this URI, it recognizes the XSLT elements. (This course material doesn't deal with XSLT. It is enough for you to know that XSLT documents are associated with an URI and that URI has no real resource behind itself).

Unified Resource Name (URN) is another special case of URI. URNs are persistent, location-independent, resource identifiers. A URN by itself cannot be used to locate the resource; an additional database is needed by the system processing the URN that maps the URN to network address. Example URN is: URN:foo:a123,456. This means that we are referring to resource a123,456 from the foo namespace.

URI is specified by RFC2396.

3.3 HTTP examples

Let's see it an HTTP example (recorded between Netscape 7.1 browser and Apache webserver)!

The request (¶ represents CR/LF. Long lines were wrapped over for easier readability):

```
GET /test/example.html HTTP/1.1¶
Host: localhost¶
User-Agent: Mozilla/5.0 (Windows;U; Windows NT5.0; en-US; rv:1.4) Gecko/20030624
Netscape/7.1 (ax)¶
Accept: application/x-shockwave-flash,text/xml,application/xml,
application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,video/x-mng,image/png,
image/jpeg,image/gif;q=0.2,*/*;q=0.1¶
Accept-Language: en-us,en;q=0.5¶
Accept-Encoding: gzip,deflate¶
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7¶
Keep-Alive: 300¶
Connection: keep-alive¶
¶
```

Note the additional CR-LF after the last header. The header block is closed by an empty line then the request content follows. In our case there is no request content so the request finishes with the empty line).

The response looks like the following:

```
HTTP/1.1 200 OK¶
Date: Fri, 26 Sep 2003 12:11:49 GMT¶
Server: Apache¶
Last-Modified: Fri, 26 Sep 2003 12:08:03 GMT¶
ETag: "2041b-54-66790ec0"¶
Accept-Ranges: bytes¶
Content-Length: 84¶
Connection: close¶
Content-Type:text/html; charset=ISO-8859-1¶
¶
<html><head><title>Example</title></head>¶
<body><h1>Example</h1></body></html>¶
```

Let's see what happened during this request-response!

1. The request starts with the HTTP command line. This time it was a GET that instructed the server to send back the addressed resource. The resource identifier comes after the HTTP command and says that we want the resource called

/test/example.html. The line is closed with the protocol identifier that is used for versioning. This identifies our web client as HTTP 1.1-capable.

2. A set of request headers follows. Request headers convey information about the request. Interpreting these headers is optional. We will talk about request headers later.

3. An empty line closes the header block. This is followed by the request content, if any. This case we have no request content because the GET command doesn't use request content.

4. The response starts with identification of the HTTP version that the server speaks and the status of the request. The status is always a 3-digit number followed by a short message about the meaning of the status code. This time the request was executed correctly.

5. Response headers follow. The most important response header is the Content-Type: it tells the web client how to interpret the returned response content. This time it was text/html which tells the browser that the content must be rendered as HTML page.

6. Empty line closes the header block. The content comes after this empty line. In this case it was a textual HTML file but it may well be a binary content like an image because the TCP stream is 8-bit clean. The Content-Length header ensures that the web client knows, how many bytes it should read from the TCP stream to get the content.

7. The TCP socket is closed after the response content. HTTP has a mechanism to ensure that the TCP stream is reused for multiple requests if possible. By default, HTTP 1.1 uses persistent connections so the client and the server doesn't close the TCP stream but uses it for multiple HTTP request/responses. In our case the Connection: close header means that the server doesn't wish to use this TCP stream anymore and it will close it after the response content was sent.

Let's see now another case when request content is sent! The response is not interesting here, hence it was not included. I also removed the request headers that we have seen at the previous example.

```
POST /cgi-bin/test-cgi HTTP/1.1¶
Referer: http://localhost/test/phbook1.html¶
Content-Type: application/x-www-form-urlencoded¶
Content-Length: 55¶
¶
firstname=John&familyname=Doe&phonenumber=%2B1777111111¶
```

In this case we used the POST command. The difference between the GET and the POST command is that POST carries request content. In this case the request content is a WWW form data (see the Content-Type header). The Content-Length is always present because closing the TCP stream cannot signal the end of the request content. The request content can also be binary so you can upload any media type.

Let's talk a bit about the request and response headers. RFC2616 defines 49 headers that I clearly cannot present here. I will describe shortly the headers that we saw in the examples.

Header name	Meaning
Accept	The client sends the media types it is able to render. The media types can also be weighted, for example application/xhtml+xml,text/html;q=0.9 means that the application/xhtml+xml media type is favored by the weight 1.0 while text/html is favored by the weight 0.9. If the resource is available in both media types, the more favored should be served.
Accept-Charset	Defines the character sets that the web client is able to handle.
Accept-Encoding	Specified the encoding that this web client can accept. For example gzip, deflate means that the content can be compressed by the gzip or Unix compress tools when transmitted over the network.
Accept-Language	Specifies the favored language of the content. For example en-us,en;q=0.5 means that the US English is favored by the weight 1.0 while any other English is favored by the weight 0.5.
Accept-Ranges	Declares that the server accepts byte-range requests for this resource. If the transmission of this resource breaks for some reason, the web client can request the remaining part of the resource with a new request containing a Range header.
Connection	Defines the headers that must not be forwarded by a proxy if there's a proxy server between the web client and the web server. If the header value is "close", it means that the TCP connection must be closed after this request-response is finished.
Content-Length	Length of the request or response content in bytes.
Content-Type	Defines the media type of the request or response content.
Date	The Date general-header field represents the date and time at which the message was originated. The format must conform to RFC1123.
ETag	Entity tag, a hash value of the content to provide alternative cache comparator mechanism (see Last-Modified)
Host	The Host request-header field specifies the Internet host and port number of the resource being requested, as obtained from the original URI given by the user or referring resource. We will see later that proxy servers need the Host information.
Keep-Alive	Proprietary header, not part of HTTP 1.1 standard. It says that the web client will try to keep the TCP stream open for 300 msec so that the stream can be reused for another HTTP request.
Last-Modified	Time when the resource was last modified (for example file modification date). Used for cache control, if the web client caches the response content, next time it can send an If-Modified-Since request header then the server can respond with 304 Not Modified status and no response content. In this case the client can use the content from the cache. If the resource was modified, the server responds with 200 OK, sends the new update time with the Last-Modified header and sends the updated content in the response content.
Server	Identifies the web server that generated the response
User-Agent	Identifies the web client software that sent the request

GET and POST are by far the most widely used HTTP commands. I introduce here shortly the other, less used commands.

Command name	Function
OPTIONS	Allows examining the server response to a certain combination of resource identifier and request headers. The resource is not retrieved but analysis of the returned response headers convey information about the server.

HEAD	Same as GET except that the response doesn't contain the requested resource. Can be used to inspect the resource (like modification date) without retrieving it.
PUT	Similar to POST but it requests the server to upload the request content under the given resource identifier. Difference between POST and PUT that POST represents a static resource identifier that consumes the request content while PUT states that this resource identifier should be used to access the content sent in the request content file. Used for file upload.
DELETE	Requests delete of the resource identifier.
TRACE	The request (command, header and content) is sent back in the response with a 200 OK status. Useful to find out what is the HTTP request that the server receives (it is not evident if there are proxies between the client and the server)
CONNECT	Used to switch a proxy into tunnel mode, used between a web client and a proxy server. The remaining part of the communication will not be interpreted as HTTP request/responses but will be blindly relayed to the address that follows the CONNECT command. Useful if the HTTP proxy is used to relay some other protocol like secure communication protocols.

3.4 Authentication in HTTP

Authentication can be done at basically every level of the protocol stack. HTTP also has an authentication mechanism. It requires careful consideration to determine whether HTTP-level authentication is really the solution we need.

Let's see a transaction where this authentication is used. In the first request-response the server tells the web client that this resource is protected (irrelevant headers are deleted) and HTTP Basic authentication is required to access it.

```
GET /test/protected/example.html HTTP/1.1¶
Host: localhost¶
Keep-Alive: 300¶
Connection: keep-alive¶
¶

HTTP/1.1 401 Authorization Required¶
Date: Mon, 29 Sep 2003 14:44:55 GMT¶
Server: Apache¶
WWW-Authenticate: Basic realm="protected domain"¶
Content-Length: 401¶
Connection: close¶
Content-Type: text/html; charset=iso-8859-1¶
¶
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">¶
<html><head>¶
<title>401 Authorization Required</title>¶
</head> …response content is a web page telling the user what happens …
```

In its response the server says that authentication is requested for the realm called "protected domain". The realm is the URL and the directories below this URL and the server administrator who set up the realm gave the realm name. The realm name is intended to be displayed to human users and should be descriptive for the users so that they know what kind of username/password should be entered.

Seeing 401 status, the browser displays a message box for the username and the password and displays the realm name. The user enters the username and the password and the request is repeated.

```
GET /test/protected/example.html HTTP/1.1¶
Host: localhost¶
Keep-Alive: 300¶
Connection: keep-alive¶
Authorization: Basic dXNlcjpzZWNyZXQ=¶
¶

HTTP/1.1 200 OK
…
```

The web client requests the resource again, this time with authentication. The information submitted in the Authorization header is actually the “user:secret” string (username and password) encoded in the so-called Base64 encoding. This encoding can be easily decoded and the plaintext username and password recovered. The HTTP Basic authentication is therefore unsafe, anybody who can listen to the network traffic can find out usernames and passwords submitted with basic authentication.

HTTP transactions are independent and if the resource is accessed again, username and password must be supplied again. Browsers hide this fact by remembering the realm URL, username and password and supplying automatically when a resource in the realm is accessed.

HTTP offers another authentication, the HTTP Digest authentication. In this case the password itself is not transmitted over the network, only a value (so-called hash) that is generated from the password. HTTP Digest authentication is significantly harder to break but it offers only limited protection against dictionary or replay attacks. We will talk more about security later.

To understand this method, we have to know that there are hash algorithms that can take byte streams of practically unlimited length and they can create “fingerprints”, secure checksums of the byte streams. Secure checksum means that small change in the byte stream causes drastical changes in the fingerprint in such a way that it is very hard to compute. A good hash function guarantees, that if a certain byte stream gives fingerprint X then it is very hard to change the bytestream in such a way that the fingerprint is still X. A good hash function therefore can be used to guarantee the authenticity of a byte stream and it also guarantees that knowing the fingerprint and the bytestream, the attacker needs very significant amount of computation to create a fake bytestream that yields the same fingerprint.

The MD5 (RFC1321) used in HTTP digest authentication generates 128-bit fingerprint.

Let’s see an example! (unrelevant headers were edited out again)

```
GET /test/digest/example.html HTTP/1.1¶
Host: localhost¶
Keep-Alive: 300¶
Connection: keep-alive¶
¶

HTTP/1.1 401 Authorization Required¶
Date: Tue, 30 Sep 2003 10:59:08 GMT¶
WWW-Authenticate: Digest realm="digest",
nonce="Ithn54nIAwA=6e091a444ef5029635986909bd2e5156ff865bfe", algorithm=MD5,
domain="/test/digest/", qop="auth"¶
Content-Length: 401¶
Connection: close¶
Content-Type: text/html; charset=iso-8859-1¶
¶
… HTML content comes here informing the user that authentication is needed …
```

The web client innocently requested for the protected resource. The server talked back that the Digest authentication is required to access the resource. At this point the web client prompts the user for username and password.

The password is protected by the nonce mechanism. Nonce is a string of randomly generated characters. The nonce is appended to the password when the hash is generated. The usage of nonce requires the attacker to know the password because the hash of a session recorded earlier doesn't help, the hash cannot be sent again because the nonce changed meanwhile. We will return to the computation of the hash later.

Having the nonce, the realm, the username and the password the client can now repeat the request with proper authentication.

```
GET /test/digest/example.html HTTP/1.1¶
Host: localhost¶
Keep-Alive: 300¶
Connection: keep-alive¶
Authorization: Digest username="user", realm="digest",
nonce="Ithn54nIAwA=6e091a444ef5029635986909bd2e5156ff865bfe",
uri="/test/digest/example.html", algorithm=MD5,
response="748d48d70b2e8095295e128461312566", qop=auth, nc=00000001,
cnonce="082c875dcb2ca740"¶
¶

HTTP/1.1 200 OK¶
Date: Tue,30 Sep 2003 10:59:14 GMT¶
Authentication-Info: rspauth="c0d2c87a3a61cf82c6096f5cb657e267",
cnonce="082c875dcb2ca740", nc=00000001, qop=auth¶
… normal response continues …
```

The client also generates a nonce, called cnonce. The response value is calculated as appending the access method (GET or POST), request URL, username, the realm name, the password, the nonce and the client nonce separated by colon (:) and applying the hash function on it (see exact calculation in RFC2617, it is a bit more complicated). The result is sent in the response field. When receiving this request, the server looks up the user's password in its authentication database and calculates the hash value. If it matches with the hash vaue sent by the client, the authentication is accepted.

The server can also prove that it knows the user's password (it would be too easy for servers to accept everything and snoop on users). The server calculates the same digest as the client did except that the access method is not included and the hash is sent back to the client in the rspauth field.

The security of this solution against replay attacks (when the attacker intercepts a valid request-response and sends it to the server on his/her behalf) depends on the lifetime of the nonce. As long as the same nonce values are used, the attacker can replay a previously recorded transaction. Clients should use different client nonces each time they issue a new transaction and servers should update the server nonces regularly with the nextnonce parameter of the Authentication-Info header. The nc parameter counts the requests that were issued using the same server nonce so the server can implement a nonce update policy.

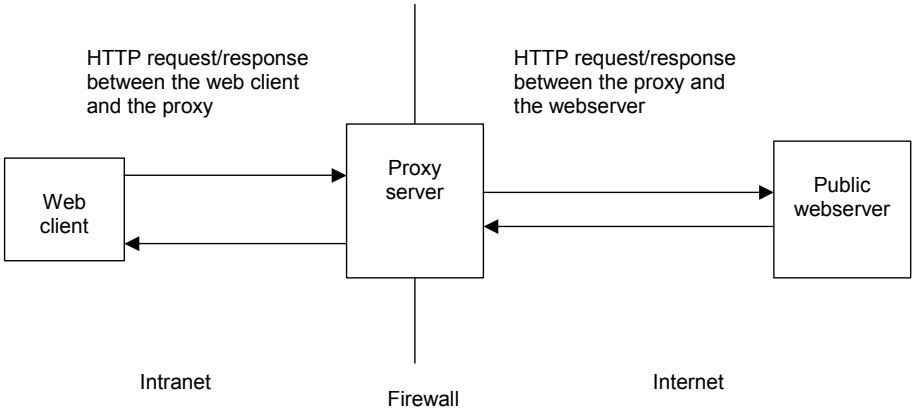
The HTTP Digest authentication offers significantly better security than the Basic authentication. It should be noted, however, that its main function is to protect the user's password from eavesdroppers. HTTP Digest solves this problem quite efficiently although it is still susceptible for dictionary attacks (if the user chooses an "easy" password (short

password, well-known names, etc.) the attacker can compute the digest for a series of possible passwords and find out the digest that matches). We will see more efficient security technologies later.

3.5 HTTP-based architectures

HTTP (along with the e-mail protocols) is the most widely deployed protocol on the Internet. HTTP has been adapted to work with corporate security solutions. Key element of any Internet security architecture is the corporate firewall. This firewall separates the corporate intranet from the public Internet. Unfortunately the basic principle in security also holds: security and accessibility are contradictory requirements. In our case this means that a hole needs to be opened on the corporate firewall for outgoing HTTP traffic. The most secure way of doing it is to install a proxy server.

Proxy server is a program that is connected to both the intranet and the public Internet. It receives requests from intranet clients and responds to them, as a web server would do. On the other end, it issues the request on the web clients' behalf on the public Internet and when the response is received, it is returned to the web clients. The proxy server is therefore very safe because there is no direct routing of IP packets between the intranet and the public Internet and the proxy server can enforce policy on HTTP communication because it interprets the HTTP protocol. Web client security weaknesses can be controlled in one place, in the proxy server. You can see the architecture below.



Note that proxies can be chained; there can be multiple proxy servers along the way before the request reaches the webserver.

The proxy operation raises an interesting issue. So far the target URL was implied and was actually not transmitted by HTTP. Whoever was the receiver of the TCP connection request (target IP and port) served the request and only the directory/file part of the URL was sent in the request. This clearly cannot work in the proxy case because it is the proxy that receives TCP connections for all web clients indepently of the target webserver. If HTTP is used with proxy, the protocol is slightly modified for proxy operation.

Let's see an example!

```
GET http://172.24.169.107/test/example.html HTTP/1.1¶
Host: 172.24.169.107¶
Keep-Alive: 300¶
```

```
Proxy-Connection: keep-alive¶
¶



HTTP/1.1 200 OK¶
Date: Wed, 01 Oct 2003 12:27:56 GMT¶
Proxy-Connection: close¶
Via: 1.1 proxy1 (NetCache NetApp/5.3.1R4), 1.1 proxy2 (NetCache NetApp/5.2.1R1D4)¶
¶
… response content …
```

As we could see, there are minor differences. The request command now carries the full URL because it is needed by the proxy to connect to the right server. The Connection headers now instruct the proxy-web server connection details, instead of the web client-proxy (the header is different, instead of Connection, Proxy-Connection is used). The proxies also add and modify request and response headers, for example in our case the two proxies that participated in serving the request identified themselves in the Via header.

The widespread deployment of HTTP infrastructure often results in solutions when HTTP is used to transfer some content that HTTP does not really fit. It is possible, for example, to select HTTP as a transfer protocol in the RealAudio player although reliable protocols like TCP are not the best for transferring real-time data (timing is more important in real-time applications than the guaranteed delivery of the packets). On the other hand, one has little choice if an application needs to be deployed in a corporate environment where HTTP may be the only way out to the public Internet.

We will see several “creative” uses of HTTP when we discuss application-level protocols.

4. SECURITY SOLUTIONS ON THE INTERNET

In this chapter we are going to talk about security solutions on the Internet. There are many aspects of network security; we will pick this time only the ones affecting network protocols. These aspects are the following:

- Secure authentication so that the client and the server can mutually authenticate to each other even if eavesdroppers are present or different kinds of attacks can be assumed.
- Secure communication so that eavesdroppers can’t see the content being communicated.
- Message integrity is guaranteed; the content transferred cannot be manipulated so that the receiver doesn’t notice it.
- Non-repudiation may be provided. This means that some kind of signature is attached to the message that proves that the sender did agree with the content of the message

4.1 Very short introduction to cryptography

A whole course can be devoted to cryptography but we have only limited time here. We will review shortly the common algorithm classes and terms used in today’s cryptography.

4.1.1 Symmetric ciphers

A cipher algorithm receives unencrypted message (called cleartext or plaintext) and turns it into encrypted message called ciphertext. The deciphering part of the algorithm receives the ciphertext and turns it into plaintext. The ciphering/deciphering process is controlled by a piece of information called the key. Keys can come in several formats, for example in the well-known book cipher (when a passage of a book is used to encrypt/decrypt the message) the key consists of the title and publishing date of the book, the page and the paragraph number. In computer systems the key is normally a (quite large) number.

Every cyptosystem is susceptible of the so-called brute force attack. In this attack we just try all the possible keys and we trust that our computing power is large enough to find the right key in reasonable timeframe. There is only one way to defeat brute force attack: the key space (possible values of the keys) must be large enough to make this attack unfeasible. As the power of computing systems grow, key spaces that were thought large enough previously become susceptible to brute force attacks.

The cipher is symmetric if the same key is used to encrypt and decrypt the message. The book cipher for example is a symmetric cipher. Before the Diffie-Hellman public-key cipher was invented in 1976, all known ciphers were symmetric ciphers.

The weak point of any symmetric cipher is the key exchange. There must be a secret channel to communicate the key to the receiver of the ciphertext. If the key is compromised when transmitted on the secret channel, the symmetric cipher offers no protection.

In today's cryptosystems the symmetric ciphers are implemented as complicated combinations of bit reordering, shifts and XOR operations. Normally the cipher contains several (10-20-30) rounds of these basic operations. Design of symmetric ciphers is not exact science; the strength of the cipher can be determined after several years of attacks. Symmetric ciphers are very fast

Widely used symmetric ciphers in today's cryptosystems are DES (56 bit keylength, already obsolete), IDEA (128 bit keylength), triple-DES (3 rounds of DES with different keys, yields only 112 bits of effective keylength) and RC4 (1 byte to 256 bytes keylength, widely used with 16 byte keys (128 bits)).

4.1.2 Asymmetric (or public-key) ciphers

The introduction of the Diffie-Hellman cipher brought a breakthrough in cryptography. Unlike symmetric ciphers the Diffie-Hellman algorithm uses two separate keys. One key can be used to encrypt, the other is to decrypt the message. The two keys are generated together during the key generation process and they are related in such a way that (presumably) it is very hard to calculate one key from the other. One key can be distributed therefore freely for the communicating parties and they can use this freely available (public) key to encrypt messages to the recipient. Knowing the public key doesn't allow decrypting the ciphertext, the recipient must have the other half of the key pair (the private key) to obtain the plaintext.

There is another interesting use of the public-key ciphers when the algorith is used for digital signatures. In this case the signer encrypts the signature with his/her private key and sends the encrypted signature to the public domain. Anybody having the sender's public key is able to verify that the signature was indeed created by a person having access to the private key.

Asymmetric ciphers are very slow. It is not feasible to use them on larger messages; therefore they are used in combination with symmetric ciphers when the asymmetric cipher is used for key exchange of the symmetric cipher.

The most popular asymmetric cipher is RSA. RSA with 1024 bit of keylength is considered safe for commercial use.

4.1.3 Hash algorithms

We have already seen a hash algorithm (MD5) in section 3.4. Hash algorithms are essentially “secure checksums”. The problem is to create a checksum on a byte stream. In the security world, we have two additional tough requirements.

- Small changes in the document must cause dramatic changes in the checksum
- It must be very hard to create a document that matches a certain checksum.

Clearly, simple CRC or checksum algorithms are not able to satisfy these requirements. In the secure communication we use special checksum algorithms that we call hash or digest algorithms. The most popular hash algorithms are MD5 (128 bit checksum) and SHA (160 bit checksum).

4.2 The TLS (SSL) security protocol

4.2.1 Motivation and basics

It is clear that cryptography algorithms are only part of the security framework. These algorithms have to be used together in a secure and efficient way so that applications can rely on a consistent and compatible framework. Netscape managed to create an industry standard in the area called Secure Sockets Layer (SSL) which later got standardized under the name of Transport Layer Security (TLS). There are minor differences between TLS and SSL but the two protocols don’t interoperate. We focus on TLS in this course. TLS specification can be found in RFC2246.

The primary goal of the TLS Protocol is to provide privacy and data integrity between two communicating applications. The security layer is missing from the OSI model (it is still heavily debated where is the optimal place of security in the protocol stack), the TLS has elements in the session and the presentation layers.

The protocol is composed of two layers: the TLS Record Protocol and the TLS Handshake Protocol. At the lowest level, layered on top of some reliable transport protocol (e.g., TCP), is the TLS Record Protocol. The TLS Record Protocol provides connection security that has two basic properties:

- The connection is private. Symmetric cryptography is used for data encryption (e.g., DES, RC4, etc.) The keys for this symmetric encryption are generated uniquely for each connection and are based on a secret negotiated by another protocol (such as the TLS Handshake Protocol). The Record Protocol can also be used without encryption.
- The connection is reliable. Message transport includes a message integrity check using a keyed message authentication code. Secure hash functions (e.g., SHA, MD5, etc.) are used for MAC computations. The Record Protocol can operate

without a MAC, but is generally only used in this mode while another protocol is using the Record Protocol as a transport for negotiating security parameters.

The TLS Record Protocol is used for encapsulation of various higher-level protocols. One such encapsulated protocol, the TLS Handshake Protocol, allows the server and client to authenticate each other and to negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data. The TLS Handshake Protocol provides connection security that has three basic properties:

- The peer's identity can be authenticated using asymmetric, or public key, cryptography (e.g., RSA). This authentication can be made optional, but is generally required for at least one of the peers.
- The negotiation of a shared secret is secure: the negotiated secret is unavailable to eavesdroppers, and for any authenticated connection the secret cannot be obtained, even by an attacker who can place himself in the middle of the connection.
- The negotiation is reliable: no attacker can modify the negotiation communication without being detected by the parties to the communication.

One advantage of TLS is that it is application protocol independent. Higher-level protocols can layer on top of the TLS Protocol transparently. The TLS standard, however, does not specify how protocols add security with TLS; the decisions on how to initiate TLS handshaking and how to interpret the authentication certificates exchanged are left up to the judgment of the designers and implementors of protocols which run on top of TLS.

TLS allows a wide variety of cipher and hash protocols to be used so it doesn’t depend on a certain cryptographic algorithm.

4.2.2 TLS Record Protocol

The TLS Record Protocol is a layered protocol. At each layer, messages may include fields for length, description, and content. The Record Protocol has the following layers:

- Fragementation, cuts the input byte stream into manageable blocks
- Compression, it has to be done here because the encrypted blocks cannot be compressed. Optional.
- Inserting message authentication code (MAC). The message authentication code is computed from the MAC key and the message itself. The MAC is normally implemented using a hash algorithm where the key and the message are both used as input for the hash algorithm). The MAC key is created during the key exchange.
- Encryption

When the data is received, the process is done in the opposite direction: decryption, checking message authentication code, decompression and reassembly.

TLS defines four possible protocols on top of the Record Protocol: the handshake protocol, the alert protocol, the change cipher spec. protocol, and the application data protocol. We will see them later.

4.2.3 The TLS handshake protocol

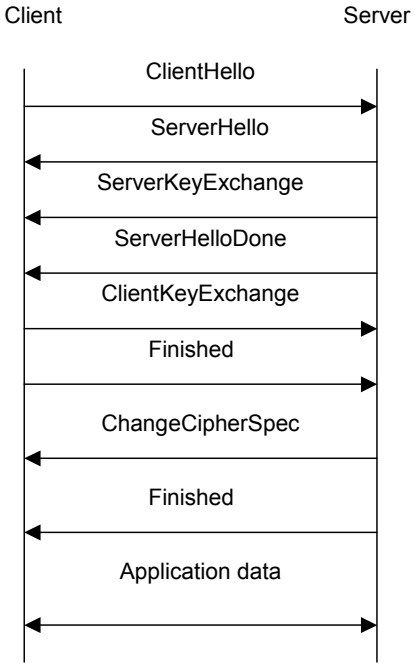
The TLS Handshake Protocol consists of a suite of three sub-protocols that are used to allow peers to agree upon security parameters for the record layer, authenticate themselves, instantiate negotiated security parameters, and report error conditions to each other. The Handshake Protocol is responsible for negotiating a session, which consists of the following items:

- Session identifier, an arbitrary byte sequence chosen by the server to identify an active or resumable session state.
- Peer certificate, X509v3 certificate of the peer. This element of the state may be null. We will talk about certificates later.
- Compression methodt, he algorithm used to compress data prior to encryption.
- Cipher spec. Specifies the bulk data encryption algorithm (such as null, DES, etc.) and a MAC algorithm (such as MD5 or SHA). It also defines cryptographic attributes such as the hash size.
- Master secret. 48-byte secret shared between the client and server.
- Is resumable. A flag indicating whether the session can be resumed after it was paused for some reason. When the protocol is resumed, key negotiation doesn't happen, session data negotiated for the paused session will be used.

The TLS Handshake Protocol involves the following steps:

- Exchange hello messages to agree on algorithms, exchange random values, and check for session resumption.
- Exchange the necessary cryptographic parameters to allow the client and server to agree on a premaster secret.
- Exchange certificates and cryptographic information to allow the client and server to authenticate themselves.
- Generate a master secret from the premaster secret and exchanged random values.
- Provide security parameters to the record layer.
- Allow the client and server to verify that their peer has calculated the same security parameters and that the handshake occurred without tampering by an attacker.

The basic handshake is the following:



The ClientHello message is sent when the client wants to establish a new session. The message contains timestamp, 28-byte random number and session identifier that is empty if a new session is to be established (used to re-establish already negotiated sessions). The message also contains the list of cipher suites and compression methods that the client supports. A cipher suite is a set of cryptographic algorithms that are used together during the session. This tipically means a public-key cipher, a symmetric cipher and a hash algorithm.

Note that there is a default cipher suite, TLS_NULL_WITH_NULL_NULL that provides no encryption and no message integrity protection. This ensures, however, that the TLS Record Protocol is able to carry handshake messages even when the encryption parameters have not been negotiated.

In response to the ClientHello, the server sends ServerHello. This message contains the session ID that the server assigns to this session and one single cipher suite that the server selects according to the client cipher suite and the cipher suites that the server supports. This is presumably the strongest cipher suite supported by both the client and the server. Although TLS is pretty safe against attacks trying to enforce the communicating parties to a weaker cipher suite, higher-level layers must always check whether the selected cipher suite is strong enough for the value of the data that TLS protects.

The server continues with the ServerKeyExchange message. This message contains the public key of the server. The client must know the server's public key in order to be able to use the public-key cipher for the key exchange. We will return to the server public key problem later.

At this point the server sends ServerHelloDone telling the client that the server finishes this stage of the handshake.

Critical point of the handshake is the next message when the client sends the ClientKeyExchange message. The client generates a random number (the 3rd random number during the protocol), encrypts is with the server's public key and sends it to the server. At this point both the client and the server knows three random numbers. The first two random numbers may have been sent without encryption (if the original cipher suite was TLS_NULL_WITH_NULL_NULL which is the default for new sessions) but the eavesdropper cannot obtain the third one because the server's prvate key would be needed for that. The client and the server now shares a secret that the eavesdropper cannot decipher.

The client finishes this round with a Finished message. This message contains a hash that is calculated from all the handshake messages that the client sent and received before the Finished message.

The protocol is now ready to switch on the encryption. The 3 random numbers exchanged during the handshake are used to generate key material. The key material contains all the keys for the symmetric ciphers (different keys are used in client->server and server->client directions) and the keys for the MAC computations (again, there are two MAC keys, one for each direction). The random numbers are used as an input of a quite complicated hash calculation so it is not relevant that the attacker saw the first 2 random numbers.

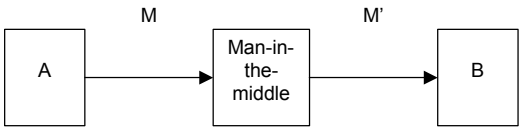
The server updates the session variables and sends out a ChangeCipherSpec message. When the client sees this message, it also updates the session variables. At this point the TLS Record Layer protocol starts to use the negotiated cryptographic parameters.

The next message, a Finished from the server goes already encrypted. In this message the server sends the client a hash of all the messages it sent and received during the session. If the attacker manipulated any handshake messages (e.g. it tried to force the client and the server to use a weaker encryption), the hash will not match on the client and server side.

The handshake is finished now; the TLS Record Layer protocol will carry application data in both directions.

4.2.4 Certificates

If you remember, there is a point during the handshake when the server sends its public key with the ServerKeyExchange message. This step is vulnerable against the so-called man-in-the middle attack. Let's suppose that the attacker is able to insert itself into the communication chain between A and B and beside reading all the messages, it is able to modify them or even to generate new, fake ones.



This is actually a very realistic setup with IP; any router administrator is in this position. The man-in-the-middle can then have an own key pair and when the server sends its own public key, it can replace it with the public key of the attacker's keypair. When the client sends back the encrypted 3rd random number, it will be encrypted *for the attacker* that can

decrypt, read, and encrypt again with the server's own public key. A and B managed to make a key exchange in such a way that the attacker knows everything about the session.

There is another interesting attack type, the fake server. During the handshake session in section 4.2.3, the client and the server negotiated a secure channel. What is the guarantee that the session was established to the real server? Let's say somebody has set up a fake server and manipulated the routing tables in such a way that communication to the real server goes instead of to the fake server. The TLS connection will be set up and the data between the fake server and the client will be protected with strong security – except that the valuable client data goes right to the attacker.

Clearly, there must be a way to propagate the server's public key safely to the client. There is a saying in security: if you want security, you have to have something that you trust. You can then build on that trusted point and control untrusted points as well but there's no security if you don't have at least one trusted point in the system.

TLS (and other Internet security solutions) employ the same system as for example the system used in the real-world passports. We believe the authenticity of the passport because there is a whole range of proofs that it was issued by an authorithy. Also, anybody can present my passport (so owning a passport doesn't mean anything) but there is a way to crosscheck the passport with the person who presented it (e.g. by checking the photo, the fingerprint, etc.). There is therefore two expected property of a *certificate* (as we call these passports in the Internet world):

- The validity and the authenticity of the certificate can always be checked with the authorithy that issued it
- There is a way to check that the certificate does indeed belong to the entity that presented it

I would emphasize once again the last point: anyone can present any certificate, certificates are not secret. The authentication mechanism involves presenting the certificate and being able to handle the data that the system accepting the certificate sends. Just being able to send a certificate doesn't prove anything.

The certificate is a data structure consisting of human-readable data on the entity owning the certificate like issuer name, owner name, validity period, etc.) Other data fields of the certificate are not human-readable. The most important such field is the public key of the subject. The issuing authority signs the whole data structure. Signing means that first a hash is calculated on the data structure then this hash value is encrypted by the issuing authorithy's private key. Anybody having the issue's public key can decrypt the hash value then check it against the data structure, the certficate's authenticity can be checked. On the other hand it is impossible to create certificates without access to the issuer's private key. This also means that the issuer's public key must be widely known. The certificate data structure is specified in RFC3280.

Of course this system is as safe as its weakest element. First of all there is a highly trusted body in the picture, the issuer of the certificate that is called Certificate authorithy (CA). CA is trusted in two ways:

- the CA makes sure that the entity the CA creates certificate for is indeed the entity requesting the certificate. In other words, if Citibank wants a certificate, the CA must make sure that it is really Citibank requesting the certificate. Otherwise it is possible

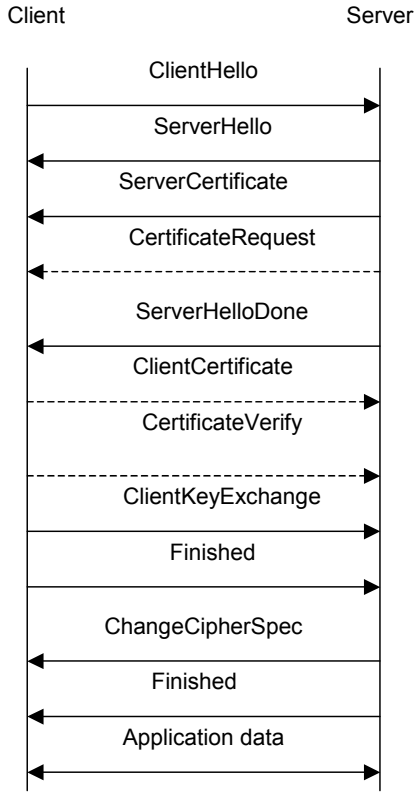
to set up a fake Citibank site and unsuspecting users would innocently feed sensitive data to this site believing in the certificate's human-readable fields.

- the private key of the CA is heavily protected. The CA public key is built into millions of clients; it is extremely hard to revoke a compromised CA public key.

Understanding that CA's business is about trust, it's no wonder that there are only about dozen relevant CAs in the world. You can check them by opening your browser's certificate manager and asking for authorities. If a large organization would like to issue own certificates, it can operate its own CA. This CA would not be recognized by clients (because they know only well-known CAs) so the well-known CA must eventually sign the certificate. This is done by creating a trust relationship between the organization's CA and the well-known CA. The lower-level CA creates and signs the certificates then sends them up to the well-known CA (called root CA) for additional signing. The certificate will therefore have two signatures: first the root CA certifies the public key of the lower-level CA then the lower-level CA signs the certificate data. This is called *certificate chain.*

4.2.5 Using certificates in TLS

We can beat the man-in-the-middle attack with the help of certificates. TLS can use certificates to protect the server public key and to authenticate the client. The protocol flow is slightly modified.



I marked the protocol messages related to client authentication with dashed arrows.

Let's concentrate first on the server authentication! Instead of ServerKeyExchange we see now the ServerCertificate message. At this step the server sends its certificate. The client verifies the certificate (because it has the root CA's public key) then uses the server public key in the certificate to encrypt the 3rd random number. The server is able to continue only if it really has the private key belonging to the server public key in the certificate. Thus our two requirements are satisified: the server public key is authorized by a trusted party (the CA) and the certificate is really crosschecked against the the entity presenting it – in our case the server must have the private key because it cannot continue the protocol.

Let's see how the client authenticated with certificates! These steps are optional; other client authentication methods are often used. If we use client certificates, the server must be configured to request the client certificate. Then the server requests the client's certificate with CertificateRequest message. In response, the client must send its certificate (ClientCertificate) and a proof that the certificate really belongs to the client (CertificateVerify). When verifying the certificate, the client has to encrypt the hash value calculated on all the handshake messages up to this message using its private key and send this value in the CertificateVerify message. The server will have therefore the client's public key from the client certificate and a value encrypted by the client's private key. The server can use the client public key to decrypt the hash then check the hash because the server also knows what were the messages the hash was generated on.

4.2.6 Using TLS with HTTP

TLS is widely deployed on the Internet. Although TLS can be used with any application protocol based on reliable transport (like TCP), it is used most often with HTTP. Any time when the URL's scheme part is https and not http, the web client talks TLS (or its predecessor, SSL) with the server and uses HTTP on top of TLS.

TLS cannot be used with proxies in the same way as HTTP is used. Proxies don't have access to the HTTP traffic because it is encrypted, the proxy won't know any detail on the HTTP traffic. When the proxy transfers TLS traffic, it works as a blind relay. The web client sends CONNECT method in the HTTP command line and tells the proxy what is the target server. The proxy connects to the target server and blindly relays everything that the client sends afterwards and also relays the server's response to the client. This allows the client and the server to accomplish encrypted communication through the proxy.

It is important to note that the details of the traffic, except from the target server, will be hidden even from the proxy administrator

5. DISTRIBUTED APPLICATION ARCHITECTURES

You may have noticed that we are climbing consistently from lower-level network functions to higher abstractions, from low-level network communication to distributed application architectures. So far we talked about network nodes communicating with each other or clients talking to servers. When we talk about distributed application architectures, we don't care about network addresses or ports anymore, we care about services. These architectures always provide the following services.

- Presentation protocol to transmit relatively complex data types like integral types (integers, floats, etc.) and aggragated types composed of integral types like arrays, vectors and even objects.
- Description language to describe a service to service consumers.

- Registry service so that services can be discovered and their descriptions retrieved.

We will examine these patterns in two distributed application architectures; one for Internet-based applications (Web Services) and one for more closely connected networks (CORBA).

6. WEB SERVICES

Web services have received quite an attention – and hype – recently. Web service model allows access to applications over standard web protocols. Web services communication normally happens between two computer systems, opposed to the browser-based model where the communication happens between the end user and the web application. Web services protocols are not intended to be seen by end users, they provide access from a client application to a server application.

The Web services technology suite contains the following elements.

- Simple Object Access Protocol (SOAP) is used as a presentation-layer protocol between the Web Services clients and servers to carry requests and responses.
- Web Services Description Language (WSDL) is used to describe services offered over SOAP.
- Universal Description, Discovery and Integration (UDDI) protocol provides access to Web Services application registry.

SOAP and WSDL are managed by the World Wide Web Consortium (W3C, www.w3c.org) while UDDI is managed by Organization for the Advancement of Structured Information Standards (OASIS, http://www.oasis-open.org).

The Web Services technology suite builds heavily on Extensible Markup Language (XML, a W3C standard itself) so first we take a look at XML.

6.1 Short overview of XML

6.1.1 The XML language

Extensible Markup Language, abbreviated XML, describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them. XML is an application profile or restricted form of SGML, the Standard Generalized Markup Language [ISO 8879]. By construction, XML documents are conforming SGML documents.

XML documents are made up of storage units called entities, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form markup. Markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism to impose constraints on the storage layout and logical structure.

XML allows definition of markups by the XML document’s designer therefore the structure of the XML document can be adapted to the application where it is used. Every markup can have unlimited number of parameters, a content and closing markup. For example:

```
<tag parm1="hello" parm2="hallo">Text, text, text</tag>
```

The name of our markup is “tag”, it has two parameters (parm1 and parm2), the tag’s content is “Text, text, text” and there is a closing tag that is mandatory. If the markup has no content, it can be written in simplified format:

```
<tag parm1="hello" parm2="hallo"/>
```

Markups can contain other markups, eventually the XML document is equivalent to a tree.

```
<tag1>
  <tag2> … </tag2>
  <tag3> … </tag3>
</tag1>
```

XML documents contain processing directives. The most important processing directive is the XML identification tag. Every XML document must start with this processing directive.

```
<?xml version="1.0"?>
```

This tag can be omitted if XML format is the default.

Markups can be associated to namespaces. A namespace is a collection of markups and is associated with a URI. Namespaces allow application of multiple markup collections without the danger of defining the same markup twice in two collections. Namespaces are declared in markups:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:fo="http://www.w3.org/1999/XSL/Format"
   xmlns:svg="http://www.w3.org/Graphics/SVG/SVG-19990812.dtd">
…
</xsl:stylesheet>
```

The example defines three namespaces (xsl,fo and svg), markups from a certain namespace is identified by appending the name of the namespace in front of the markup (like xsl:stylesheet). The namespace names (xsl, fo, svg) are totally arbitrary, it is the namespace URI that uniquely identifies the namespace. For example later on you can write:

```
<st:stylesheet xmlns:st="http://www.w3.org/1999/XSL/Transform"
…
</st:stylesheet>
```

and the two stylesheet elements would still be of the same type because they come from the same namespace. It doesn’t matter that once we referred to this namespace as xsl, the other time as st, the URI is same so they are equivalent.

Note that the namespace URI is used as a unique identifier of the namespace and not as a real network address, if you check the content behind these URIs above, you will not find meaningful content behind them.

If there’s no namespace name when defining a namespace, that namespace will become the default namespace. For example:

```
<stylesheet xmlns="http://www.w3.org/1999/XSL/Transform"
  xmlns:fo="http://www.w3.org/1999/XSL/Format"
  xmlns:svg="http://www.w3.org/Graphics/SVG/SVG-19990812.dtd">
```

```
…
</stylesheet>
```

In this case elements in the first namespace (xsl namespace in the previous example) can be referenced without namespace declaration (like <stylesheet> instead of <xsl:stylesheet>).

6.1.2 XML Schema

XML has two document format specification languages. These languages are able to describe when the document is valid. A valid document conforms to certain rules and these rules are described by the specification language. For example it is possible to specify that certain markup must always have a certain parameter or certain markup must always contain some other markup. The original format specification language was the Document Type Definition (DTD) language. Although DTD was fast and simple, it is being phased out in favor of the XML Schema specification that allows more complex data structures. We have no time for proper explanation of XML Schema, I will demonstrate it rather with an example. Let's say we have an XML format describing books.

```
<?xml version="1.0" encoding="UTF-8"?>
<book isbn="0836217462">
  <title>Being a Dog Is a Full-Time Job</title>
  <author>Charles M. Schulz</author>
  <character>
     <name>Snoopy</name>
     <friend-of>Peppermint Patty</friend-of>
     <since>1950-10-04</since>
     <qualification>extroverted beagle</qualification>
  </character>
  <character>
    <name>Peppermint Patty</name>
    <since>1966-08-22</since>
    <qualification>bold, brash and tomboyish</qualification>
  </character>
</book>
```

This matches the following XML Schema document.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="book">
    <xs:complexType>
        <xs:sequence>
          <xs:element name="title" type="xs:string"/>
          <xs:element name="author" type="xs:string"/>
          <xs:element name="character"
               minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="name" type="xs:string"/>
                <xs:element name="friend-of"
                 type="xs:string" minOccurs="0"
                 maxOccurs="unbounded"/>
                <xs:element name="since" type="xs:date"/>
                <xs:element name="qualification" type="xs:string"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <xs:attribute name="isbn" type="xs:string"/>
    </xs:complexType>
```

```
  </xs:element>
</xs:schema>
```

(Example taken from: http://www.xml.com/pub/a/2000/11/29/schemas/part1.html)

The schema is therefore the formal description of the document format. You can consider a valid XML document matching the schema as an *instance* of the schema, similarly as an object is an instance of a class in an object-oriented system. As with objects and classes, the schema defines the data format (like the class) while the XML document contains the data that conforms to the data format (like the object).

The XML schema processor can validate XML documents using the schema that makes text file processing much easier. Off-the-shelf XML processors have contributed largely to XML's success.

We may also want to organize the data structures defined by the schemas into namespaces. For example we may want to put the book element we have just defined into the http://bookschema.org/book1/ namespace. (remember, this is an URI, it doesn't need to exist as a real network location but it does have to be unique). Then the book element instance would look like:

```
<book isbn="0836217462" xmlns="http://bookschema.org/book1/">
…
</book>
```

When you define the schema, you instruct the parser to put the book schema into the http://bookschema.org/book1 namespace.

```
<xs:schema targetNamespace=http://bookschema.org/book1/
  xmlns:xs=http://www.w3.org/2001/XMLSchema
…
</xs:schema>
```

XML Schema defines several basic datatypes and allows the contruction of complex data structures from these datatypes. The basic datatypes, like string or int have standard serializations (standard representations in the XML text). You can state that your XML element's content conforms to the standard serialization by using the type attribute from the XML Schema instance namespace (http://www.w3.org/2001/XMLSchema-instance). For example we can state that the "arg" element is an integer.

```
<arg xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
     xmlns:xs=http://www.w3.org/2001/XMLSchema
     xsi:type="xs:int">2356</arg>
```

Of course, it is unnecessary to include the namespace declarations with every tag, normally only the XML document's top-level element declares the namespaces only once.

Although we have only scratched the surface of XML Schema, this basic knowledge on XML is enough to understand the Web Services protocols.

6.2 The SOAP protocol

6.2.1 Protocol core

Simple Object Access Protocol (SOAP) is the workhorse of the Web Services architecture. Actors in this architecture exchange SOAP messages. SOAP is transferred on top of a session protocol and deals with the representation of the transferred data (data format), therefore it is a presentation-layer protocol.

World-Wide Web Consortium endorsed SOAP 1.2 as their recommendation. SOAP 1.2, however, is not yet widely deployed. This material will describe SOAP 1.1 which is widely supported in different solutions.

SOAP messages can be used in many messaging scenarios. In SOAP parlance, these scenarios are called Message Exchange Patterns (MEP). Some message exchange patterns are for example: Request with confirmation (when there's no response document, just the status that the request was accepted), Response (when the sender sends an asynchronous response in relation to an earlier request), Single in-Multi out (when the sender sends one request but the responder sends multiple responses). By far the most popular MEP is the Request-Response. The Request-Response MEP is most frequently implemented on top of HTTP.

The SOAP messaging core is very simple. SOAP messages are put inside an envelope and the envelope contains header and body parts.

```
<?xml version='1.0'?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
 …
 </env:Header>
 <env:Body>
 …
 </env:Body>
</env:Envelope>
```

There can be as many header and body parts as needed.

The user of the SOAP protocol decides what is to be put into the header and body parts. SOAP allows great flexibility (basically the whole content of the SOAP message can be customized by means of XML schemas), on the other hand developers can choose from predefined SOAP transaction and encoding formats if they don't want to fiddle with own formats. In the next section we present the most popular predefined SOAP format, the RPC representation with SOAP encoding.

6.2.2 SOAP RPC with SOAP encoding

Remote Procedure Call (RPC) is a very popular network application model. RPC model is the extension of the local function call paradigm when we pass parameters to a function, wait for the execution of the function and receive return parameters. In RPC the function call is done exactly like in the local case, except that the function to be called can be found on a different computer and this other computer can be accessed over the network. The RPC mechanism is responsible of

- encoding the request parameters so that they can be transmitted over the network on the invoker's side
- transmitting the encoded parameters to the receiver over the network
- decoding the request parameters on the receiver's side
- calling the function on the receiver computer
- encoding the return value on the receiver side
- transmitting the encoded return value back to the sender
- decoding the return value on the sender computer
- passing the return value to the caller program

With the appropriate RPC programming interface doing a remote call (over the network) is not much more complicated than doing a local call. An RPC is of course definitely slower than the local call and the network load must also be considered.

RPC is such a popular model that it became the main use case for SOAP. SOAP RPC has its own SOAP representation and is normally transmitted over HTTP that suits naturally to the request-response nature of RPC. HTTP is also able to traverse firewalls so SOAP RPC on top of HTTP can access functions on Internet-based servers as easily as browsers can access web pages.

SOAP RPC adds the following conventions on top of the SOAP core.

- The request SOAP message contains one XML structure that has elements for each input parameter of the RPC function to be called. The elements in the request structure are labeled according to the RPC function parameter names. The number and the type of the parameters is identical to the RPC function's parameter signature.
- The request XML structure is named and typed identically to the RPC function it invokes.
- The response SOAP message contains an XML structure that has elements for each output parameter of the RPC function to be called.

The actual XML structure that carries the request/response is not specified by the SOAP RPC representation. Any XML document instance that conforms to the SOAP RPC Body requirements can be used. The encoding when the programmer defines own XML structures is called *literal* encoding.

SOAP itself defines encodings for frequent data structures like arrays in addition to the XML Schema datatypes. This is called the SOAP encoding and if SOAP encoding is used, the SOAP message is said to follow the *encoded* representation. SOAP encoding is optional but is widely used.

Let's see now a SOAP RPC encoded call that invokes the "add" function with two integer parameters and returns one integer as result! This time I left the HTTP envelope in this example to demonstrate, how SOAP and HTTP works together. The SOAP messages

themselves are in the HTTP request/response content parts. The program that generated the request/response can be found in the example program appendix of this course material.

Here is the request:

```
POST /axis/Calculator.jws HTTP/1.0
Content-Type: text/xml; charset=utf-8
Accept: application/soap+xml, application/dime, multipart/related, text/*
User-Agent: Axis/1.1
Host: localhost: 8080
Content-Length: 475

<?xml version="1.0" encoding="UTF-8"?>
<soapenv: Envelope xmlns:soapenv=http://schemas.xmlsoap.org/soap/envelope/
                   xmlns:xsd=http://www.w3.org/2001/XMLSchema
                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body>
  <ns1:add soapenv:encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
           xmlns:ns1="urn:Calculator">
    <ns1:arg0 xsi:type="xsd:int">3</ns1:arg0>
    <ns1:arg1 xsi:type="xsd:int">4</ns1:arg1>
  </ns1:add>
</soapenv:Body>
</soapenv:Envelope>
```

We see the normal HTTP headers. The message was posted according to SOAP's HTTP binding that is defined to use POST method. The SOAP message itself travels as request content. The SOAP message has just one body part. The body contains the SOAP-encoded RPC invocation content. If the request is SOAP-encoded, it must contain only one child, this is the ns1:add element. The ns1 namespace is an URN specific for our application (urn:Calculator). The add element contains the two RPC arguments (arg0 and arg1), both serialized according to XML Schema.

And now let's see the response:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Date: Fri, 17 Oct 2003 14:31:37 GMT
Server: Apache Coyote/1.0
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv=http://schemas.xmlsoap.org/soap/envelope/
                  xmlns:xsd=http://www.w3.org/2001/XMLSchema
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body>
  <ns1:addResponse soapenv:encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
                   xmlns:ns1="urn:Calculator">
     <ns1:addReturn xsi:type="xsd:int">7</ns1:addReturn>
  </ns1:addResponse>
</soapenv:Body>
</soapenv:Envelope>
```

It should be clear; I would add only one comment. The fact that the response body element is addResponse is totally coincidental. The reason is that we gave this (arbitrary) name. We will see later how we can describe this fact in a format that can be processed easily by programs.

Let's see a bit more complex example (this time without HTTP parts). This time we will transfer an array of integers in the request.

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv=http://schemas.xmlsoap.org/soap/envelope/
                  xmlns:xsd=http://www.w3.org/2001/XMLSchema
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body>
   <ns1:addArray soapenv:encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
                 xmlns:ns1="urn:Calculator">
      <ns1:arg0 xsi:type="soapenc:Array" soapenc:arrayType="xsd:int[5]"
                xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
         <item>1</item>
         <item>2</item>
         <item>3</item>
         <item>4</item>
         <item>5</item>
      </ns1:arg0>
   </ns1:addArray>
</soapenv:Body>
</soapenv:Envelope>
```

Arg0 carries this time an array. SOAP encoding defines the serialization of arrays to XML. Its type is soapenc:Array because the Array type is defined by SOAP encoding, not XML Schema. The <item> tags are part of the SOAP encoding array serialization mechanism. Each array element is carried as a separate <item>. SOAP 1.1 array encoding in addition supports partially transmitted arrays (when just elements in an index range are transmitted) or sparse arrays (where only the elements are transmitted that are different from some default value).

6.2.3 SOAP Document representation

RPC is a useful invocation model but the SOAP world is richer than that. Alternative presentation style to RPC is the document style. The difference between document style and RPC style is that document-style SOAP data is not restricted in the way RPC-style document is restricted. In particular:

- Document-style SOAP message may not be consumed by a single function, therefore the RPC function name is not identified
- Document-style SOAP message may not map to RPC function input/output parameters, therefore there are no restrictions on the format of the XML structure in the Body
- Document-style SOAP messages are not necessarily transferred according to request-response MEP. The coupling of request and response documents may be meaningless for them.

Although presentation style (RPC or document) and encoding (literal or SOAP) are orthogonal to each other (any combination is valid) document-style SOAP messages normally use the literal encoding.

As document-style messages are not tied to the request-response MEP, they can be transferred over asynchronous protocols (like e-mail). In reality, most of the document-style SOAP messages are transferred over HTTP using request-response MEP.

6.3 Web services description language (WSDL)

You may wonder, how do client and servers know about the complex data structures when exchanging SOAP messages. In the previous example you could see that in order to be able to exchange SOAP RPC messages, the client and the server must know

- The name of the RPC function
- The number and type of the input/output parameters
- The namespace where elements needed for the call reside
- Encoding and presentation of the request/response
- Name of the element in the body of the output message

Web Service Description Language (WSDL) is able to describe all this information and more. WSDL is not a protocol, it is a document format that is able to describe web services. Servers publish WSDL documents (we will see at UDDI how). Clients download these documents and use them when they generate their requests to the server or when they process server responses. Theoretically WSDL is able to describe any messaging protocol but it is almost always used with SOAP and mostly In the request-response MEP.

Currently the WSDL 1.0 is widely deployed; this section is based on the WSDL 1.0 version.

Let's see a simple WSDL document that describes our simple adder service!

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="urn:Calculator"
                          xmlns:impl="urn:Calculator"
                          xmlns:intf="urn:Calculator"
                          xmlns:wsdlsoap=http://schemas.xmlsoap.org/wsdl/soap/
                          xmlns:soapenc=http://schemas.xmlsoap.org/soap/encoding/
                          xmlns:xsd=http://www.w3.org/2001/XMLSchema
                          xmlns:wsdl=http://schemas.xmlsoap.org/wsdl/
                          xmlns="http://schemas.xmlsoap.org/wsdl/">
<wsdl:message name="addResponse">
   <wsdl:part name="addReturn" type="xsd:int"/>
</wsdl:message>

<wsdl:message name="addRequest">
   <wsdl:part name="in0" type="xsd:int"/>
   <wsdl:part name="in1" type="xsd:int"/>
</wsdl:message>

<wsdl:portType name="MiniCalculator">
   <wsdl:operation name="add" parameterOrder="in0 in1">
      <wsdl:input name="addRequest" message="impl:addRequest"/>
      <wsdl:output name="addResponse" message="impl:addResponse"/>
    </wsdl:operation>
</wsdl:portType>


<wsdl:binding name="CalculatorSoapBinding" type="impl:MiniCalculator">
  <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
     <wsdl:operation name="add">
       <wsdlsoap:operation soapAction=""/>
       <wsdl:input name="addRequest">
         <wsdlsoap:body use="encoded"
           encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
```

```
            namespace="urn:Calculator"/>
        </wsdl:input>
        <wsdl:output name="addResponse">
          <wsdlsoap:body use="encoded"
                  encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
                  namespace="urn:Calculator"/>
        </wsdl:output>
      </wsdl:operation>
</wsdl:binding>

<wsdl:service name="MiniCalculatorService">
  <wsdl:port name="Calculator" binding="impl:CalculatorSoapBinding">
    <wsdlsoap:address location="http://localhost:8080/axis/services/Calculator"/>
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

The WSDL document is divided into the following sections.

- “Types” sections define the XML Schema types used in the WSDL document. This time we had no such section.
- “message” sections describe input and output messages exchanged between the client and the server. This time we had two messages: addRequest carries two integers (input parameters) while addResponse returns one integer. Due to the constraints of RPC, the addRequest data structure is not sent, only its children nodes (the two elements described by wsdl:parts). These children nodes will be embedded into an add element (having the same name as the RPC function).
- “portType” sections describe logical operations. A portType define an operation (with operation name for RPC) and input/output messages. This time we have only one operation (add) that receives addRequest requests and produces addResponse responses.
- “binding” sections connect operations defined in the “portType” sections to transport-specific information. In our case the binding section defines that we do SOAP RPC and that the request/response is encoded with SOAP encoding.
- “service” section connects the binding with the service parameters like service name and service address.

Let’s see now a bit more complex case where we have to define our own types! We will describe the array adder service in the previous section (only the differences are shown here, you can find the entire calculator-rpc.wsdl file in the example code appendix).

We have a types section:

```
<wsdl:types>
  <schema xmlns=http://www.w3.org/2001/XMLSchema
      targetNamespace="urn:Calculator">
    <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
    <complexType name="ArrayOf_xsd_int">
       <complexContent>
          <restriction base="soapenc:Array">
             <attribute ref="soapenc:arrayType" wsdl:arrayType="xsd:int[]"/>
          </restriction>
       </complexContent>
    </complexType>
```

```
  </schema>
</wsdl:types>
```

This (quite complex) schema declaration references the integer array encoding from the SOAP encoding. This schema is then referenced in a message declaration:

```
<wsdl:message name="addArrayRequest">
    <wsdl:part name="in0" type="impl:ArrayOf_xsd_int"/>
</wsdl:message>
```

WSDL is definitely unfinished and doesn't satisfy the latest developments in the web services field. For example WSDL's support of MEPs is quite poor, the relevant part of the standard is missing. In practice, WSDL can be used only with request-response MEP.

6.4 Universal Description Discovery & Integration (UDDI)

Web services are meaningful only if potential users may find information sufficient to permit their execution. The focus of Universal Description Discovery & Integration (UDDI) is the definition of a set of services supporting the description and discovery of businesses, organizations, and other Web services providers, the Web services they make available, and the technical interfaces that may be used to access those services.

The UDDI specification is under the control of OASIS (Organization for the Advancement of Structured Information Standards, http://www.oasis-open.org) standardization body. This course material refers to 3rd version of the specification.

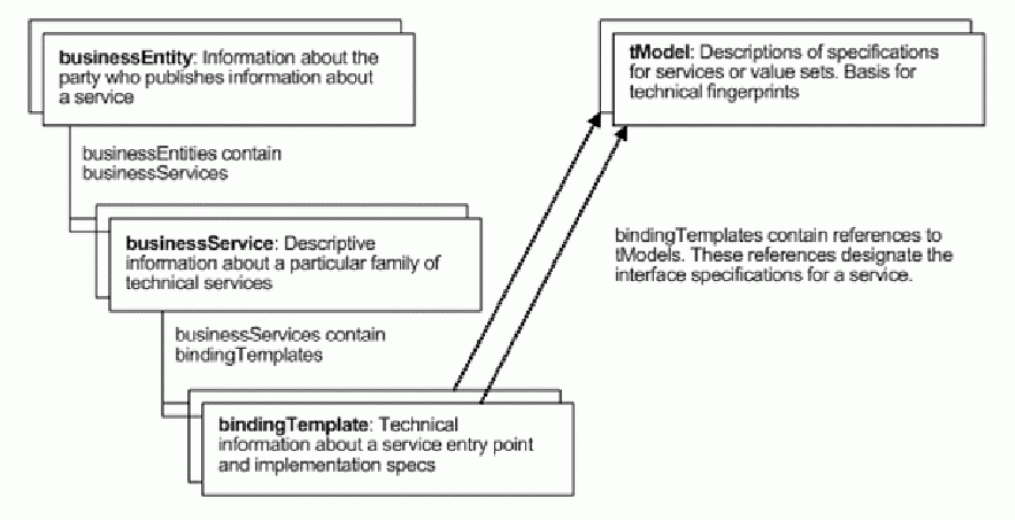
Key element of the UDDI is the registry. The registry is a large database whose address is well known by the clients. When clients want to access a web service whose details are not known to the client, it performs a search in the registry and retrieves the necessary information to contact the web service over SOAP.

UDDI specification has two main parts: registry data model and access APIs. The data model is used to describe business information and technical details needed to access the service. The access APIs provide functions on the data model.

6.4.1 UDDI data model

The UDDI data model is based on XML documents defined by XML schemas. The documents can exist individually inside the registry or can be embedded into higher-level documents. Documents always have a key by which they can be referenced. For individual documents, the publisher of the document sets the key, for embedded documents the key is generated by the registry and is often the same as the key of the higher-level document.

The main UDDI document types are the following:



- BusinessEntity is normally the highest-level document. A businessEntity describes the business in terms of address, contact and business area. International identifier systems are used for the business area, like the D-U-N-S numbers provided by Dun & Bradstreet or the Global Location Number system numbers defined in the EAN UCC.

- Embedded or referenced by the businessEntity document are the businessService documents. Each businessService describes a particular logical service offered by the business entity. One businessService may contain multiple web services.

- The individual web services comprising a logical businessService are described by bindingTemplate documents. BindingTemplate contains technical information how the service can be accessed. Most important is the URL of the WSDL document of the web service.

- Optional tModel documents may be used to describe the technical fingerprint of web service. The technical fingeprint may contain information on the transport to access the web service, the version of SOAP and WSDL the web service uses and other technical details. This information can be used to select the web service appropriate for the client. TModel documents may or may not be stored in the registry. If they are not stored in the registry, they are accessible by HTTP GET and the registry stores only the URL.

6.4.2 UDDI API functions

UDDI API functions can be accessed over the network as SOAP services, UDDI is therefore an application on top of SOAP. UDDI uses document-style presentation with literal encoding, in practice it means that UDDI client and server exchange XML documents in SOAP envelopes, in the Body part. UDDI specifies several transports (HTTP, e-mail, FTP) but it is always used in request-response MEP. The most frequent deployment uses HTTP as transport.

The API functions can be divided into the following groups:

- Inquiry API can be used to find entities in the registry and query the data associated with the entity.

- Publication API can be used to manipulate the data in the registry, add and delete entities and create connections among entities.
- Security Policy API can be used to obtain and discard authentication tokens. Authentication tokens are data structures that may be required by other functions if the registry policy prescribes authentication to access certain functionality.
- Custody and Ownership Transfer API allows transferring a certain entity from the control of one publisher to another. The custody transfer may happen inside the same registry or between two registries.
- Subscription API allows UDDI clients to subscribe for registry change notifications. The client defines data items and when these items are changed, the registry sends notification to the client.
- Value Set API allows external validation services to be connected to the registry. A validation service is attached to data type. When the registry saves that data type, it invokes the validation service over SOAP to check whether the data item is valid.

In addition to the API functions available for the client, a Replication API is available for inter-registry communication.

To demonstrate the APIs, let's see an example from the Inquiry API! This request document is embedded into the request SOAP Body and the response Body contains the selected businessEntity document.

```
<find_business xmlns="urn:uddi-org:api" generic="1.0" maxRows="100">
  <findQualifiers>
    <findQualifier>
      uddi:uddi.org:findQualifier:approximateMatch
    </findQualifier>
  </findQualifiers>
  <name>SUN</name>
</find_business>
```

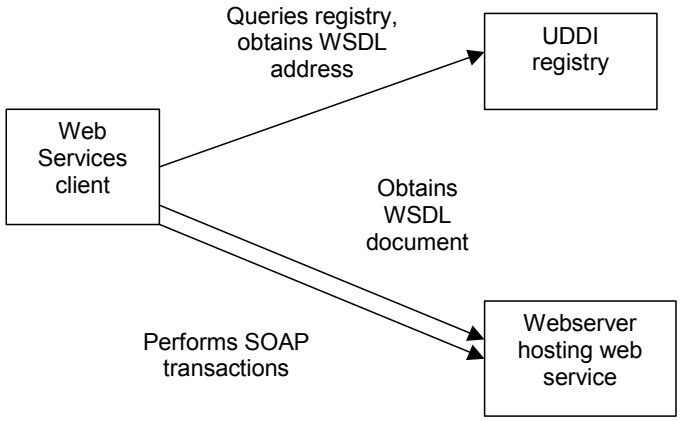
And the response:

```
<businessList generic="1.0" operator="Microsoft Corporation" truncated="false"
                       xmlns="urn:uddi-org:api">
<businessInfos>
  <businessInfo businessKey="f293ee60-8285-11d5-a3da-002035229c64">
  <name>Sun Microsystems</name>
  <description xml:lang="en">Leading provider of industrial-strength hardware,
software  and services that power the Net</description>
   <serviceInfos>
     <serviceInfo serviceKey="02d4b1a0-8291-11d5-a3da-002035229c64"
      businessKey="f293ee60-8285-11d5-a3da-002035229c64">
       <name>Products &amp;amp; Solutions</name>
     </serviceInfo>
    <serviceInfo serviceKey="1c65cc80-8291-11d5-a3da-002035229c64"
                businessKey="f293ee60-8285-11d5-a3da-002035229c64">
     <name>Developer Connection</name>
    </serviceInfo>
   </serviceInfos>
  </businessInfo>
</businessInfos>
</businessList>
```

Several high-level APIs are available (for example Java API for XML Registries (JAXR)) for Java that allow more convenient access of UDDI services than the SOAP-based API.

6.4.3 Using UDDI registries in web services architectures.

In the web services architecture, UDDI is used to discover web services dynamically. The setup is shown in the following picture.



The client first locates the web service by querying the registry with web service provider business function, web service technology fingerprint (like SOAP version), etc. The query can take several steps, for example first the businessEntities can be located then the client can drill down through businessServices and bindingTemplates until an appropriate service is found. Then the web services client downloads the WSDL document assigned to the web service(s) with a normal HTTP transaction. The WSDL document doesn't have to be on the same webserver where the SOAP interface is located although it is a quite practical setup.

By analyzing the WSDL document, the client is able to connect to the SOAP interface generating appropriate requests to the server and analyzing the server responses properly.

This "architecture vision" is very poorly deployed. First of all, existing SOAP implementations are not able to process WSDL documents directly. WSDL documents are first preprocessed by some system tool that generates plugin code for the SOAP engine. Existing SOAP deployments are therefore very static, client programs actually have the structure and the address of their servers compiled into them. In this environment, dynamic query of UDDI registries doesn't add too much value.

Time will tell whether the "web services revolution" will be widely deployed in all sorts of computer systems. The web services architecture is a very important lesson for us, however, because it is rare to see distributed application architecture so clearly. In the next chapter we will see a much more complicated system.

7. CORBA

7.1 Object Request Brokers

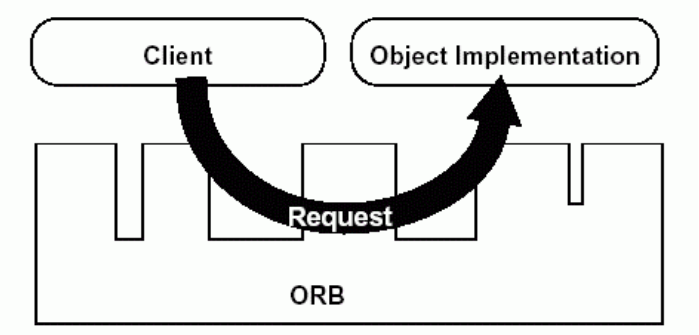
The Common Object Request Broker Architecture (CORBA) originates from 1995. Presently the specification is at the 3rd version and is maintained by the Object Management Group (OMG, http://www.omg.org).

CORBA is based on communication of *objects*. Object is a well-known term in software engineering. Object is an encapsulation of variables and functions (called *methods* in object-oriented terminology) that operate on these variables. First we define the structure of the object by specifying the variables and the methods. This structure is called the object's *class.* The class can be thought of as the type of the objects that are generated from this class. Then the object is *instantiated,* once or many times as needed. Each object instance has its own set of variables defined in the class (called *instance variables*). When we call a method on a particular object, the method will use the instance variables of that particular object instance.

An object-oriented program looks completely identical from the operating system point of view to any other program; the object-based operation is internal to the program. CORBA extends object-oriented operation to separate programs. These programs can run on the same computer or can run on different computers connected by network. CORBA's promise is that a program written in CORBA model is independent on the actual location of these objects, CORBA objects can be accessed from any network-connected computer and the system can be distributed to multiple computers without changing a single line of code in the program.

It is very important to note the difference between a CORBA object and an ordinary object in an object-oriented program. A CORBA object is under the control of the ORB. It is created, invoked and destroyed by the ORB. An ordinary object is under the control of the program that uses it, there are no intermediaries. A program written in CORBA model exposes some of its objects (or structures equivalent to objects if the host language is not object-oriented) in order to realize a distributed architecture. It would be a very bad idea (and it would result in a very slow system) if all the objects in the program were accessible through the ORB.

Key element in the CORBA architecture is the Object Request Broker (ORB). CORBA objects are under the control of ORBs. Objects are created, accessed and destroyed through the ORB that covers the particular implementation details of the objects.



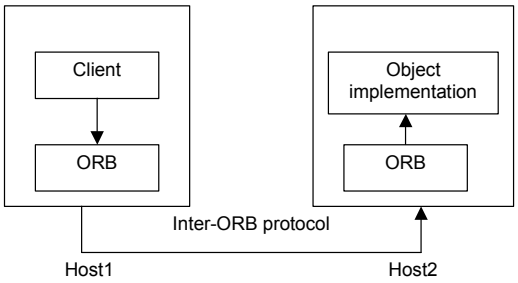
It is invisible for the client or the object's implementation how the request travels from the client to the server; they are only connected to the ORB. This architecture allows the following important features.

- As the client and the object are not connected directly, only through the ORB, they can be implemented in different programming languages and/or different object mapping. For example it is possible to access an object implementation written in C from Java. The interesting fact is that C is not an object-oriented language, CORBA specifies language mapping for C so that CORBA objects can be expressed in C. The Java client which is written in a perfect object-oriented language will not notice

that the object implementation it calls is written in a different language which is in fact not even object-oriented.

- The client and the object implementation may be on the same computer but may be on different computers too. In this case the client and the object implementation are connected to different ORBs. The two ORBs are connected through the network and the call is propagated from one ORB to the other by means of a network protocol.

- The client's ORB can also do clever object location operations and the client doesn't have to know about it. For example the target ORB may inform the client's ORB that the object implementation is now located on a different ORB (because of overload or redundancy). The client request will be forwarded to that second ORB and the client will see only that it was able to access the object implementation.



ORBs are normally full CORBA implementations. It means that they are able to act as clients and object containers at the same time. An ORB that invoked a method on another ORB may be target of a method invocation at the same time. The distinction between the client and target ORB is therefore artificial and is valid only in one particular method invocation context.

The CORBA specification suite is composed of the following specifications:

- ORB specification, how ORBs work and how they interact with clients and object implementations

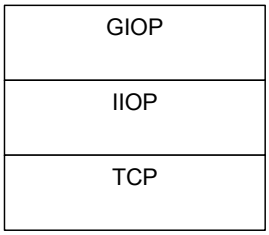
- Inter-ORB protocol so that ORBs from different vendors running on different platforms can interoperate

- Interface Definition Language that is able to describe object interfaces in a way that it doesn't depend on the object implementation's language

- Several standard CORBA services so that programmers can have a standard component library for the most frequent tasks.

7.2 GIOP and IIOP

CORBA defines standard Inter-ORB protocol so that ORBs from different vendors running on different computers can interoperate. The protocol has two layers. The transport-independent message layer is called General Inter-ORB protocol (GIOP). This message layer is mapped to different transports. If GIOP is mapped to TCP, the standard mapping is called Internet Inter-ORB protocol (IIOP). GIOP-IIOP combines presentation and session-level features.



There are only 7 GIOP messages. These are:

- **Request:** CORBA object invocation request
- **Reply:** CORBA object invocation response
- **CancelRequest:** Cancellation of a previous Request message. It may be time-consuming to produce the Reply and this process is stopped.
- **LocateRequest:** finds out if requests of a certain object reference are handled by the target ORB or some other ORB. Request message can also be used for this purpose but LocateRequest doesn't carry (potentially lengthy) invocation parameters.
- **LocateReply:** Response to LocateRequest.
- **CloseConnection:** sent when either party wants to close the connection. The other side is also informed by this message that all the pending requests on this connection are cancelled.
- **MessageError:** The receiving party notifies the sender that it found some message that doesn't conform to the GIOP specification (for example the header is incorrect or uses a higher GIOP version than this ORB is able to handle).

All messages carry a message identifier that uniquely identifies request-response message pairs. This allows GIOP to send responses to requests in any order, as soon as the processing finishes. For example the following sequence is valid.

1. Request 1 on obj1::method1
2. Request 2 on obj1::method1
3. Response 2 on obj1::method1
4. Response 1 on obj1::method1

Although request 1 arrived earlier, for some reason its processing took longer time (for example the parameters of the requests triggered more complicated processing). Meanwhile, a new request was served (request2-response2). As response messages carry the identifier of the request message, this doesn't cause confusion, the client can sort out what response it receives.

Starting with GIOP 1.2, the communication between the client and the server is bi-directional. This means that after the connection was established, both the client and the

server can send requests and receive responses. Before GIOP 1.2 only the client could send requests and the server could only respond.

Let's see what is the content of a Request-Response message pair!

Every GIOP message starts with a header block. The GIOP header starts with the "GIOP" identification string, GIOP version information, byte ordering information, GIOP message type and the length of the GIOP message.

The Request message starts with GIOP header then the Request header block follows. The Request header looks like the following.

- Request ID is a unique, 32 bit number that identifies the request. The response carries this number so that the client can pair requests and responses.
- Response flags is an 8-bit value. In current implementations the flag denotes when the client program gets back the control. Possibilties: SYNC_NONE (after the ORB received the request), SYNC_WITH_TRANSPORT (after the message was accepted by the transport protocol layer), SYNC_WITH_SERVER (after the target ORB received the message), SYNC_WITH_TARGET (after the target object's response was received). This enables one-way operations (with no response).
- Target is the identifier of the target object. Originally CORBA treated object references as opaque values that only the ORB handling the object can understand.them. In order to guarantee interoperation of different ORBs, CORBA defined the Interoperable Object Reference (IOR) data structure that carries information about the object's type, whether the object reference is null, the transports that the object supports and so on. The target field of the header block is able to carry target object identifiers in different formats.
- Operation is the name of the object's method that is being invoked
- Service context list may carry service-specific information. For example one such service context is the bi-directional service context; if the communicating ORBs support bi-directional transport, the client sends the bi-directional service context to the server. The server may use this context to find out, what requests it can send in the other direction (in this case a CORBA object is accessed on the original client). The context in this case carries information what client ports can be accessed on the connection that was originally opened by the client. Several other service contexts are defined in CORBA.

Following the header block, the object method's input parameters follow. The input parameters are serialized according to the CDR (Common Data Representation) notation. CDR parameter streams simply put the parameters after each other in binary form without any additional field separator. Alignment is specified for primitive types. For example unsigned long is a 4-byte type (32 bits) and its alignment is 4 byte. Char type has alignment of 1 byte. Each primitive type must be placed in the byte stream at integer multiple of its alignment, for example the unsigned long type must always start at *k**4 offset where *k* is an integer number. If char is followed by an unsigned long, the byte stream looks like the following:

<char value: 1 byte> <padding: 3 bytes> <unsigned long value: 4 bytes>

Padding was used to guarantee that the unsigned long value starts at 4-byte boundary.

The CORBA parameter block doesn't contain type information and can be interpreted only with the help of object interface description that is not transmitted with the message. This is a basic difference between CORBA and Web Services: a Web Services message always carries typed data while CORBA messages are opaque byte streams that don't carry type information. This makes CORBA messages more compact and faster to parse but the target of the CORBA message must be known exactly: a CORBA message is created exactly for the receiver object method. Speaking Web Services terms, CORBA knows only the RPC mechanism and document-type messages are not supported explicitly by CORBA.

The Reply message is similar to the Request message. After the mandatory GIOP header comes the Reply header block.

- Request ID: the Reply carries the ID of the request it responds to
- Reply status: informs the client on the status of the request. The status can be successful completion, exception (from the ORB or from the called method) or forward message if the target object is located at another ORB and the client must resend the request to the other ORB.
- Service context list similar to the Request message

The Reply header is followed by the CDR representation of the method's output parameters.

If GIOP is sent over TCP, the TCP usage is determined by the IIOP protocol specification. IORs used with IIOP contain the target ORB's address and port. If the client wants to access the target ORB over IIOP, it takes the target ORB's address and port from the IOR and creates a TCP connection to the target ORB. GIOP messages are then sent over this TCP connection until any party issues CloseConnection GIOP message and closes the TCP stream.

7.3 The IDL interface specification language

As we have seen, CORBA is able to connect object implementations written in different programming languages. CORBA messages, however, don't contain type information. It is extremely important therefore to have an interface definition language that can be mapped to any programming language. In CORBA, this language is the Interface Definition Language (IDL).

IDL is not a programming language. Its sole purpose is to define object interfaces. IDL descriptions are compiled to the object implementation's host language (to C for example if the CORBA object is implemented in C).

The IDL file contains interface declarations and type declarations that are necessary for the interfaces. Interfaces contain method and member variable declarations, just like class declarations in ordinary programming languages. Example:

```
typedef sequence<long> LongVect;

interface AdderInterface {
  long add( in long n1, in long n2 );
  long addArray( in LongVect nums );
  string reverse( in string str );
};
```

(Java-based implementation of this interface can be found in the programming appendix)

IDL uses its own types that are mapped to the object implementation host language's own types. For example *long* in IDL denotes a 32-bit signed value. When mapped to Java, it becomes Java's *int* type. Similarly, the LongVect type declaration that defines LongVect as sequence of *long* values will be mapped to an equivalent Java data type, array of *int* values (int[]).

IDL mandates that all parameters have direction specifier. A parameter can be

- **in –** this means that the parameter is consumed by the method call. This is equivalent to call by value.
- **inout –** this means that the parameter is consumed by the method call but the method manipulates the parameter and the resulting value is returned to the caller. This is equivalent to call by reference.
- **out –** this means that the parameter is an output parameter. In CORBA a method may have multiple output parameters. This feature is often mapped to host languages as call by reference.

IDL interfaces can use *inheritance*. This means that an interface can inherit methods and fields from another interface and can redeclare methods and fields that are changed compared to the parent interface. Multiple inheritance is also supported, an interface can inherit from more than one parent.

```
interface A { ... }
interface B: A { ... }
interface C: A { ... }
interface D: B, C { ... }
```

Interfaces of CORBA objects connected to a particular ORB are placed into the interface repository of the ORB. When a method is invoked, the ORB will use its interface repository to find the matching interface. Normally IDL files are not used directly by the ORB, a tool belonging to the ORB transforms the IDL files into some ORB-specific format and the interface repository stores files in this ORB-specific format. For example IDL files are often translated into modules of the host language and the interface repository stores the components compiled from these modules.

7.4 The CosNaming name service

A CORBA object is unambiguously identified by its IOR. The IOR tells us the target ORB where we have to connect and identifies the object inside that ORB. There is one problem with IORs: they are not to be used by humans because they are long and change often. You can find an example program client-server pair in the programming appendix (AdderClientIOR and AdderServerIOR) that communicate by means of IOR. The server saves the IOR into a file and the client reads the IOR from the file. No further help is needed. You can examine how an IOR looks like, check out the text file called AdderIOR.

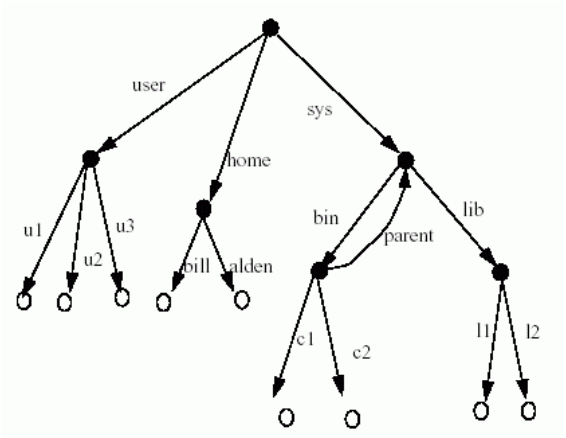
This kind of communication was already problematic with Web Services (what happens if the server's name changes?) but it causes intractable problems with CORBA. Each time the server ORB is restarted, a new IOR is created. It is not enough that IORs are hard to communicate, they also change often.

The naming service is therefore a much more integral part of CORBA than UDDI was in case of Web Services. The basic CORBA name service is called CosNaming. CosNaming is just like any other CORBA object but there is a trick: if CosNaming were just an ordinary CORBA object, you wouldn't be able to use it because you would need the naming service in order to access the naming service. For this reason ORBs use bootstrap process to obtain the CosNaming object, by the time you want to resolve names, the CosNaming object is already available. Getting the CosNaming IOR is therefore not exactly the same as getting any other object's IOR.

CosNaming stores object references (most often IORs) in a hierarchical database. In this database objects are assigned to names. A name-to-object association is called a name binding. A name binding is always defined relative to a naming context. A naming context is an object that contains a set of name bindings in which each name is unique. Different names can be bound to an object in the same or different contexts at the same time. There is no requirement, however, that all objects must be named.

To resolve a name is to determine the object associated with the name in a given context. To bind a name is to create a name binding in a given context. A name is always resolved relative to a context — there are no absolute names. Because a context is like any other object, it can also be bound to a name in a naming context. Binding contexts in other contexts creates a naming graph — a directed graph with nodes and labeled edges where the nodes are contexts. A naming graph allows more complex names to reference an object. Given a context in a naming graph, a sequence of names can reference an object. This sequence of names (called a compound name) defines a path in the naming graph to navigate the resolution process.

Let's see an example naming graph!



CosNaming object provides the following services: binding objects to names, resolving names to object references, unbinding names, creating, deleting and listing of contexts.

8. CLOSING WORDS

I wanted to introduce the whole breadth of communication methods used in today's network-based applications. Today's network technology is dominated by IP-based solutions. There exist only very few exotic systems (mostly in the telecommunication and field network area) where IP is not available. In this course material you were introduced to higher and higher-level services built on the IP layer. We have seen at least one widely deployed network service on every level of the OSI hierarchy starting from IP. As the time

was short, it was not possible to introduce every aspect of complex systems like CORBA but hopefully this course gives you a good overview.

The most important lesson that I would like you to recognize is that you have to be able to use your knowledge and select the appropriate network access solution for your application. Each of the methods we have seen has merits and drawbacks. Select the appropriate method for your application. I present below a short (and rather inexact) guidelines for selection.

- Use IP directly only if you write some application that works directly with the IP backbone.
- Use datagram-oriented protocols (like UDP) only if your network communication has real-time constraints (like you can tolerate only a certain packet delay on streaming audio)
- Use TCP directly if your application protocol is simple and your client and server programs have memory footprint restrictions.
- Deploy HTTP whenever you can; not because HTTP is a particularly good protocol but because the HTTP architecture is so widely available.
- Be careful with security solutions like TLS/SSL. TLS/SSL is a very secure protocol but also requires significant resources (both in program size and CPU load). Also, you have to be ready to deploy public-key infrastructure elements like private keys and certificate stores. Can't you satisfy your security requirements on application level?
- If your program size footprint allows, use some good representation layer on top of HTTP because you can suffer a lot if you want to encode-decode your complex data structures by hand.
- Use web services if you have enough network bandwidth and CPU time to generate, transmit and parse complex and relatively large XML data structures. At the time of writing this course material, web services are mainly used to integrate systems with different architectures.
- If you develop a networked application where you don't have exotic requirements, choose a middleware solution that hides the complexity of network communication. CORBA may be a good solution for more complex systems; small-footprint middleware solutions (like Java RMI) exist for simpler systems.