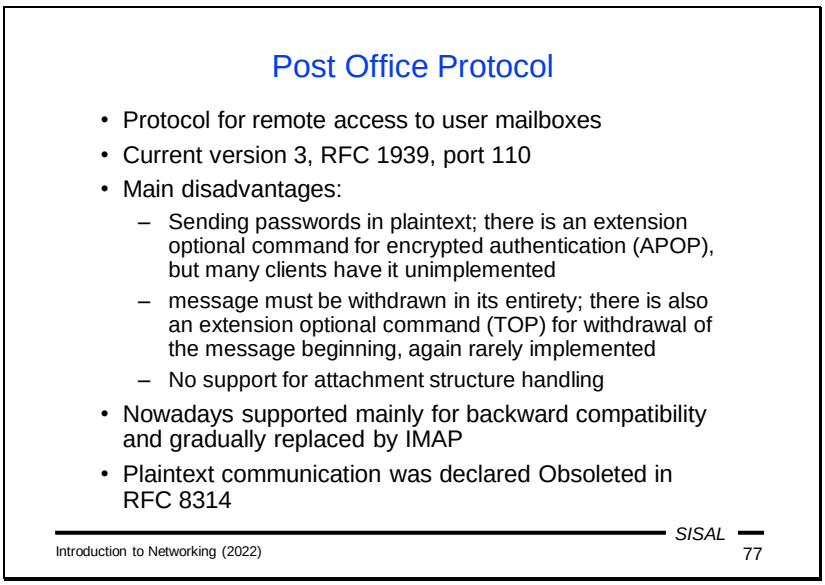
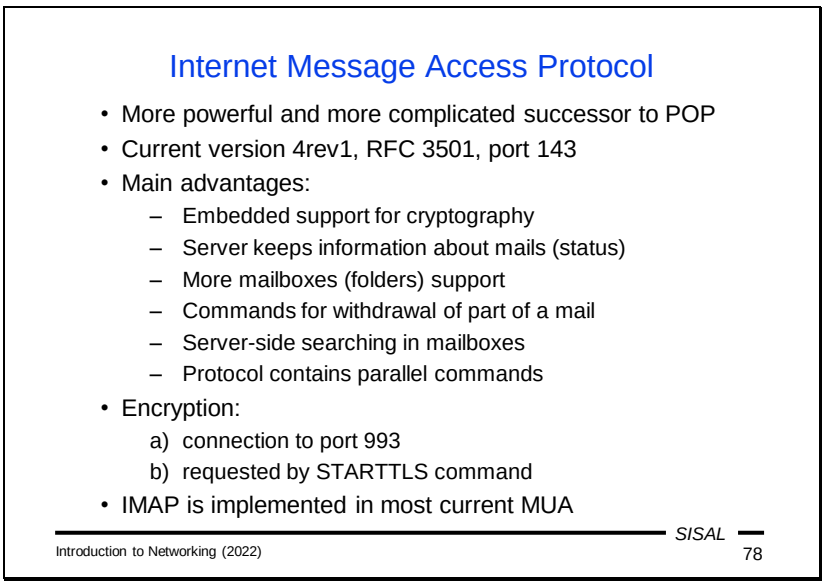


Users can use either the POP or IMAP protocol to remotely access messages in their mailbox.

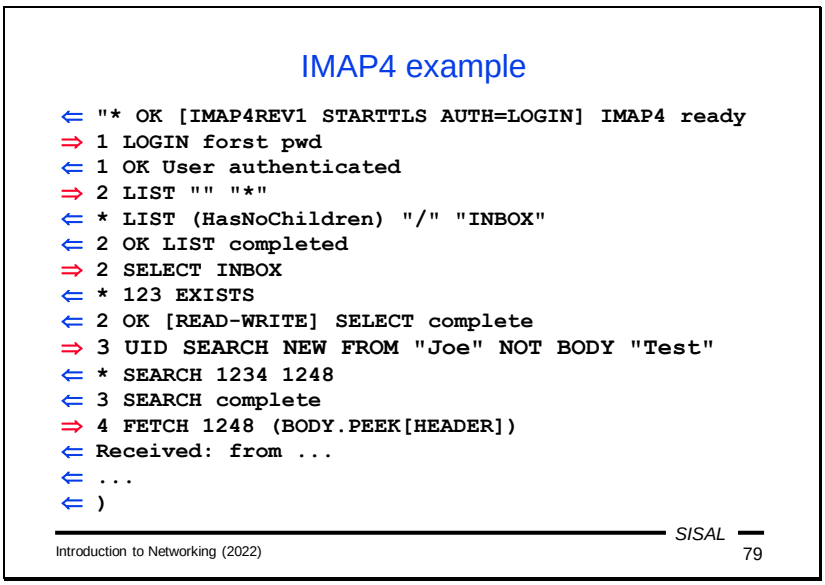
POP is the older of the two protocols and contains many known security concerns. Over time, there have been extensions that have tried to eliminate them, but in the meantime IMAP has gained more popularity, so implementing these extensions isn't common. Today, therefore, TLS security is more likely to be used, and in 2018 plaintext communication in the protocol was even declared Obsoleted.

Interestingly, the first two versions of POP used the *push* principle, that is, the server initiated the transfer of messages to the client. The most recent version now uses the *pull* principle. However, the main drawback of the protocol remains its limited capabilities for working with a mailbox.



Although the first version of the IMAP protocol appeared only two years after POP1, its author put more emphasis on working with a remote mailbox from the beginning, and IMAP2 included e.g. commands to search for a message in a mailbox according to several criteria, or to download only certain information about a message. The development of version 4 has been taken over by an IETF working group, and this version includes a number of security and functional extensions.

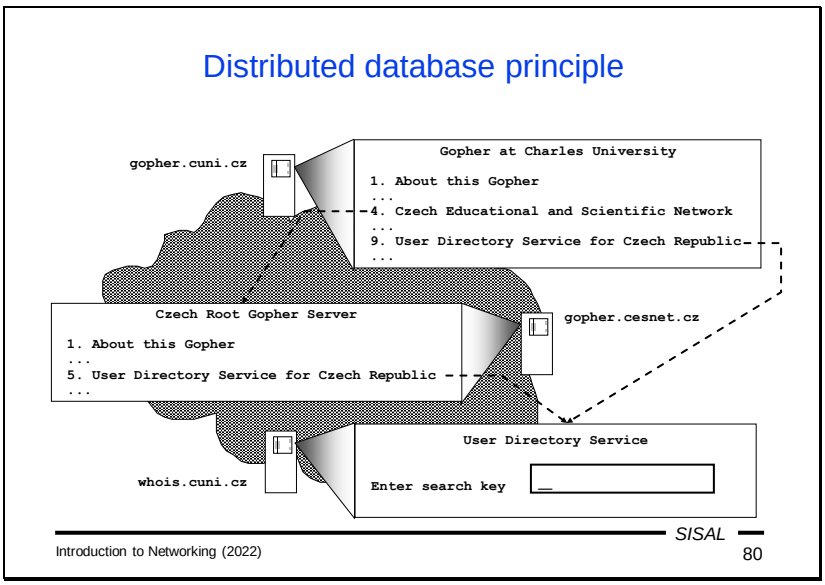
This version is now widely supported by mail applications and, in accordance with current standards, is used in combination with TLS, either on the dedicated port 993 instead of 143, or on demand by a STARTTLS command issued by the client.



The protocol is also text-based, but compared to POP, it is much more user-friendly. The client can use commands to work with various folders and subfolders, can search for letters according to complex conditions, download only parts of letters etc.

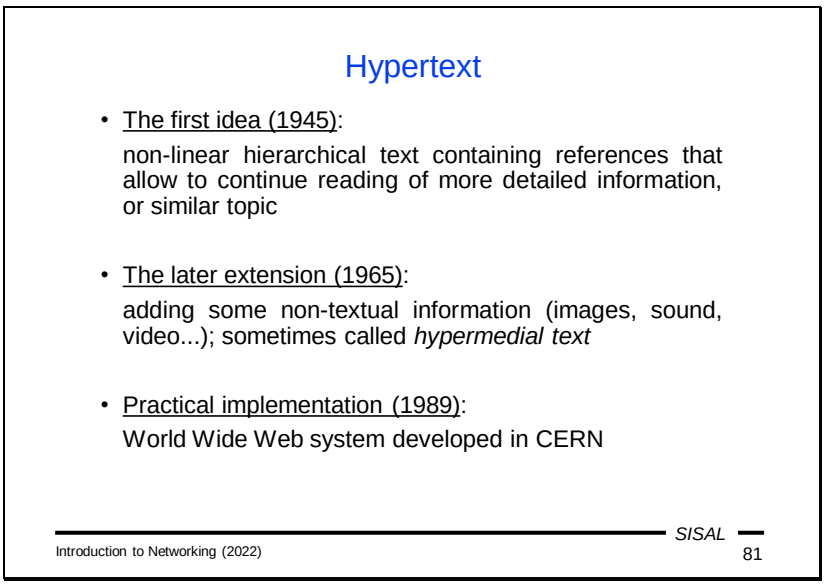
An interesting feature is also the tagging of commands and responses, which makes it easier to assign responses to sent commands. Thus, the client can send multiple commands to the server in parallel. The absence of this mechanism complicates e.g. in FTP the implementation of some operations, such as aborting a file transfer.

Slide 80



In 1991, the Gopher system appeared, the world's first distributed database service, a database of information that was stored on a huge number of servers and interconnected so that a user could navigate from one server to another without necessarily realizing it. At the time, Gopher was an information retrieval service with some of the same functionality as the web today. Even then, for example, all universities offered access through Gopher to study information, telephone directories, etc. When user connected to a Gopher server, a menu was displayed. After the user selected a menu item, another page appeared containing another menu, text with information the user requested, or a form which the user could fill out and submit to see some dynamically generated information. Links between individual pages could lead to a completely different server. As you can see, working with Gopher was similar to how users work with the web today. The only difference was that Gopher provided only textual information. If a link led to an image, Gopher downloaded it, but the user had to view it in another program.

Slide 81



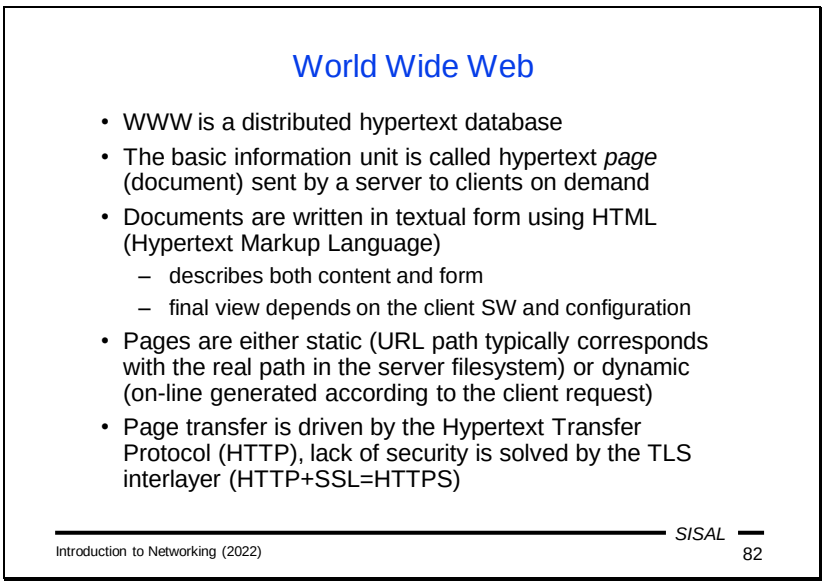
What Gopher lacked was hypertext.

The idea of hypertext is surprisingly very old. As early as the middle of the last century, at a time when printed materials were the only source of information, there were ideas that a text might contain links to explanatory or related texts. This would be somehow similar to an index, but available directly from the text, not as a summary at the end of the book. However, implementation at that time was of course not possible.

Twenty years later, the basic idea expanded to include the possibility of directly including non-text elements in the text, and an alternative name, *hypermedia* text, appeared. However, the idea still had to wait for the technology that could implement it.

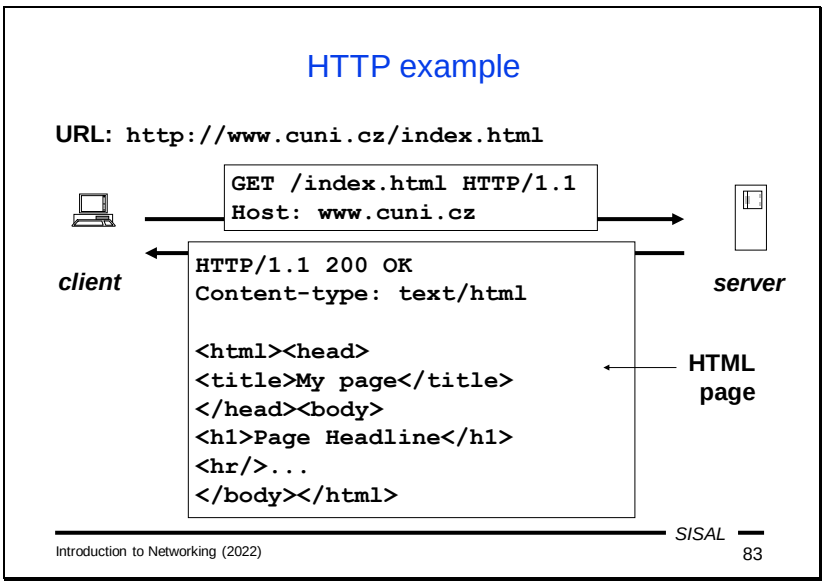
After another twenty years, more precisely in 1989, the idea of a system for disseminating information of various kinds was born at the Swiss center CERN using a service later called the World Wide Web.

Slide 82



Simply said, the web is a distributed hypertext database. As with Gopher, global information is distributed on a huge number of servers around the world and is interconnected so that you can browse between them without users having to worry about which server they are connected to.

The basic unit of information is a *page*. This term corresponds to what existed in Gopher before the web, but its meaning is quite different. A web page is sometimes referred to as a *document*, which better corresponds to the fact that a web page has nothing to do with a page in some printed material. A web page can be stored as a file on a server (then we call it *static*), or it can be generated *dynamically* by a server based on a parameterized user request. In both cases, the client sends a request to the server and the server responds with the text of the requested page (or an error message). Pages are usually written in HTML (Hypertext Markup Language), which contains means for expressing both content (i.e. text, embedded images, links, etc.) and form (font, color, tables, alignment, etc.). HTML describes an author's idea of how a page should be displayed on the client's screen, but the exact appearance of the displayed page is determined by many other factors (implementation of the language's properties in the browser, user settings, etc.).



The Hypertext Transport Protocol (HTTP) is used to transfer web pages. In its basic version (1.1) it is a text protocol in which

- the client establishes a connection to a server and sends a text request containing a URL (written partly on the first line which contains a command and partly in the headers that follow it) and some other parameters
- the server responds with a line containing a response code (200 in our example), headers containing (among other things) the MIME type of the requested document, and the document itself (in our case an HTML page). In case of an error, the response will contain a different completion code and the HTML text of an error page.

Hypertext Transfer Protocol v.1

- Currently still mostly version 1.1, RFC 7230, port 80
- General message form:
  - the first line (request/response)
  - additional header lines
    - request: language, charset, page age, authentication,...
    - response: document type, encoding, expiration,...
  - (optional) document body
- Status (response) codes:
  1xx **informational** (provisional response, processing continues)
  2xx **success** (final response)
  3xx **redirection** (some additional client request expected)
  4xx **client error** (incorrect request)
  5xx **server error** (transient or permanent error on server)

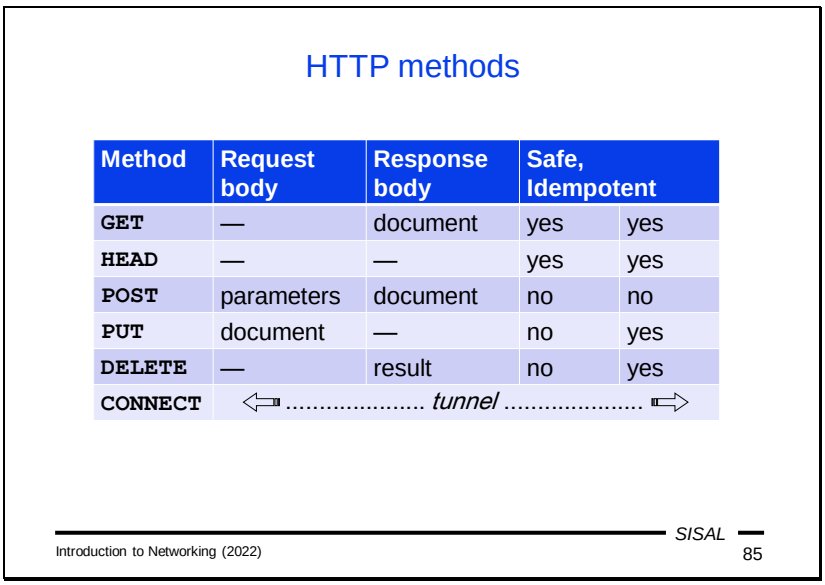
Currently, version 1.1 of the protocol is most widely used. This version is text-based; the request and response contain an initial line, usually followed by a series of header lines (with basically the same format as mail headers), a blank line, and in some situations can be followed by the body of a document.

In the request, the initial line consists of a method name (e.g. GET for requesting a page download), a path (i.e. the part of the URL after the server name) and the protocol version. Of the headers, in version 1.1 only the Host header is mandatory, which specifies which server the client is connecting to. This is because multiple servers can run at the same IP address, and it is this header that distinguishes which server a request is addressing. Other useful headers include the language and encodings that the client accepts, the maximum allowed page age, authentication data, etc. The body will not be present in a normal request, with the exception of special types of requests where either additional parameters are sent to the server or an entire document is uploaded.

In the server response, the first line is a status line, again containing the protocol number, a three-digit response code, and a verbal description of the response. Headers relate to formalities (such as the time of the last change or the expiration time of a page), details of the protocol (such as the method of transmission), and the properties of the document being sent (such as the MIME header). The body of the response contains the required document or the text of an error message.

The response codes basically correspond to what we know from previous protocols. 2xx is a final positive reply, 1xx and 3xx mean that the request is more or less OK; 1xx means that the ball is in the server's court (another answer is coming in a while), while 3xx plays the ball onto the client side (the client must enter another URL, if it

wants to get the page content). Only error response codes have semantics that slightly changed from other protocols; 4xx means an error on the client side (bad URL, authentication, etc.), while 5xx means an error on the server side.



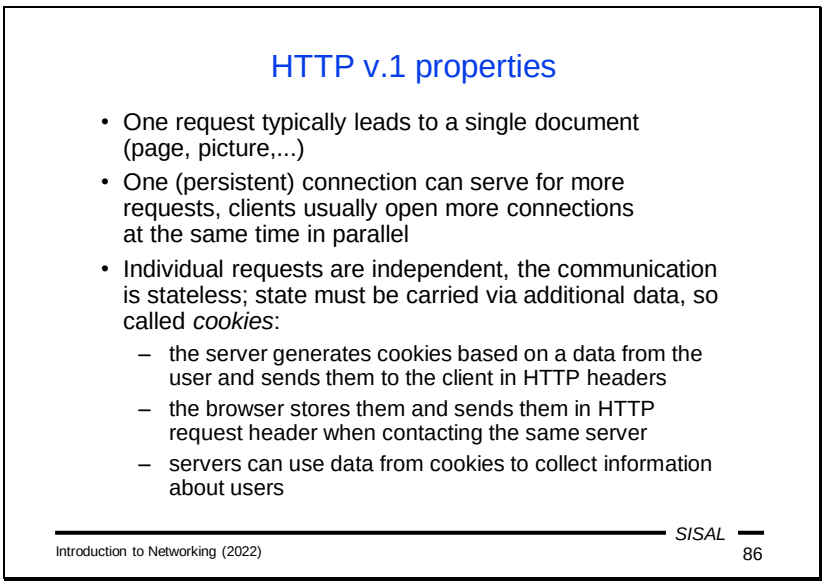
The most common HTTP methods are GET, HEAD, POST, PUT, DELETE, and CONNECT.

- GET is the basic method by which a client requests a page or document. The request body is empty; the response body contains the requested page. In the case of an error, the body of the response (as with other methods) usually contains the text of an HTML page describing the error. This method is declared **safe** in the RFC specification, which means that it **does not change** the content of the server. Even if a request was created by filling out and submitting an HTML form and it contains the form data in the query section of the URL, it should not cause the data stored on the server to change. This method is also defined as *idempotent*, meaning that **repeated** use has the **same** effect.
- HEAD is a simplified form of the GET method, where the server replies only with headers and not a document. Its purpose is to find out whether the requested page is available, how large it is, whether it is available in a certain language, etc.
- POST is a method whose purpose is to send certain information (usually data from a form) to a server and get the content of a certain document in response. It can be used as a way to get a dynamic page, but it is also valid to **change** the content of the data on the server by using this method. Therefore, the method is not idempotent, because each call can have a different effect (for example, the content of a file on the server could expand with each call).
- PUT is a method that overwrites the content of a document on the server with the document sent in the request. So it is definitely not safe, but it is idempotent – repeating the request should still have the same effect.
- DELETE is a method for deleting a document on the server. This method is also idempotent – when repeated we get another message (saying that the document

has already been deleted), but the result is in any case the non-existence of the document.

- CONNECT is a specific method which opens a connection according to the specified parameters and thus actually builds a tunnel through which it is possible to forward another connection in a completely different protocol via an HTTP connection. While this is a useful feature that web servers often use, it also poses a security risk because it allows you to bypass network security policy restrictions.

Slide 86

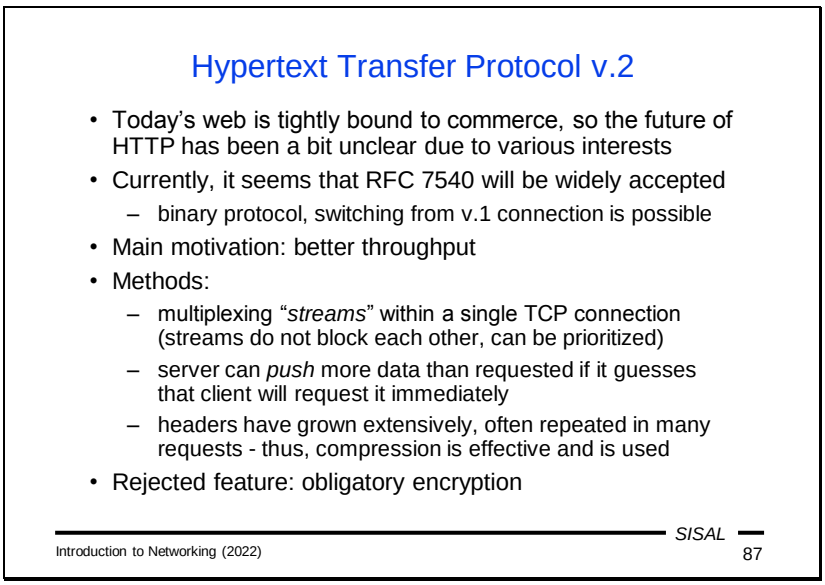


The way a web browser is used leads some users to think that when they visit a server, a communication channel is established between the client and the server, which transmits data back and forth for the entire duration if their visit to the server. But this is completely at odds with reality. In fact, each user request is sent by the client as completely **independent**. If a web page consists of text and three images, then there will even be four independent requests. Clicking on a link on the page will generate several more independent HTTP requests. Version 1.1 introduced a new feature, a **persistent** connection, which means that after one request, the client does not have to close the TCP connection and establish a new one, but can use the same TCP connection for multiple serial requests. In addition, browsers often manage this by opening multiple parallel TCP connections directly when connecting to a server, as they expect most pages to contain multiple components from the same server, so it makes sense to have appropriate TCP connections ready for subsequent requests.

The fact that the individual requests are independent means that the entire communication is **stateless**, or that the server generally has no idea of which requests from the same client "belong together" or about the current state of the interaction with a given user. If a user sends a server some information that will be needed for its further work with that user (e.g. some settings), the server has to receive it again from the client on each request. The problem of stateless communication is solved using so-called *cookies*. A cookie is a piece of data that the server generates based on information from the user and sends to the client when responding, in the form of Set-Cookie (or Set-Cookie2) headers. The browser saves this data and then sends it in the form of a Cookie header each time another request is submitted to the same server. A cookie lets a server identify the connection and/or user and ensure that the user's request is processed in a server environment that matches their existing (or previous) interaction.

This nature of cookies is important because it has several security implications:

- Cookies themselves do not pose any direct risk; they cannot, for example, contain viruses, etc.
- However, they pose a secondary risk, because a web server can store any information it finds about the user in them and then use it at will. For example, with the help of advertising banners placed on various pages, a server can collect comprehensive information about a user and carry out personalized advertising.
- There is also a risk if cookies stored on your computer fall into foreign hands.



The newest version of the HTTP protocol tries to capture current trends in web usage in order to increase "speed" from the user's perspective if possible. A fundamental difference is the change from a textual form to a binary one. This is, of course, annoying for many reasons: with version 1, a more experienced user appreciates that he can try to contact a server directly, enter his request and see exactly how the server responds, and administrators appreciate the ability to monitor any malicious activity of users or servers. However, this idyllic age of HTTP transparency did not actually end with HTTPv2, but rather with the massive adoption of HTTPS. At the transport layer, we now rarely see HTTP in text form, so this change to a binary protocol is not really that crucial from an operational point of view.

The authors of version 2 were not satisfied with the services of TCP and created the concept of multiple HTTP *streams* running over a single TCP connection. These custom streams are more efficient because they do not block each other and can be prioritized according to the needs of the application layer.

In version 2, the server can also send a client data that it has not yet requested, but that the server expects that the client will need in a while. A classic example is a page with images – in version 1 we learned that the client must create a new request for each image, but in version 2 it is not necessary.

Another feature of the new version of the protocol is the ability to compress headers. Especially with the increased use of authentication, the headers have also swelled a lot, moreover, they also often have the same content (authentication, offered languages, coding; all these will not change practically throughout the entire dialogue and it is unnecessary to send everything again with each request).

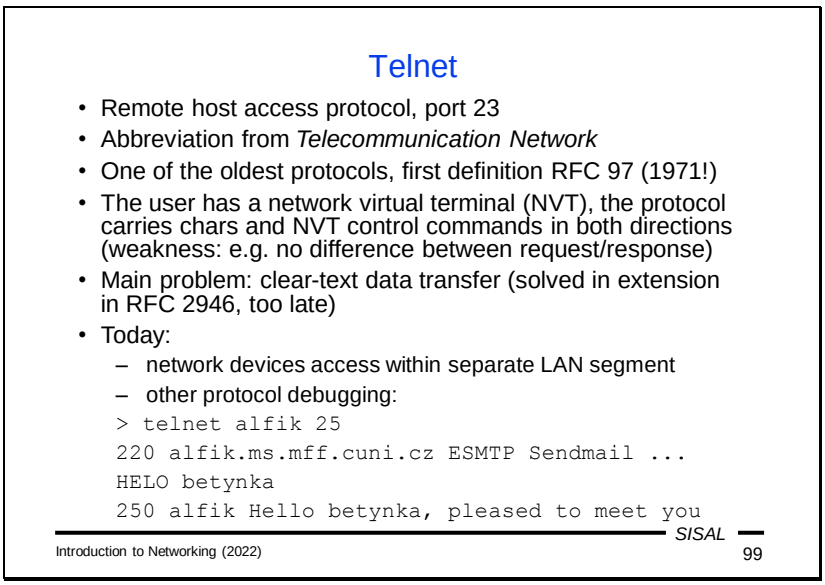
Hypertext Markup Language

- Progress in past years has been a little bit dramatic, 2014 released a compromise version 5
- Textual page content is supplemented by meta-tags: structural (e.g. paragraph), semantic (e.g. post address), formatting (e.g. bold)
- Application of older SGML (Standard Generalized ML) and predecessor to XML (Extensible ML)

- Tag form: **<***tag* [*attributes*]**>**
- Whitespaces not significant
- Special chars - entities (**`<`**, **`>`**, **`&`**, **` `**...)
- Comments (**`<!-- ... -->`**)

The curriculum on HTML and related topics will be taught in the second part of the course.

Slide 99



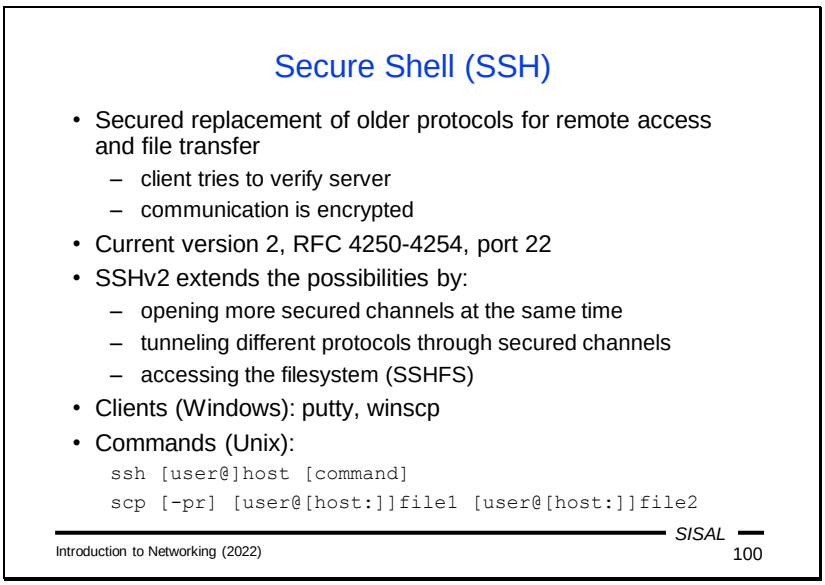
Telnet is another very old protocol and represents the first solution for remote login to another machine, in the form of terminal emulation. The client and server transmit user commands and host reactions so that the user has an environment similar to sitting at a real terminal.

An educational element of this communication is the way in which control commands are sent. The principle of terminal emulation is based on the fact that the client and the server must agree on which part of the work each one will do. When the user presses a key, someone must cause it to be displayed on the screen (a so-called echo). The client can either do it itself or wait until the server sends back an instruction to display the character after it as processed the keystroke. Conversely, if the user is typing a password, neither the client nor the server will echo. But they have to agree on that. There are four messages DO ECHO, DON'T ECHO, WILL ECHO and WON'T ECHO. The problem is that the message **does not indicate** whether it is a command or a response! So if the meaning of messages is accidentally out of sync, it has fatal consequences. Let's say a client sends a command DO ECHO, and because no response is received, it repeats it, but does not note that it was sent twice. The server correctly responds twice WILL ECHO, but the client will consider the second response as a new message and respond to it DO ECHO. This starts an endless loop of extremely fast DO ECHO / WILL ECHO message exchanges. The lesson is not to save money at all cost in the protocol planning phase at all costs and definitely sacrifice one bit per request/response flag.

From a security point of view, Telnet has all the weaknesses of the old Internet protocols. Open password transmission was solved by RFC 2946, but it came at a time when there was already a better replacement, SSH. However, even today, due to its relative simplicity, the protocol is used in places where there is no risk of

password interception – for example, on a separate segment of a local network intended for infrastructure management. However, even these devices are switching to SSH over time.

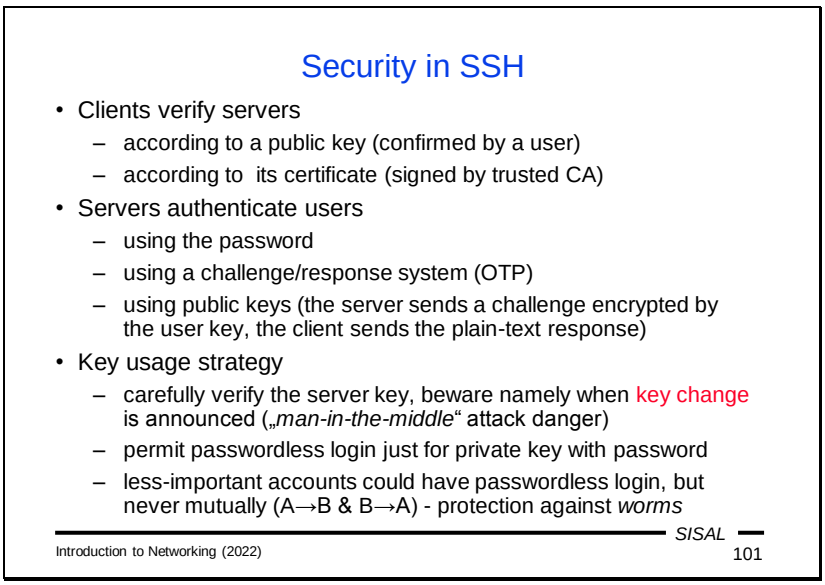
However, the **telnet** application can still be used for another purpose, namely to establish a connection to a server working in another (ideally text-based) protocol. For example, we can test how an SMTP or HTTP server responds to individual commands.



SSH (Secure shell) is a protocol used for remote login and file transfers. Unlike older protocols, it is based on consistent authentication of the server to which we connect and encryption of the entire communication.

Currently, version 2 is used, which expands the capabilities of SSH with the following:
- opening parallel channels; it is possible, for example, to be logged in using a virtual terminal and to transfer files at the same time,
- tunneling; communication on one side of the SSH channel is transmitted through the channel and on the other side is directed to a local server (a way how to bypass a firewall),
- SSHFS; the server makes part of its file system available through the channel so that it appears to the client as a local file system.

On the MS Windows platform, freely available programs for remote login (e.g. **putty**) and file transfer (e.g. **WinSCP**) are available; on UNIX there are line commands **ssh** and **scp** for this purpose.



Before a client starts communicating with the server about user authentication, it verifies that it is actually connected to the correct server. Either a server certificate is used, or the client only displays the fingerprint of the server key to the user and the user decides whether it is the correct key and therefore the correct server. If the user confirms the identity of the server, the application saves this decision and does not ask again at the next login.

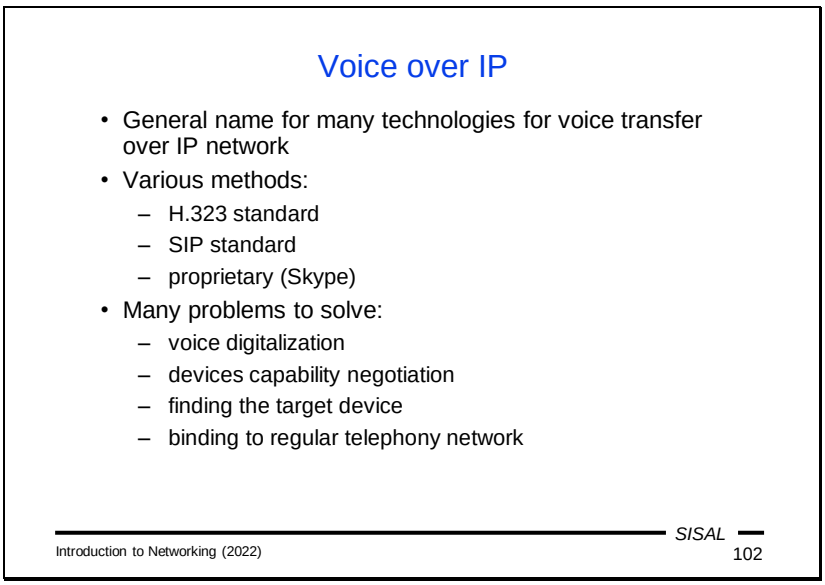
If we are logging in to a server for the first time using SSH, theoretically we should always thoroughly verify the correctness of the key that the server sends. In fact, the risk is normally very low. We must evaluate how important the server is, how likely it is to be attacked, how secret the password is that we have on that server (on how many other servers do we have the same password), and if we evaluate these risks as small, verification can be skipped. However, this is definitely **not the case** if the message about the key change on the server arrives **unexpectedly**. If the server we have been logging in to for a year suddenly uses a different key, it is necessary to pay attention and thoroughly verify everything! This could be a symptom of an attack. For example, we can verify with the server administrators by phone that they have really reinstalled the system and have a new key. If such verification is not possible, we should **terminate** logging in.

Once the server is verified, the client begins exchanging information with it to authenticate the user. The most basic option is to use a name and password. At this point, the communication is already encrypted, so there is no risk of disclosure. The second common way is to use keys. The user creates a public/private key pair on the client (or can share the same key pair on multiple clients), and stores the public key in a specified location on the server. If such a key exists and the client can prove that

he holds the corresponding private key, the user does not have to enter the password for the remote account.

Setting up logging with a key is very convenient, but it carries a risk. If our account is attacked, we open the door to other servers. Many people use a two-tier key system: they use one pair of keys on machines of low importance and another pair on important machines, while the secret key from the latter pair is protected by a local password. If the key is not password protected, an attacker can use it to log onto other machines. Theoretically, the attacker has no information about what other machine he should try to log into. However, he has information about other computers **from which** you are allowed to access this account (just going through the list of stored public keys). So he can try to log into each account from this list, in case you also happen to have reciprocally set up login without a password there. And if so, he'll get to another computer. This is the working principle of Internet **worms**. Protection against this kind of attack is simple – avoid having logins enabled between your two accounts back and forth, and if you really need to have it set up like this, password protect the secret keys.

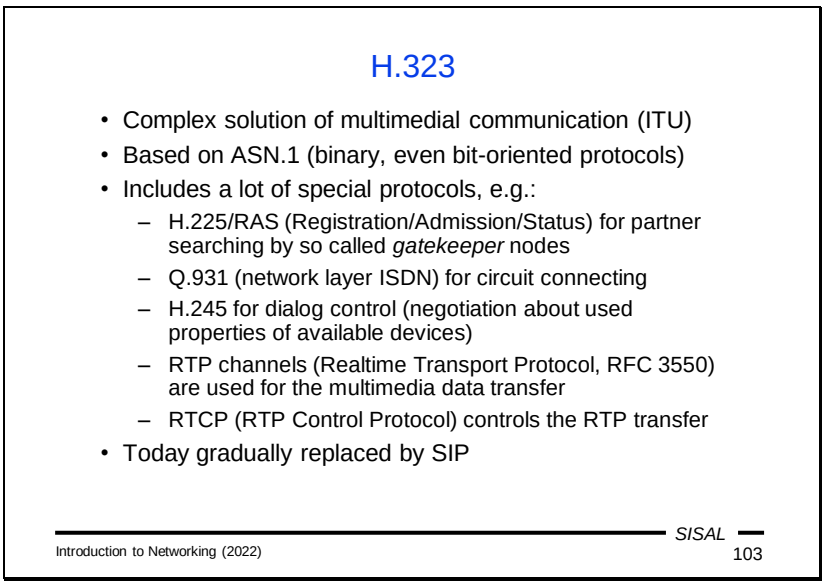
Slide 102



The term **Voice over IP** does not refer to one specific protocol, but generally to any tool for transmitting (not only) voice over a TCP/IP network. There are a number of options for such a transmission, some of which are implemented through more general protocols (such as Skype over HTTP), but we will be mainly interested in those that use their own protocols.

This whole issue is of course very complex. It includes the area of voice digitization with all its methods, variants, parameters, etc., it includes how the counterparties agree on the communication parameters for both sides and how to adapt the behavior of one counterparty to another, there must be some way to find a partner for communication and ideally to interconnect the entire system to a regular telephone network so that it is possible to communicate from a regular telephone to a number represented by a VoIP device and vice versa.

Slide 103



Telecommunication companies (or ITU, International Telegraph Union) long ago embarked on a path to digitize individual processes. There are a number of separate protocols for each area, collectively referred to as **H.323**, which address the issues of searching of a partner, connection establishment, agreement on device properties, and finally the actual transmission of audio or video data. Unfortunately, these protocols are not all freely available and come from a time when data flows did not have the capacity we have today, and therefore they were designed to desperately save every bit. Yes, these protocols are not only non-textual, but are **binary** in the true sense of the word. The boundaries of the values here are not bytes, but **bits**, and in addition, each bit can affect the boundaries of other values. The protocols are not only unreadable by a text editor, but even very difficult to implement – imagine how you would program the reading of a ten-bit number that spans three bytes, but only sometimes…

The protocols in charge of the initial phase of call preparation are now gradually being replaced by the SIP protocol. The audio or video data itself is sent using the Realtime Transport Protocol (RTP) and the RTCP protocol is used to control its operation. Although both protocols are also binary this is adequate for their purpose, since their task is to transmit binary multimedia data; in addition, their description is freely available in RFC 3550.

Abstract Syntax Notation 1

- Formal definition of data structures, e.g.:

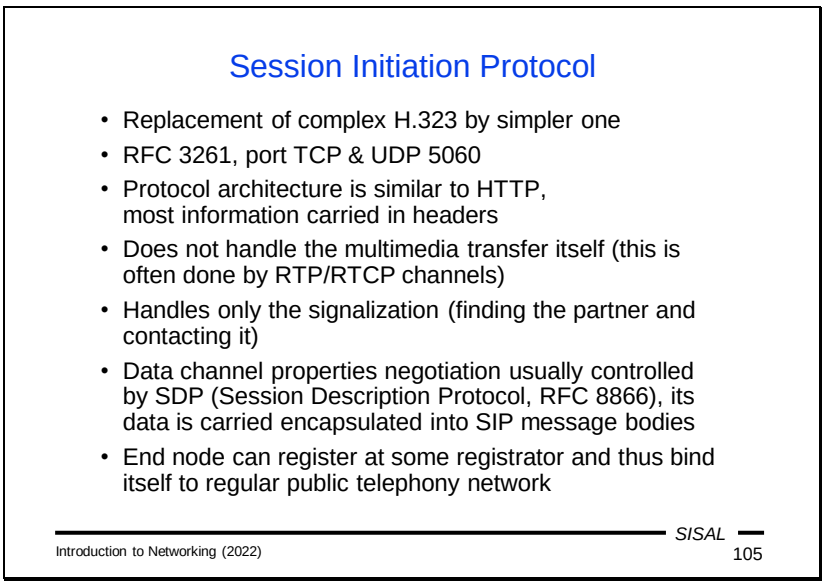
```
Answer ::= CHOICE {
   word PrintableString,
   flag BOOLEAN }

AlgorithmIdentifiers ::= SET OF AlgorithmIdentifier

SignedData ::= SEQUENCE {
   version CMSVersion,
   digestAlgorithms AlgorithmIdentifiers,
```

- Comes from the 80's (and implementation looks like this)
  - e.g.: enumeration value is by default stored into as many **bits** as needed, preceded by a bit signaling an extension, meaning a **different** number of bits used for this value
- Automatic generation of protocol parser possible
- Usage examples: H.323, X.509

The definition of H.323 protocols is based on a method called Abstract Syntax Notation. It is a method of defining a data structure or content of protocol data units using a formal definition with mnemonic identifiers (including enumerations, sets of named values) and special constructs to express a sequence (record), an array, selection from several alternatives (something like case or select constructs from programming languages), etc. As a means of formal description, it is a very useful tool.

The problem is its default implementation. As I have already indicated, each value is written only as many bits as needed for the item. However, the authors anticipated that protocols were evolving and built into the design a general mechanism for indicating whether each value uses the original design or its extension. Thus, each actual value is preceded by one bit, which tells whether the value is extended or not, and thus actually determines how many bits it actually occupies… The implementation of such an encoding is extremely complex and is solved by purchasing a library that converts a text notation in ASN.1 to source code that implements writing and reading it.

In addition to this extremely economical variant of value encoding, there are also variants that at least respect the byte boundaries, so that their decoding is much easier.

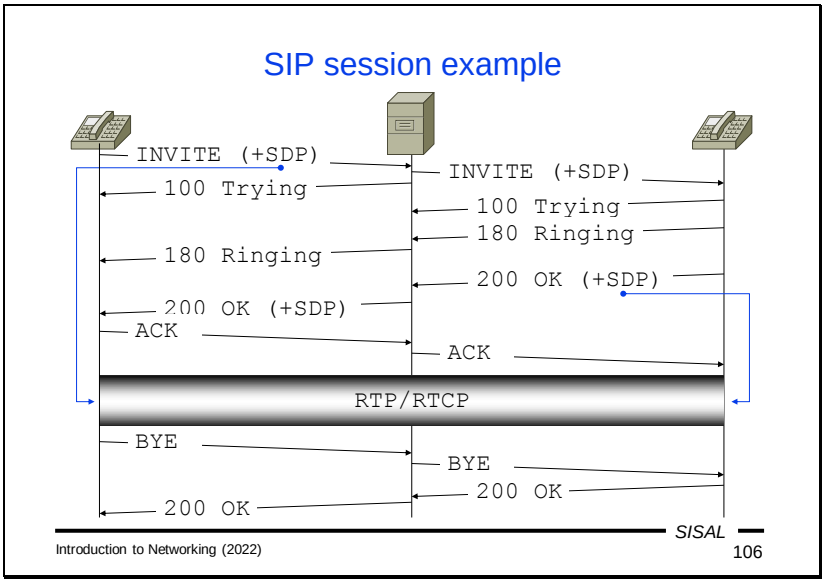


Session Initiation Protocol (SIP) is a more modern solution for tasks related to establishing connections and negotiating device properties. Unlike the H.323 family of protocols, it is a text protocol with a message structure that resembles HTTP, but can run on both TCP and UDP. The advantage is that the protocol messages are simply readable. Today, this protocol is gradually replacing H.323 and is becoming the standard for building a telephone network in newly designed buildings.

It is interesting that this protocol is one of the first that already in the initial proposal introduces the term **proxy**, i.e. a node in the communication chain that facilitates communication across the boundaries of various networks, including private ones. As with SMTP, special headers are inserted into the message during transmission that record the path (here they are called **Via** and **Record-Route**) and according to which the other party and all nodes along the path can correctly route the response. Likewise, the headers include data such as a call identifier, the caller and the called SIP URL (usually a number plus a domain), etc.

SIP itself only solves the problem of finding a destination, finding a route and establishing a connection. The next stage, i.e. the agreement on device properties and data channel parameters, is solved by the Session Description Protocol (SDP). It is again a text protocol consisting of lines of the format *keyword*=*value* and is transmitted using SIP protocol messages. A SIP message carrying SDP information has the MIME type **application/sdp** and there is an SDP block in the body of the message.

As soon as the counterparties agree on the form of the data channels and open them, they will start sending audio (and possibly also video) data via them, using the RTP/RTCP protocols.



An SIP call example:

- The calling device sends an INVITE command, which contains the URL being called and an offer of data channels as an SDP message.
- The command arrives at the nearest proxy, which begins to resolve the target. Depending on its configuration and the destination URL, it will find another node in the path to the device being called. For simplicity, we will imagine a situation where the target network is directly behind the proxy. In this case, the proxy sends the request to the target device, but checks the contents of the SDP message and modifies it so that it is usable for the partner – it de facto makes adjustments in it corresponding to the address translation (NAT) function.
- At the same time, the proxy sends a temporary 100 Trying response to the calling device.
- Once the called device processes the request, it also sends a temporary response 100 Trying. This response is a *node-to-node* message and therefore the proxy does not forward it to the calling device.
- The called device starts ringing and informs the calling device with another temporary response 180 Ringing. Unlike the previous response, it has an *end-to-end* character, so it is forwarded by the proxy to the calling device.
- If the called party picks up the phone, the called device will send a final answer 200 OK. This response also carries an SDP message with an offer of data channels on the called device.
- When this message arrives at the proxy, it checks the contents of the SDP again, adjusts it to address translation needs, and sends the message to the calling device.
- To complete the initial phase, it is still necessary for the calling device to acknowledge receipt of the SDP message, and therefore the entire phase ends by sending an acknowledgement message (command) ACK.

- From this point on, both devices can start sending multimedia data over RTP/RTCP data channels offered to them by the other party.
- The call is terminated by any end device by sending a BYE command and receiving the 200 OK answer. Of course, the proxy also responds to this pair of messages and cancels its data channels, which it has opened toward both parties.

Summary 5

- What is the purpose of the IMAP protocol?
- What is the purpose of HTML?
- Describe what happens in the HTTP/1.1 protocol if a user requests a page with two embedded images.
- Briefly describe the purpose and properties of the most common HTTP methods.
- What are cookies for and what are their risks?
- Compare the Telnet and SSH protocols.
- Compare the H.323 and SIP protocols.