Network Applications and the Web

Antonio Carzaniga

Faculty of Informatics
Università della Svizzera italiana

October 4, 2017

Outline

- General concepts for network applications
- Client/server architecture
- The world-wide web
- Basics of the HTTP protocol

Examples of Network Applications

Examples of Network Applications

- The world-wide web

Examples of Network Applications

- The world-wide web
- Electronic mail

Examples of Network Applications

- The world-wide web
- Electronic mail
- Instant messaging

Examples of Network Applications

- The world-wide web
- Electronic mail
- Instant messaging
- Peer-to-peer file sharing

Examples of Network Applications

- The world-wide web
- Electronic mail
- Instant messaging
- Peer-to-peer file sharing
- Video streaming

Examples of Network Applications

- The world-wide web
- Electronic mail
- Instant messaging
- Peer-to-peer file sharing
- Video streaming
- Multi-user networked games

Examples of Network Applications

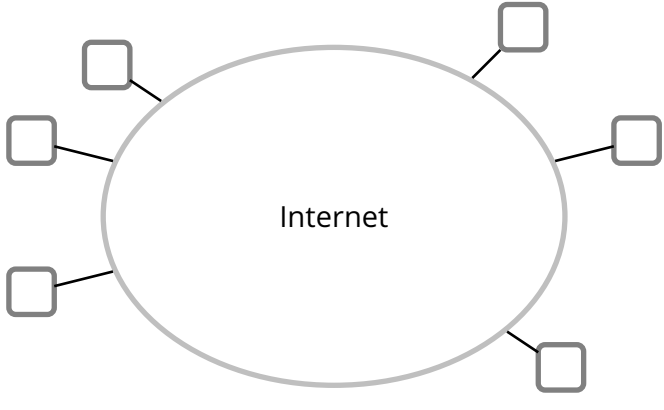
- The world-wide web
- Electronic mail
- Instant messaging
- Peer-to-peer file sharing
- Video streaming
- Multi-user networked games
- . . .
- Remote login
- . . .

Examples of Network Applications

- The world-wide web
- Electronic mail
- Instant messaging
- Peer-to-peer file sharing
- Video streaming
- Multi-user networked games
- . . .
- Remote login
- . . .
- Remote on-line banking
- Network telephony
- . . .

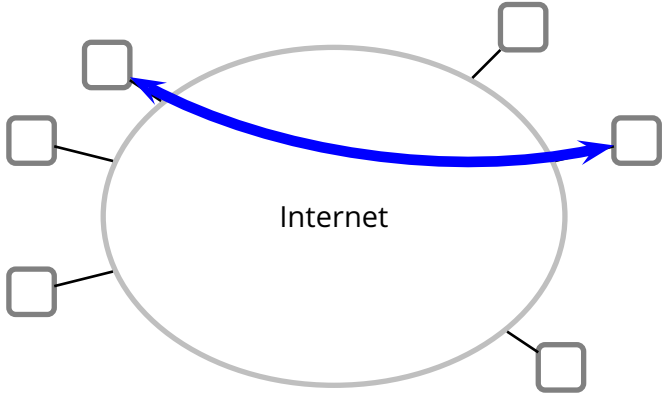
End System Applications

Internet applications are *end system* applications



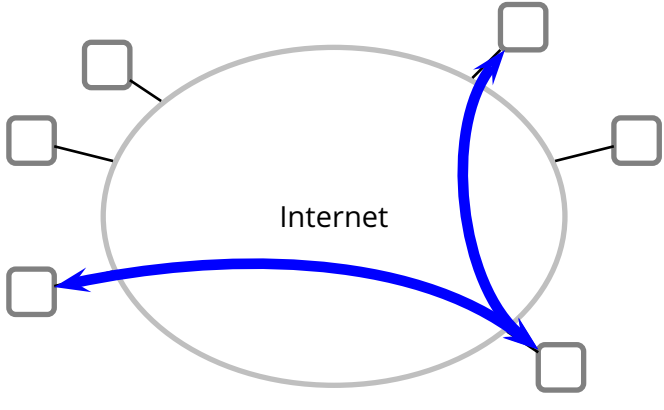
End System Applications

Internet applications are *end system* applications



End System Applications

Internet applications are *end system* applications



Processes

- A *process* is an execution of a program

Processes

- A *process* is an execution of a program
- A single *sequential* program
  - i.e., a single thread

Processes

- A *process* is an execution of a program
- A single *sequential* program
  - i.e., a single thread
- Processes may exchange messages
  - obviously, received messages can be considered as input to a process (program)

Processes

- A *process* is an execution of a program
- A single *sequential* program
  - i.e., a single thread
- Processes may exchange messages
  - obviously, received messages can be considered as input to a process (program)
- Different processes may be running on different end systems
  - possibly on different computers
  - running different operating systems
  - a process must be able to *address* another specific process

Example

```
while(browsing) {
  url = read_url(keyboard);
  page = get_web_page(url);
  display_web_page(page);
}
```

```
while(serving_pages) {
  page_name = read_web_request(network);
  page = read_file(page_name, disk);
  write_page(page, network);
}
```

Example

```
while(chatting) {
  msg = read_message(keyboard);
  write_message(msg, network);
  msg = read_message(network);
  write_message(msg, screen);
}
```

```
while(chatting) {
  msg = read_message(network);
  write_message(msg, screen);
  msg = read_message(keyboard);
  write_message(msg, network);
}
```

Clients and Servers

- For each pair of communicating processes, we distinguish two *roles*

Clients and Servers

- For each pair of communicating processes, we distinguish two *roles*
- ***Client:*** process that ***initiates the communication***
  - specifically, if the communication is carried over a connection-oriented service, then the client is the process that establishes the connection

Clients and Servers

- For each pair of communicating processes, we distinguish two *roles*
- ***Client:*** process that ***initiates the communication***
  - specifically, if the communication is carried over a connection-oriented service, then the client is the process that establishes the connection
- ***Server:*** process that ***waits to be contacted***
  - specifically, if the communication is carried over a connection-oriented service, then the server is the process that passively accepts the connection

Clients and Servers

- For each pair of communicating processes, we distinguish two *roles*
- ***Client:*** process that ***initiates the communication***
  - specifically, if the communication is carried over a connection-oriented service, then the client is the process that establishes the connection
- ***Server:*** process that ***waits to be contacted***
  - specifically, if the communication is carried over a connection-oriented service, then the server is the process that passively accepts the connection
- Some applications have processes that act both as clients and servers. This is often called *peer-to-peer* architecture

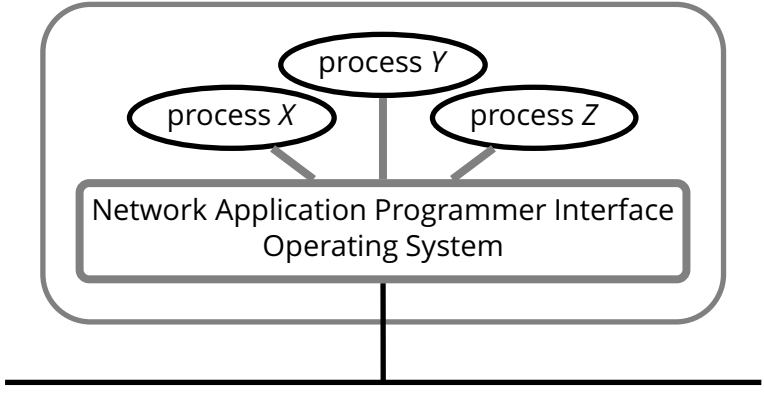
Clients and Servers

- For each pair of communicating processes, we distinguish two *roles*
- ***Client:*** process that ***initiates the communication***
  - specifically, if the communication is carried over a connection-oriented service, then the client is the process that establishes the connection
- ***Server:*** process that ***waits to be contacted***
  - specifically, if the communication is carried over a connection-oriented service, then the server is the process that passively accepts the connection
- Some applications have processes that act both as clients and servers. This is often called *peer-to-peer* architecture
- *Caveat:* this classification is useful, but it is little more than nomenclature. Some applications and protocols mix and confuse those terms (e.g., FTP)

Processes and Hosts

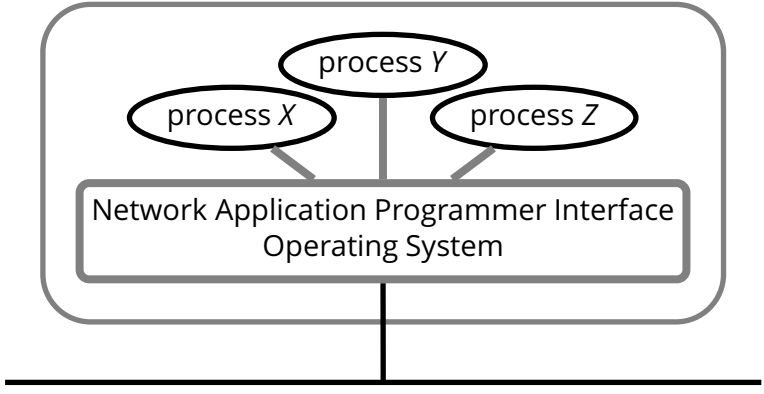
- An end system (host) may run multiple processes

- An end system (host) may run multiple processes



Processes and Hosts

- An end system (host) may run multiple processes



- A process is addressed (within its host) by its ***port number***

- The *operating system* manages the network interfaces

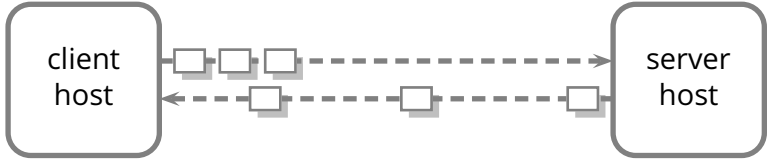
Sockets

- The *operating system* manages the network interfaces
- Applications use the network through *sockets*

Sockets

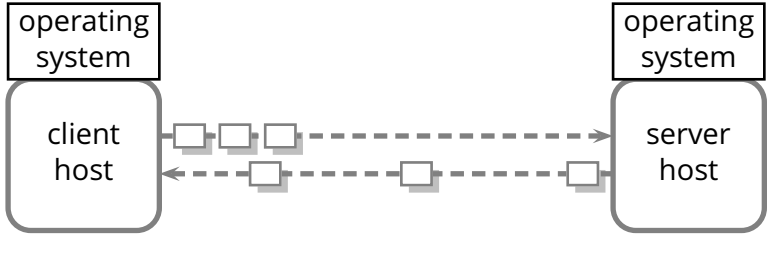
- The *operating system* manages the network interfaces
- Applications use the network through *sockets*

- The *operating system* manages the network interfaces
- Applications use the network through *sockets*

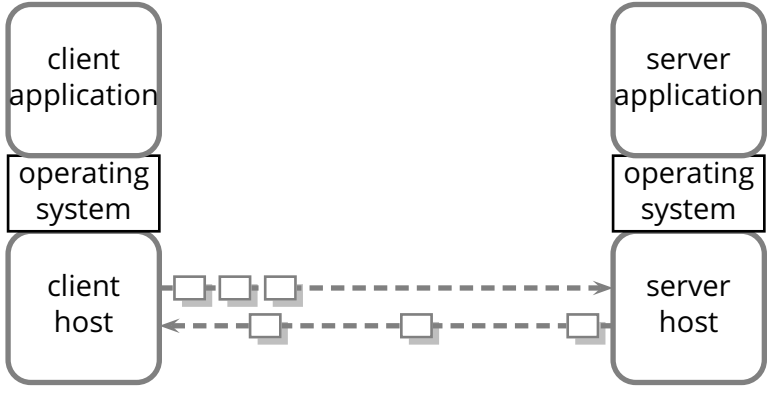


Sockets

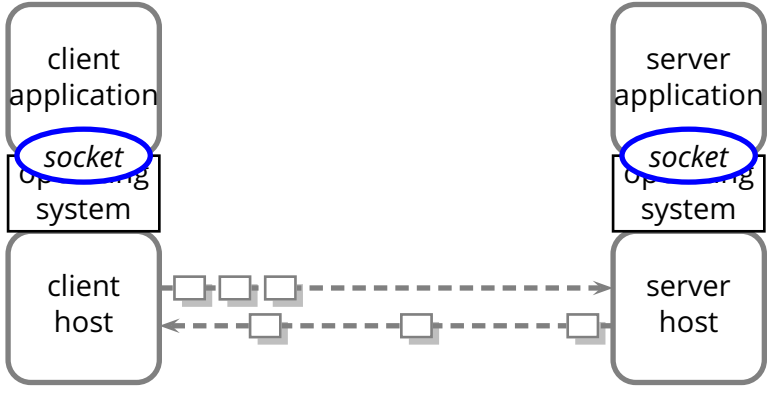
- The *operating system* manages the network interfaces
- Applications use the network through *sockets*



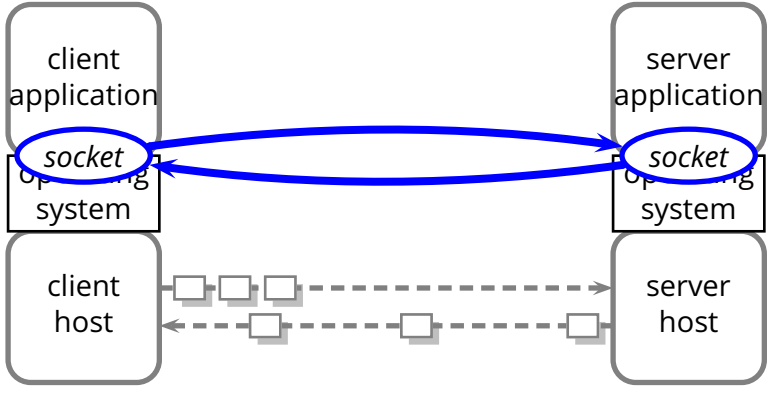
- The *operating system* manages the network interfaces
- Applications use the network through *sockets*



- The *operating system* manages the network interfaces
- Applications use the network through *sockets*



- The *operating system* manages the network interfaces
- Applications use the network through *sockets*



Application Programs

- Client application

- Client application
  1. create a socket *C* by "connecting" to the server application
     - i.e., connect to host *H* on port *P*

- Client application
  1. create a socket *C* by "connecting" to the server application
     - i.e., connect to host *H* on port *P*
  2. use socket *C* by reading and writing data into it
     - this is the body of the client application protocol

- Client application
  1. create a socket *C* by "connecting" to the server application
     - i.e., connect to host *H* on port *P*
  2. use socket *C* by reading and writing data into it
     - this is the body of the client application protocol
  3. disconnect and destroy *C*

Application Programs

- Client application
  1. create a socket *C* by "connecting" to the server application
     - i.e., connect to host *H* on port *P*
  2. use socket *C* by reading and writing data into it
     - this is the body of the client application protocol
  3. disconnect and destroy *C*
- Server application (running on host *H*)

Application Programs

- Client application
  1. create a socket *C* by "connecting" to the server application
     - i.e., connect to host *H* on port *P*
  2. use socket *C* by reading and writing data into it
     - this is the body of the client application protocol
  3. disconnect and destroy *C*

- Server application (running on host *H*)
  1. create a socket *S* by "accepting" a connection on port *P*
     - a port is often called a "server socket"

Application Programs

- Client application
  1. create a socket *C* by "connecting" to the server application
     - i.e., connect to host *H* on port *P*
  2. use socket *C* by reading and writing data into it
     - this is the body of the client application protocol
  3. disconnect and destroy *C*

- Server application (running on host *H*)
  1. create a socket *S* by "accepting" a connection on port *P*
     - a port is often called a "server socket"
  2. use socket *S* by reading and writing data into it
     - this is the body of the server application protocol

Application Programs

- Client application
  1. create a socket *C* by "connecting" to the server application
     - i.e., connect to host *H* on port *P*
  2. use socket *C* by reading and writing data into it
     - this is the body of the client application protocol
  3. disconnect and destroy *C*

- Server application (running on host *H*)
  1. create a socket *S* by "accepting" a connection on port *P*
     - a port is often called a "server socket"
  2. use socket *S* by reading and writing data into it
     - this is the body of the server application protocol
  3. disconnect and destroy *S*

Example 3 (HTTP)

```
while(browsing) {
  url = read_url(keyboard);
  socket = open_connection(url);
  request = compose_http_request(url);
  write_message(request, socket);
  reply = read_message(socket);
  display_web_page(reply); }
```

```
while(serving_http) {
  socket = accept_connection();
  request = read_message(socket);
  reply = serve_http_request(request);
  write_message(reply, socket); }
```

Example 3 (HTTP)

```
while(browsing) {
  url = read_url(keyboard);
  socket = open_connection(url);
  request = compose_http_request(url);
  write_message(request, socket);
  reply = read_message(socket);
  display_web_page(reply); }
```

```
while(serving_http) {
  socket = accept_connection();
  request = read_message(socket);
  reply = serve_http_request(request);
  write_message(reply, socket); }
```

Example 3 (HTTP)

```
while(browsing) {
  url = read_url(keyboard);
  socket = open_connection(url);
  request = compose_http_request(url);
  write_message(request, socket);
  reply = read_message(socket);
  display_web_page(reply); }
```

```
while(serving_http) {
  socket = accept_connection();
  request = read_message(socket);
  reply = serve_http_request(request);
  write_message(reply, socket); }
```

Example 3 (HTTP)

```
while(browsing) {
  url = read_url(keyboard);
  socket = open_connection(url);
  request = compose_http_request(url);
  write_message(request, socket);
  reply = read_message(socket);
  display_web_page(reply); }
```

```
while(serving_http) {
  socket = accept_connection();
  request = read_message(socket);
  reply = serve_http_request(request);
  write_message(reply, socket); }
```

Example 3 (HTTP)

```
while(browsing) {
  url = read_url(keyboard);
  socket = open_connection(url);
  request = compose_http_request(url);
  write_message(request, socket);
  reply = read_message(socket);
  display_web_page(reply); }
```

```
while(serving_http) {
  socket = accept_connection();
  request = read_message(socket);
  reply = serve_http_request(request);
  write_message(reply, socket); }
```

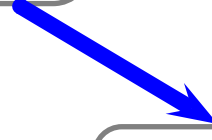
Example 3 (HTTP)

```
while(browsing) {
  url = read_url(keyboard);
  socket = open_connection(url);
  request = compose_http_request(url);
  write_message(request, socket);
  reply = read_message(socket);
  display_web_page(reply); }
```

```
while(serving_http) {
  socket = accept_connection();
  request = read_message(socket);
  reply = serve_http_request(request);
  write_message(reply, socket); }
```

Example 3 (HTTP)

```
while(browsing) {
  url = read_url(keyboard);
  socket = open_connection(url);
  request = compose_http_request(url);
  write_message(request, socket);
  reply = read_message(socket);
  display_web_page(reply); }
```



```
while(serving_http) {
  socket = accept_connection();
  request = read_message(socket);
  reply = serve_http_request(request);
  write_message(reply, socket); }
```

Example 3 (HTTP)

```
while(browsing) {
  url = read_url(keyboard);
  socket = open_connection(url);
  request = compose_http_request(url);
  write_message(request, socket);
  reply = read_message(socket);
  display_web_page(reply); }
```

```
while(serving_http) {
  socket = accept_connection();
  request = read_message(socket);
  reply = serve_http_request(request);
  write_message(reply, socket); }
```

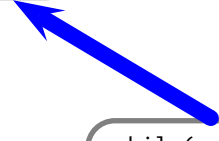
Example 3 (HTTP)

```
while(browsing) {
  url = read_url(keyboard);
  socket = open_connection(url);
  request = compose_http_request(url);
  write_message(request, socket);
  reply = read_message(socket);
  display_web_page(reply); }
```

```
while(serving_http) {
  socket = accept_connection();
  request = read_message(socket);
  reply = serve_http_request(request);
  write_message(reply, socket); }
```

Example 3 (HTTP)

```
while(browsing) {
  url = read_url(keyboard);
  socket = open_connection(url);
  request = compose_http_request(url);
  write_message(request, socket);
  reply = read_message(socket);
  display_web_page(reply); }
```



```
while(serving_http) {
  socket = accept_connection();
  request = read_message(socket);
  reply = serve_http_request(request);
  write_message(reply, socket); }
```

Example 3 (HTTP)

```
while(browsing) {
  url = read_url(keyboard);
  socket = open_connection(url);
  request = compose_http_request(url);
  write_message(request, socket);
  reply = read_message(socket);
  display_web_page(reply); }
```

```
while(serving_http) {
  socket = accept_connection();
  request = read_message(socket);
  reply = serve_http_request(request);
  write_message(reply, socket); }
```

The World-Wide Web

- Developed in the early 1990s
- Based on the idea of ***hypertext*** and ***links***

- Developed in the early 1990s
- Based on the idea of ***hypertext*** and ***links***
- Extremely successful, even though…
  - the *HyperText Transfer Protocol (HTTP)* is just a glorified file transfer protocol
  - the idea of *hypertext* and *links* was already quite old at the time HTTP was developed

- Developed in the early 1990s
- Based on the idea of ***hypertext*** and ***links***
- Extremely successful, even though…
  - the *HyperText Transfer Protocol (HTTP)* is just a glorified file transfer protocol
  - the idea of *hypertext* and *links* was already quite old at the time HTTP was developed
- Success factors
  - simplicity (openness) of the HTML language and
  - simplicity of HTTP (a stateless protocol)
  - low entry barrier for “publishers”
  - GUI browsers (remember Netscape? Or Mosaic?!), search engines (AltaVista?!), etc.

Web Terminology

Web Terminology

- ***document***—a web page is also called a *document*

Web Terminology

- ***document***—a web page is also called a *document*
- ***objects***—a document may contain several objects (images, applets, etc.). An *object* is simply a file

Web Terminology

- ***document***—a web page is also called a *document*
- ***objects***—a document may contain several objects (images, applets, etc.). An *object* is simply a file
- ***URL***—or *Uniform Resource Locator* specifies the address of an object

Web Terminology

- ***document***—a web page is also called a *document*
- ***objects***—a document may contain several objects (images, applets, etc.). An *object* is simply a file
- ***URL***—or *Uniform Resource Locator* specifies the address of an object
- ***browser***—also called *user agent* is the program that users run to get and display documents

Web Terminology

- ***document***—a web page is also called a *document*
- ***objects***—a document may contain several objects (images, applets, etc.). An *object* is simply a file
- ***URL***—or *Uniform Resource Locator* specifies the address of an object
- ***browser***—also called *user agent* is the program that users run to get and display documents
- ***Web server***—is an application that houses objects, and makes them available through the HTTP protocol

Overview HTTP

Overview HTTP

- The main purpose of HTTP is to provide access to Web objects

Overview HTTP

- The main purpose of HTTP is to provide access to Web objects
- Uses a connection-oriented transport mechanism (i.e., TCP)
  - alhough it can also work on UDP

Overview HTTP

- The main purpose of HTTP is to provide access to Web objects
- Uses a connection-oriented transport mechanism (i.e., TCP)
  - alhough it can also work on UDP
- Consists of ***a sequence of requests*** issued by the client, and ***responses*** issued by the server, each one in response to a single request

Overview HTTP

- The main purpose of HTTP is to provide access to Web objects
- Uses a connection-oriented transport mechanism (i.e., TCP)
  - alhough it can also work on UDP
- Consists of ***a sequence of requests*** issued by the client, and ***responses*** issued by the server, each one in response to a single request
- HTTP is *stateless*

Overview HTTP

- The main purpose of HTTP is to provide access to Web objects
- Uses a connection-oriented transport mechanism (i.e., TCP)
  - alhough it can also work on UDP
- Consists of ***a sequence of requests*** issued by the client, and ***responses*** issued by the server, each one in response to a single request
- HTTP is *stateless*
  - the behavior (semantics) of an HTTP request does not depend on any previous request

Example: Request

Example: Request

- Client request

```
GET /carzaniga/index.html HTTP/1.1
Host: www.inf.usi.ch
Connection: close
User-agent: Mozilla/4.0
Accept-Language: it

```

Example: Reply

Example: Reply

- Server reply

```
HTTP/1.1 200 OK
Connection: close
Date: Tue, 15 Mar 2005 10:00:01 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Tue, 8 Mar 2005 16:44:00 GMT
Content-Length: 2557
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-/W3C//DTD HTML 4.01//EN"
. . .
```

Protocol Features

- Request
  - protocol version
  - URL specification
  - connection attributes
  - content/feature negotiation

Protocol Features

- Request
  - protocol version
  - URL specification
  - connection attributes
  - content/feature negotiation
- Reply
  - protocol version
  - reply status/value
  - connection attributes
  - object attributes
  - content specification (type, length)
  - content

```
GET /carzaniga/index.html HTTP/1.1
Host: www.inf.usi.ch
Connection: close
User-agent: Mozilla/4.0
Accept-Language: it
```

```
HTTP/1.1 200 OK
Connection: close
Date: Tue, 15 Mar 2005 10:00:01 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Tue, 8 Mar 2005 16:44:00 GMT
Content-Length: 2557
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-/W3C//DTD HTML 4.01//EN"
. . .
```

Protocol Version

```
GET /carzaniga/index.html HTTP/1.1
Host: www.inf.usi.ch
Connection: close
User-agent: Mozilla/4.0
Accept-Language: it

---

HTTP/1.1 200 OK
Connection: close
Date: Tue, 15 Mar 2005 10:00:01 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Tue, 8 Mar 2005 16:44:00 GMT
Content-Length: 2557
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-/W3C//DTD HTML 4.01//EN"
. . .
```

Protocol Version

Protocol Version

- **Principle:** ***a protocol should always include a version number***
  - usually in the very first bits of the protocol (negotiation messages)

Protocol Version

- **Principle:** ***a protocol should always include a version number***
  - usually in the very first bits of the protocol (negotiation messages)
- A mechanism to negotiate the protocol version allows the protocol design to change
  - *design for change*

```
GET /carzaniga/index.html HTTP/1.1
Host: www.inf.usi.ch
Connection: close
User-agent: Mozilla/4.0
Accept-Language: it
```

```
HTTP/1.1 200 OK
Connection: close
Date: Tue, 15 Mar 2005 10:00:01 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Tue, 8 Mar 2005 16:44:00 GMT
Content-Length: 2557
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-/W3C//DTD HTML 4.01//EN"
. . .
```

```
GET /carzaniga/index.html HTTP/1.1
Host: www.inf.usi.ch
Connection: close
User-agent: Mozilla/4.0
Accept-Language: it

---

HTTP/1.1 200 OK
Connection: close
Date: Tue, 15 Mar 2005 10:00:01 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Tue, 8 Mar 2005 16:44:00 GMT
Content-Length: 2557
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-/W3C//DTD HTML 4.01//EN"
. . .
```

The Host Header

- http://www.inf.usi.ch/carzaniga/index.html

```
GET /carzaniga/index.html HTTP/1.1
Host: www.inf.usi.ch
Connection: close
User-agent: Mozilla/4.0
Accept-Language: it

```

The Host Header

- http://www.inf.usi.ch/carzaniga/index.html

```
GET /carzaniga/index.html HTTP/1.1
Host: www.inf.usi.ch
Connection: close
User-agent: Mozilla/4.0
Accept-Language: it

```

- The *host name* in the URL determines where the request goes
  - host name maps to a network address

The Host Header

- http://www.inf.usi.ch/carzaniga/index.html

```
GET /carzaniga/index.html HTTP/1.1
Host: www.inf.usi.ch
Connection: close
User-agent: Mozilla/4.0
Accept-Language: it
```

- The *host name* in the URL determines where the request goes
  - host name maps to a network address
- The *host name* is also passed as a parameter within the request, so that the server knows the full URL

The Host Header

- http://www.inf.usi.ch/carzaniga/index.html

```
GET /carzaniga/index.html HTTP/1.1
Host: www.inf.usi.ch
Connection: close
User-agent: Mozilla/4.0
Accept-Language: it
```

- The *host name* in the URL determines where the request goes
  - host name maps to a network address
- The *host name* is also passed as a parameter within the request, so that the server knows the full URL
  - this is to allow a single server to serve multiple “virtual” sites (e.g., atelier.inf.usi.ch and www.inf.usi.ch)

```
GET /carzaniga/index.html HTTP/1.1
Host: www.inf.usi.ch
Connection: close
User-agent: Mozilla/4.0
Accept-Language: it
```

```
HTTP/1.1 200 OK
Connection: close
Date: Tue, 15 Mar 2005 10:00:01 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Tue, 8 Mar 2005 16:44:00 GMT
Content-Length: 2557
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-/W3C//DTD HTML 4.01//EN"
. . .
```

Connection

```
GET /carzaniga/index.html HTTP/1.1
Host: www.inf.usi.ch
Connection: close
User-agent: Mozilla/4.0
Accept-Language: it

---

HTTP/1.1 200 OK
Connection: close
Date: Tue, 15 Mar 2005 10:00:01 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Tue, 8 Mar 2005 16:44:00 GMT
Content-Length: 2557
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-/W3C//DTD HTML 4.01//EN"
. . .
```

How HTTP Uses (TCP) Connections

How HTTP Uses (TCP) Connections

- The first version of HTTP used one (TCP) connection per object
  - inefficient use of the network
  - inefficient use of the operating system

How HTTP Uses (TCP) Connections

- The first version of HTTP used one (TCP) connection per object
  - inefficient use of the network
  - inefficient use of the operating system

- HTTP/1.1 introduces *persistent* connections
  - the same (TCP) connection can be used by the client to issue multiple request, and by the server to return multiple replies, and possibly multiple objects

How HTTP Uses (TCP) Connections

- The first version of HTTP used one (TCP) connection per object
  - inefficient use of the network
  - inefficient use of the operating system

- HTTP/1.1 introduces *persistent* connections
  - the same (TCP) connection can be used by the client to issue multiple request, and by the server to return multiple replies, and possibly multiple objects
  - the default behavior is to use persistent connections

How HTTP Uses (TCP) Connections

- The first version of HTTP used one (TCP) connection per object
  - inefficient use of the network
  - inefficient use of the operating system

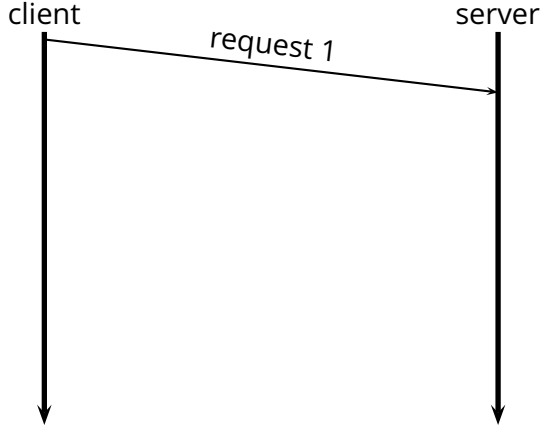
- HTTP/1.1 introduces *persistent* connections
  - the same (TCP) connection can be used by the client to issue multiple request, and by the server to return multiple replies, and possibly multiple objects
  - the default behavior is to use persistent connections
  - "`Connection: close`" in the request and response indicates the intention, of the client and server, respectively, to *not* use a persistent connection

How HTTP Uses Persistent Connections

- A persistent connection can be used to request and transfer two or more objects

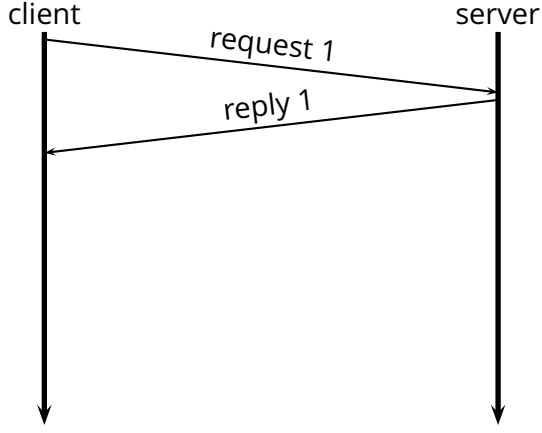
How HTTP Uses Persistent Connections

- A persistent connection can be used to request and transfer two or more objects



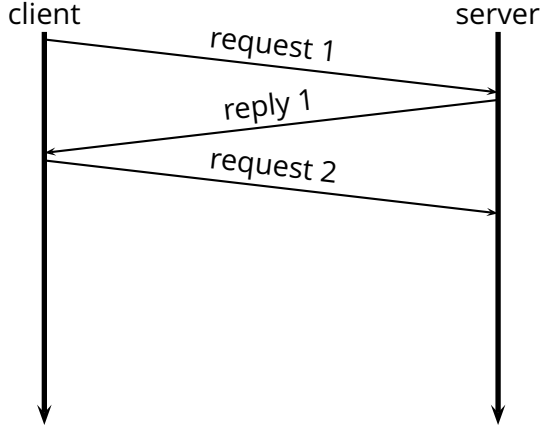
How HTTP Uses Persistent Connections

- A persistent connection can be used to request and transfer two or more objects



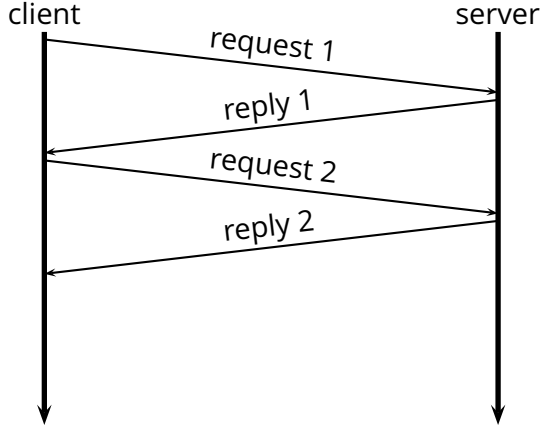
How HTTP Uses Persistent Connections

- A persistent connection can be used to request and transfer two or more objects



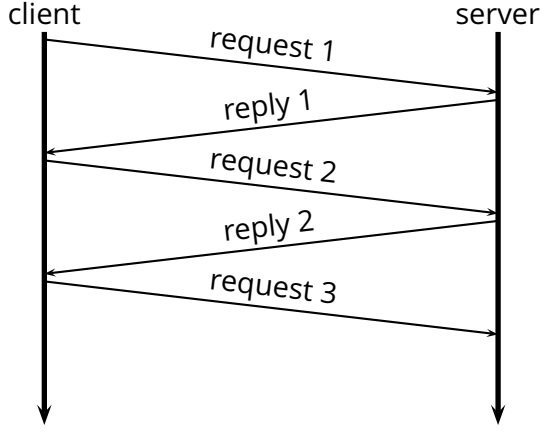
How HTTP Uses Persistent Connections

- A persistent connection can be used to request and transfer two or more objects



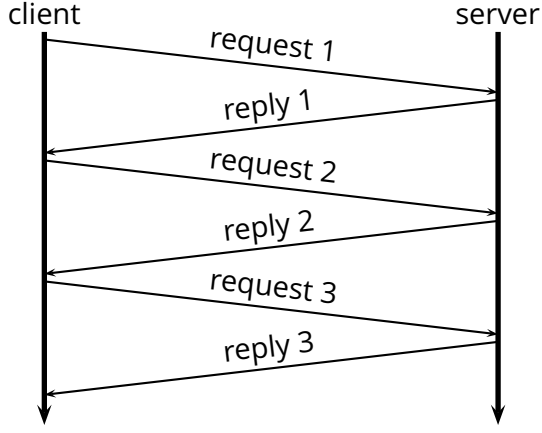
How HTTP Uses Persistent Connections

- A persistent connection can be used to request and transfer two or more objects



How HTTP Uses Persistent Connections

- A persistent connection can be used to request and transfer two or more objects

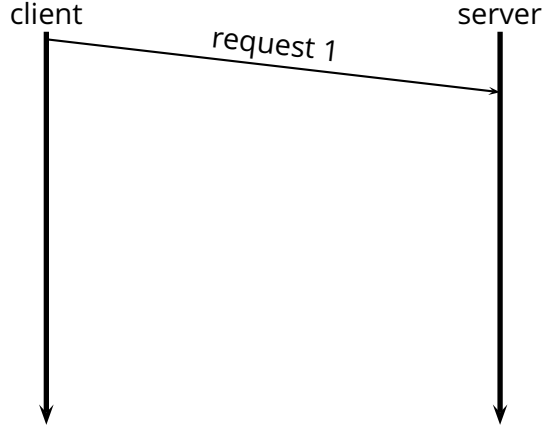


Persistent Connections With Pipelining

- A more efficient use of a connection is by *pipelining* requests

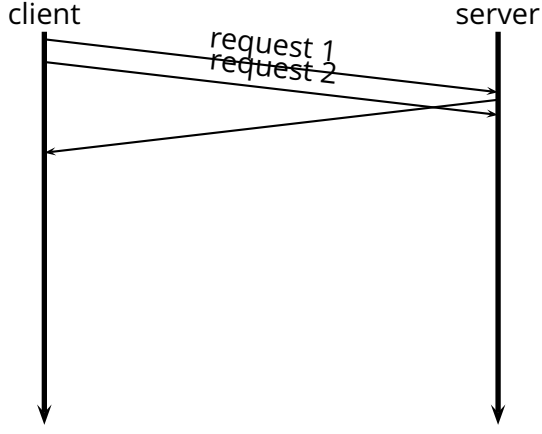
Persistent Connections With Pipelining

- A more efficient use of a connection is by *pipelining* requests



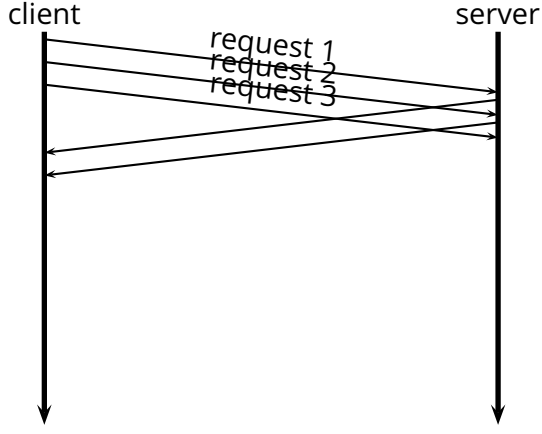
Persistent Connections With Pipelining

- A more efficient use of a connection is by *pipelining* requests



Persistent Connections With Pipelining

- A more efficient use of a connection is by *pipelining* requests



Persistent Connections With Pipelining

- A more efficient use of a connection is by *pipelining* requests