Outline

1 Computer Networks
   transmission media and network topologies
   client/server architecture
   layers, protocols, and sockets

2 Network Programming
   a simple client/server interaction
   the module socket in Python
   implementing a simple client/server

3 The SocketServer Module
   simplified development of network servers
   a server tells clients the time

MCS 260 Lecture 38
Introduction to Computer Science
Jan Verschelde, 21 November 2008
computer networks using sockets in Python

1. Computer Networks
   transmission media and network topologies
   client/server architecture
   layers, protocols, and sockets

2. Network Programming
   a simple client/server interaction
   the module socket in Python
   implementing a simple client/server

3. The SocketServer Module
   simplified development of network servers
   a server tells clients the time
Transmission Media

types of connections

The unit for transmission speed is \( bps \), bits per second.

Four categories of media used for data transmission:

1. **twisted pair wire**: pair of copper wires used for telephone communications. Transmission speeds vary from 2,400 to 33,600 \( bps \).

2. **coaxial cable**: for many telephone lines or television signal. Data transmission over short distances guarantees speeds up to \( 10^7 \) \( bps \).

3. **optical fibre**: signals transmitted via light-emitting diodes encode bits as presence or absence of light. Transmission speeds reach up to \( 10^9 \) \( bps \).

4. **electromagnetic waves**: wireless over short distances or via satellites across long distances.
Transmission Media

types of connections

The unit for transmission speed is \( \text{bps} \), bits per second.

Four categories of media used for data transmission:

1. **twisted pair wire**: pair of copper wires used for telephone communications. Transmission speeds vary from 2,400 to 33,600 \( \text{bps} \).

2. **coaxial cable**: for many telephone lines or television signal. Data transmission over short distances guarantees speeds up to \( 10^7 \text{ bps} \).

3. **optical fibre**: signals transmitted via light-emitting diodes encode bits as presence or absence of light. Transmission speeds reach up to \( 10^9 \text{ bps} \).

4. **electromagnetic waves**: wireless over short distances or via satellites across long distances.
Transmission Media

The unit for transmission speed is \( \text{bps} \), bits per second.

Four categories of media used for data transmission:

1. **twisted pair wire**: pair of copper wires used for telephone communications. Transmission speeds vary from 2,400 to 33,600 \( \text{bps} \).

2. **coaxial cable**: for many telephone lines or television signal. Data transmission over short distances guarantees speeds up to \( 10^7 \text{ bps} \).

3. **optical fibre**: signals transmitted via light-emitting diodes encode bits as presence or absence of light. Transmission speeds reach up to \( 10^9 \text{ bps} \).

4. **electromagnetic waves**: wireless over short distances or via satellites across long distances.
Transmission Media

The unit for transmission speed is \( \text{bps} \), bits per second.

Four categories of media used for data transmission:

1. **twisted pair wire**: pair of copper wires used for telephone communications. Transmission speeds vary from 2,400 to 33,600 bps.
2. **coaxial cable**: for many telephone lines or television signal. Data transmission over short distances guarantees speeds up to \( 10^7 \) bps.
3. **optical fibre**: signals transmitted via light-emitting diodes encode bits as presence or absence of light. Transmission speeds reach up to \( 10^9 \) bps.
4. **electromagnetic waves**: wireless over short distances or via satellites across long distances.
Transmission Media

types of connections

The unit for transmission speed is $\text{bps}$, bits per second.

Four categories of media used for data transmission:

1. **twisted pair wire**: pair of copper wires used for telephone communications. Transmission speeds vary from 2,400 to 33,600 $\text{bps}$.

2. **coaxial cable**: for many telephone lines or television signal. Data transmission over short distances guarantees speeds up to $10^7 \text{ bps}$.

3. **optical fibre**: signals transmitted via light-emitting diodes encode bits as presence or absence of light. Transmission speeds reach up to $10^9 \text{ bps}$.

4. **electromagnetic waves**: wireless over short distances or via satellites across long distances.
Network Topologies
structure of computer networks

Three most common regular topologies:

- a *star network*: one central node is connected to all other nodes

- a *ring network*: nodes form a closed circuit, messages circle around the ring of nodes

- a *bus network*: all nodes on a single bus, used in the Von Neumann architecture
Network Topologies
structure of computer networks

Three most common regular topologies:

- **a star network**: one central node is connected to all other nodes
- **a ring network**: nodes form a closed circuit, messages circle around the ring of nodes
- **a bus network**: all nodes on a single bus, used in the Von Neumann architecture
Network Topologies

structure of computer networks

Three most common regular topologies:

- a *star network*: one central node is connected to all other nodes

- a *ring network*: nodes form a closed circuit, messages circle around the ring of nodes

- a *bus network*: all nodes on a single bus, used in the Von Neumann architecture
Computer Networks

transmission media and network topologies
client/server architecture
layers, protocols, and sockets

Network Programming

a simple client/server interaction
the module socket in Python
implementing a simple client/server

The SocketServer Module

simplified development of network servers
a server tells clients the time
Client/Server Architecture
for software and hardware

Client/server architecture defines the communication between two computers: one is the server and the other acts as the client.

A client places a request or order to a server. The server processes the request.

The client does not need to know how the server processes the requests.

We distinguish between software and hardware client/server architectures:

- web and database servers offer software services;
- file and print servers offer hardware services.
Client/server architecture defines the communication between two computers: one is the server and the other acts as the client.

A client places a request or order to a server. The server processes the request.

The client does not need to know how the server processes the requests.

We distinguish between software and hardware client/server architectures:

• web and database servers offer software services;
• file and print servers offer hardware services.
Client/Server Architecture
for software and hardware

Client/server architecture defines the communication between two computers: one is the server and the other acts as the client.

A client places a request or order to a server. The server processes the request.

The client does not need to know how the server processes the requests.

We distinguish between software and hardware client/server architectures:

• web and database servers offer software services;
• file and print servers offer hardware services.
Client/Server Architecture
for software and hardware

Client/server architecture defines the communication between two computers: one is the server and the other acts as the client.

A client places a request or order to a server. The server processes the request.

The client does not need to know how the server processes the requests.

We distinguish between software and hardware client/server architectures:

- web and database servers offer software services;
- file and print servers offer hardware services.
Client/Server Architecture
for software and hardware

Client/server architecture defines the communication between two computers: one is the server and the other acts as the client.

A client places a request or order to a server. The server processes the request.

The client does not need to know how the server processes the requests.

We distinguish between software and hardware client/server architectures:

- web and database servers offer software services;
- file and print servers offer hardware services.
computer networks
using sockets in Python

1 Computer Networks
transmission media and network topologies
client/server architecture
layers, protocols, and sockets

2 Network Programming
a simple client/server interaction
the module socket in Python
implementing a simple client/server

3 The SocketServer Module
simplified development of network servers
a server tells clients the time
Four Layers in Internet Communication

application, transport, network, and link

1. The **application layer** consists of client and server software. The application prepares the message and defines the destination address.

2. The **transport layer** formats the message by chopping it into packets attaching a sequence number and destination address to each packet. When receiving, packets are collected and reassembled into a message.

3. The **network layer** determines the (intermediate) address for each packet. This layer detects when a packet has reached its final destination.

4. The **link layer** is responsible for the actual transmission of packets.
Four Layers in Internet Communication

application, transport, network, and link

1. The **application layer** consists of client and server software. The application prepares the message and defines the destination address.

2. The **transport layer** formats the message by chopping it into packets attaching a sequence number and destination address to each packet. When receiving, packets are collected and reassembled into a message.

3. The **network layer** determines the (intermediate) address for each packet. This layer detects when a packet has reached its final destination.

4. The **link layer** is responsible for the actual transmission of packets.
Four Layers in Internet Communication
application, transport, network, and link

1. The **application layer** consists of client and server software. The application prepares the message and defines the destination address.

2. The **transport layer** formats the message by chopping it into packets attaching a sequence number and destination address to each packet. When receiving, packets are collected and reassembled into a message.

3. The **network layer** determines the (intermediate) address for each packet. This layer detects when a packet has reached its final destination.

4. The **link layer** is responsible for the actual transmission of packets.
Four Layers in Internet Communication
application, transport, network, and link

1. The **application layer** consists of client and server software. The application prepares the message and defines the destination address.

2. The **transport layer** formats the message by chopping it into packets attaching a sequence number and destination address to each packet. When receiving, packets are collected and reassembled into a message.

3. The **network layer** determines the (intermediate) address for each packet. This layer detects when a packet has reached its final destination.

4. The **link layer** is responsible for the actual transmission of packets.
Following a Message
through all four layers

source

intermediate node

destination

Application

Transport

Network

Link

Application

Transport

Network

Link
Following a Message
through all four layers

Application
Transport
Network
Link
source

Application
Transport
Network
Link
intermediate node

Application
Transport
Network
Link
destination
Following a Message through all four layers

Diagram:

- Application
  - Transport
  - Network
    - Link
      - source
  - Link
    - intermediate node
  - Link
    - destination
Following a Message through all four layers

Diagram showing the flow of a message through the OSI model:

1. Application layer
2. Transport layer
3. Network layer
4. Link layer

Source (application) → intermediate node (network) → destination (network)
Following a Message
through all four layers

The SocketServer Module
simplified development of
network servers
a server tells clients the time
Following a Message through all four layers

```
+----------------+       +----------------+       +----------------+       +----------------+
| Application    |       | Transport      |       | Network        |       | Network        |
|                |       |                |       |                |       |                |
|                |       |                |       | source         |       |                |
|                |       |                |       | intermediate   |       |                |
|                |       |                |       | node           |       |                |
|                |       |                |       |                |       | destination    |
```
Following a Message through all four layers

Diagram:

Application -> Transport -> Network -> Link
source

Application -> Transport -> Network -> Link
intermediate node

Application -> Transport -> Network -> Link
destination
Following a Message through all four layers

Diagram:

- Application
  - Transport
  - Network
    - Link (source)
    - Intermediate node
    - Link
    - Network
    - Link (destination)

- Application
  - Transport
  - Network
  - Link
  - Destination
Following a Message
through all four layers
Following a Message
through all four layers

Diagram showing the flow of a message through the four layers of the OSI model.

Source
- Application
- Transport
- Network
- Link

Intermediate node
- Application
- Transport
- Network
- Link

Destination
- Application
- Transport
- Network
- Link
Network Protocols
TCP and UDP

Network protocols are rules for network communication.

We consider two types of protocols:

**TCP**  Transmission Control Protocol
First a message is sent that data is coming. Only after the receiver acknowledges this message will the sender send the data. All successful transmissions are confirmed, and retransmissions are acknowledged.

**UDP**  User Datagram Protocol
Unlike TCP, no connection is established prior to sending data. The sender just carries on after sending a message.

TCP is connection oriented, UDP is connectionless. TCP is more reliable whereas UDP is more streamlined.
Network Protocols

TCP and UDP

Network protocols are rules for network communication.

We consider two types of protocols:

**TCP**  
Transmission Control Protocol
First a message is sent that data is coming. Only after the receiver acknowledges this message will the sender send the data. All successful transmissions are confirmed, and retransmissions are acknowledged.

**UDP**  
User Datagram Protocol
Unlike TCP, no connection is established prior to sending data. The sender just carries on after sending a message.

TCP is connection oriented, UDP is connectionless. TCP is more reliable whereas UDP is more streamlined.
Network Protocols

TCP and UDP

Network protocols are rules for network communication.

We consider two types of protocols:

TCP  Transmission Control Protocol
First a message is sent that data is coming. Only after the receiver acknowledges this message will the sender send the data. All successful transmissions are confirmed, and retransmissions are acknowledged.

UDP  User Datagram Protocol
Unlike TCP, no connection is established prior to sending data. The sender just carries on after sending a message.

TCP is connection oriented, UDP is connectionless. TCP is more reliable whereas UDP is more streamlined.
Network protocols are rules for network communication.

We consider two types of protocols:

**TCP** Transmission Control Protocol
First a message is sent that data is coming. Only after the receiver acknowledges this message will the sender send the data. All successful transmissions are confirmed, and retransmissions are acknowledged.

**UDP** User Datagram Protocol
Unlike TCP, no connection is established prior to sending data. The sender just carries on after sending a message.

TCP is connection oriented, UDP is connectionless.
TCP is more reliable whereas UDP is more streamlined.
Network protocols are rules for network communication.

We consider two types of protocols:

**TCP** Transmission Control Protocol
First a message is sent that data is coming. Only after the receiver acknowledges this message will the sender send the data. All successful transmissions are confirmed, and retransmissions are acknowledged.

**UDP** User Datagram Protocol
Unlike TCP, no connection is established prior to sending data. The sender just carries on after sending a message.

TCP is connection oriented, UDP is connectionless. TCP is more reliable whereas UDP is more streamlined.
Network Protocols
TCP and UDP

Network protocols are rules for network communication.

We consider two types of protocols:

TCP  Transmission Control Protocol
First a message is sent that data is coming. Only after the receiver acknowledges this message will the sender send the data. All successful transmissions are confirmed, and retransmissions are acknowledged.

UDP  User Datagram Protocol
Unlike TCP, no connection is established prior to sending data. The sender just carries on after sending a message.

TCP is connection oriented, UDP is connectionless.
TCP is more reliable whereas UDP is more streamlined.
Network Protocols

TCP and UDP

Network protocols are rules for network communication. We consider two types of protocols:

TCP  Transmission Control Protocol
First a message is sent that data is coming. Only after the receiver acknowledges this message will the sender send the data. All successful transmissions are confirmed, and retransmissions are acknowledged.

UDP  User Datagram Protocol
Unlike TCP, no connection is established prior to sending data. The sender just carries on after sending a message.

TCP is connection oriented, UDP is connectionless. TCP is more reliable whereas UDP is more streamlined.
Sockets are objects programs use to connect.

Sockets were introduced in 1981 in BSD Unix. Originally used for communication between processes, i.e.: between two programs on same computer.

Sockets support communication across platforms, independent of the operating system.

In addition to the IP address of the computer, both server and client must use the same *port*. A port is a 16-bit integer, some are reserved for particular protocols. Any port between 1,024 and 65,535 is free.

Sockets support both TCP and UDP.
Sockets are objects programs use to connect.

Sockets were introduced in 1981 in BSD Unix. Originally used for communication between processes, i.e.: between two programs on same computer.

Sockets support communication across platforms, independent of the operating system.

In addition to the IP address of the computer, both server and client must use the same port. A port is a 16-bit integer, some are reserved for particular protocols. Any port between 1,024 and 65,535 is free.

Sockets support both TCP and UDP.
Sockets are objects programs use to connect.

Sockets were introduced in 1981 in BSD Unix. Originally used for communication between processes, i.e.: between two programs on same computer.

Sockets support communication across platforms, independent of the operating system.

In addition to the IP address of the computer, both server and client must use the same port. A port is a 16-bit integer, some are reserved for particular protocols. Any port between 1,024 and 65,535 is free.

Sockets support both TCP and UDP.
Sockets

Sockets are objects programs use to connect.

Sockets were introduced in 1981 in BSD Unix. Originally used for communication between processes, i.e.: between two programs on same computer.

Sockets support communication across platforms, independent of the operating system.

In addition to the IP address of the computer, both server and client must use the same port. A port is a 16-bit integer, some are reserved for particular protocols. Any port between 1,024 and 65,535 is free.

Sockets support both TCP and UDP.
Sockets are objects programs use to connect.

Sockets were introduced in 1981 in BSD Unix. Originally used for communication between processes, i.e.: between two programs on same computer.

Sockets support communication across platforms, independent of the operating system.

In addition to the IP address of the computer, both server and client must use the same *port*. A port is a 16-bit integer, some are reserved for particular protocols. Any port between 1,024 and 65,535 is free.

Sockets support both TCP and UDP.
Analogy with Telephone Exchange

client/server communication with sockets

Analogy between a telephone exchange and sockets:

1. dial company on 1-312-666-9000
   connect to IP address 127.0.0.1

2. call answered by reception
   connection established to remote host

3. ask for computer center
   route using specified port (8732)

4. call answered by computer center
   server handles request from client

5. hang up the phone
   close sockets
Analogy with Telephone Exchange

client/server communication with sockets

Analogy between a telephone exchange and sockets:

1. dial company on 1-312-666-9000
   connect to IP address 127.0.0.1

2. call answered by reception
   connection established to remote host

3. ask for computer center
   route using specified port (8732)

4. call answered by computer center
   server handles request from client

5. hang up the phone
   close sockets
Analogy with Telephone Exchange

client/server communication with sockets

Analogy between a telephone exchange and sockets:

1. dial company on 1-312-666-9000
   connect to IP address 127.0.0.1
2. call answered by reception
   connection established to remote host
3. ask for computer center
   route using specified port (8732)
4. call answered by computer center
   server handles request from client
5. hang up the phone
   close sockets
Analogy with Telephone Exchange

client/server communication with sockets

Analogy between a telephone exchange and sockets:

1. dial company on 1-312-666-9000
   connect to IP address 127.0.0.1
2. call answered by reception
   connection established to remote host
3. ask for computer center
   route using specified port (8732)
4. call answered by computer center
   server handles request from client
5. hang up the phone
   close sockets
Analogy with Telephone Exchange

client/server communication with sockets

Analogy between a telephone exchange and sockets:

1. dial company on 1-312-666-9000
   connect to IP address 127.0.0.1
2. call answered by reception
   connection established to remote host
3. ask for computer center
   route using specified port (8732)
4. call answered by computer center
   server handles request from client
5. hang up the phone
   close sockets
Computer Networks
transmission media and network topologies
client/server architecture
layers, protocols, and sockets

Network Programming
a simple client/server interaction
the module socket in Python
implementing a simple client/server

The SocketServer Module
simplified development of network servers
a server tells clients the time

1. Computer Networks
   transmission media and network topologies
   client/server architecture
   layers, protocols, and sockets

2. Network Programming
   a simple client/server interaction
   the module socket in Python
   implementing a simple client/server

3. The SocketServer Module
   simplified development of network servers
   a server tells clients the time
A simple Client/Server Interaction

Running `tcp_server.py`:

```
getafix:~ jan$ python tcp_server.py
server waits for connection
```

The client program `tcp_client.py` will run on another terminal.
A simple Client/Server Interaction

Running `tcp_server.py`:

```
getafix:~ jan$ python tcp_server.py
server waits for connection
```

The client program `tcp_client.py` will run on another terminal.
Connecting Client and Server

Running `tcp_server.py` on one terminal:

```
getafix:~ jan$ python tcp_server.py
server waits for connection
server accepted connection request from ('127.0.0.1', 49216)
server waits for data
```

Running `tcp_client.py` on another terminal:

```
getafix:~ jan$ python tcp_client.py
client is connected
Give message: 
```
Passing Data from Client to Server

Running `tcp_server.py` on one terminal:

```
getafix:~ jan$ python tcp_server.py
server waits for connection
server accepted connection request from ('127.0.0.1', 49216)
server waits for data
server received hello there!
```

Running `tcp_client.py` on another terminal:

```
getafix:~ jan$ python tcp_client.py
client is connected
Give message: hello there!
```
computer networks using sockets in Python

1. **Computer Networks**
   transmission media and network topologies
   client/server architecture
   layers, protocols, and sockets

2. **Network Programming**
   a simple client/server interaction
   the module socket in Python
   implementing a simple client/server

3. **The SocketServer Module**
   simplified development of network servers
   a server tells clients the time
Sockets in Python

the socket module

```
from socket import *
```

The socket module exports the method `socket()` which returns an object representing a socket.

The first argument of `socket` is the Address Family (AF):

- `AF_UNIX`: for UNIX sockets;
- `AF_INET`: most commonly used for internet

`AF_INET` supports both TCP and UDP, given respectively by `SOCK_STREAM` and `SOCK_DGRAM` as second argument of `socket()`. For example:

```
sock = socket(AF_INET, SOCK_STREAM)
```
from socket import *

The socket module exports the method `socket()` which returns an object representing a socket.

The first argument of `socket` is the Address Family (AF):

- **AF_UNIX**: for UNIX sockets;
- **AF_INET**: most commonly used for internet

`AF_INET` supports both TCP and UDP, given respectively by `SOCK_STREAM` and `SOCK_DGRAM` as second argument of `socket()`. For example:

```python
sock = socket(AF_INET, SOCK_STREAM)
```
Sockets in Python

the socket module

```python
from socket import *

The socket module exports the method `socket()` which returns an object representing a socket.

The first argument of `socket` is the Address Family (AF):

- `AF_UNIX`: for UNIX sockets;
- `AF_INET`: most commonly used for internet

`AF_INET` supports both TCP and UDP, given respectively by `SOCK_STREAM` and `SOCK_DGRAM` as second argument of `socket()`. For example:

```python
sock = socket(AF_INET, SOCK_STREAM)
```
from socket import *

The socket module exports the method `socket()` which returns an object representing a socket.

The first argument of `socket` is the Address Family (AF):

- **AF_UNIX**: for UNIX sockets;
- **AF_INET**: most commonly used for internet

`AF_INET` supports both TCP and UDP, given respectively by `SOCK_STREAM` and `SOCK_DGRAM` as second argument of `socket()`. For example:

```python
sock = socket(AF_INET, SOCK_STREAM)
```
Sockets in Python
the socket module

from socket import *

The socket module exports the method `socket()` which returns an object representing a socket.

The first argument of `socket` is the Address Family (AF):

- **AF_UNIX**: for UNIX sockets;
- **AF_INET**: most commonly used for internet

`AF_INET` supports both TCP and UDP, given respectively by `SOCK_STREAM` and `SOCK_DGRAM` as second argument of `socket()`. For example:

```python
sock = socket(AF_INET, SOCK_STREAM)
```
Sockets in Python

the socket module

from socket import *

The socket module exports the method `socket()` which returns an object representing a socket.

The first argument of `socket` is the Address Family (AF):

- `AF_UNIX` : for UNIX sockets;
- `AF_INET` : most commonly used for internet

`AF_INET` supports both TCP and UDP, given respectively by `SOCK_STREAM` and `SOCK_DGRAM` as second argument of `socket()`. For example:

```
sock = socket(AF_INET, SOCK_STREAM)
```
computer networks using sockets in Python

1. **Computer Networks**
   transmission media and network topologies
   client/server architecture
   layers, protocols, and sockets

2. **Network Programming**
   a simple client/server interaction
   the module socket in Python
   implementing a simple client/server

3. **The SocketServer Module**
   simplified development of network servers
   a server tells clients the time
Defining Connections
the methods bind and connect

After a socket is created, **both server and client** define
server_address = (hostname, number)

To bind the socket to the address, **the server** does
sock.bind(server_address)

and **the client** contacts the server then via
sock.connect(server_address)
Defining Connections
the methods bind and connect

After a socket is created, \textit{both server and client} define
\texttt{server\_address = (hostname, number)}

To bind the socket to the address, \textit{the server} does
\texttt{sock.bind(server\_address)}

and \textit{the client} contacts the server then via
\texttt{sock.connect(server\_address)}
Defining Connections
the methods bind and connect

After a socket is created, *both server and client* define

server_address = (hostname, number)

To bind the socket to the address, *the server* does

sock.bind(server_address)

and *the client* contacts the server then via

sock.connect(server_address)
Taking Requests
the methods listen and accept

With `listen()` the server indicates how many incoming connections will be accepted:

```python
sock.listen(2) # accept at most 2 connections
```

The server takes requests via the `accept()` method:

```python
client, client_address = sock.accept()
```

The `accept()` method returns:

1. a socket `client` for receiving data;
2. the address of the client.
Taking Requests
the methods listen and accept

With `listen()` the server indicates how many incoming connections will be accepted:

```python
sock.listen(2)  # accept at most 2 connections
```

The server takes requests via the `accept()` method:

```python
client, client_address = sock.accept()
```

The `accept()` method returns

1. a socket `client` for receiving data;
2. the address of the client.
Taking Requests
the methods listen and accept

With `listen()` the server indicates how many incoming connections will be accepted:
```
sock.listen(2)  # accept at most 2 connections
```

The server takes requests via the `accept()` method:
```
client, client_address = sock.accept()
```

The `accept()` method returns
1. a socket `client` for receiving data;
2. the address of the client.
Taking Requests
the methods listen and accept

With `listen()` the server indicates how many incoming connections will be accepted:

```python
sock.listen(2) # accept at most 2 connections
```

The server takes requests via the `accept()` method:

```python
client, client_address = sock.accept()
```

The `accept()` method returns

1. a socket `client` for receiving data;
2. the address of the client.
The **client** sends data with `send()`

```python
sock.send(data)
```

The **server** receives data applying `recv()`

```python
data = client.recv(buffer)
```

When all is over, *both client and server* do

```python
sock.close()
```
Sending and Receiving Data
the methods send and recv

The **client** sends data with `send()`

```
sock.send(data)
```

The **server** receives data applying `recv()`

```
data = client.recv(buffer)
```

to the socket **client** obtained with `accept()`.

When all is over, **both client and server** do

```
sock.close()
```
Sending and Receiving Data
the methods send and recv

The **client** sends data with `send()`

```python
sock.send(data)
```

The **server** receives data applying `recv()`

```python
data = client.recv(buffer)
```

to the socket **client** obtained with `accept()`.

When all is over, **both client and server** do

```python
sock.close()
```
The program tcp_server.py

```python
from socket import *

hostname = ''  # blank so any address can be used
number = 11267  # number for the port
buffer = 80     # size of the buffer

server_address = (hostname, number)
server = socket(AF_INET, SOCK_STREAM)
server.bind(server_address)
server.listen(2)

print 'server waits for connection'
client, client_address = server.accept()
print 'server accepted connection request from ', client_address
print 'server waits for data'
data = client.recv(buffer)
print 'server received ', data
server.close()
```
the program tcp_server.py

```python
from socket import *

hostname = ''  # blank so any address can be used
number = 11267  # number for the port
buffer = 80     # size of the buffer

server_address = (hostname, number)
server = socket(AF_INET, SOCK_STREAM)
server.bind(server_address)
server.listen(2)

print 'server waits for connection'
client, client_address = server.accept()
print 'server accepted connection request from ', client_address
print 'server waits for data'
data = client.recv(buffer)
print 'server received ', data
server.close()
```
the program tcp_server.py

from socket import *

hostname = ''    # blank so any address can be used
number = 11267   # number for the port
buffer = 80      # size of the buffer

server_address = (hostname, number)
server = socket(AF_INET, SOCK_STREAM)
sERVER.bind(server_address)
sERVER.listen(2)

print 'server waits for connection'
client, client_address = server.accept()
print 'server accepted connection request from ',
     client_address
print 'server waits for data'
data = client.recv(buffer)
print 'server received ', data
server.close()
from socket import *

hostname = '' # blank so any address can be used
number = 11267 # number for the port
buffer = 80  # size of the buffer

server_address = (hostname, number)
s
server = socket(AF_INET, SOCK_STREAM)
server.bind(server_address)
s
server.listen(2)

print 'server waits for connection'
client, client_address = server.accept()
print 'server accepted connection request from ',\
     client_address

print 'server waits for data'
data = client.recv(buffer)
print 'server received ', data

server.close()
the program tcp_server.py

```python
from socket import *

hostname = ''  # blank so any address can be used
number = 11267  # number for the port
buffer = 80    # size of the buffer

server_address = (hostname, number)
s server = socket(AF_INET, SOCK_STREAM)
server.bind(server_address)
s server.listen(2)

print 'server waits for connection'
client, client_address = server.accept()
print 'server accepted connection request from ',
     client_address

print 'server waits for data'
data = client.recv(buffer)
print 'server received ', data

server.close()
```
the program tcp_server.py

```python
from socket import *

hostname = ''  # blank so any address can be used
number = 11267  # number for the port
buffer = 80     # size of the buffer

server_address = (hostname, number)
server = socket(AF_INET, SOCK_STREAM)
sERVER.bind(server_address)
sERVER.listen(2)

print 'server waits for connection'
client, client_address = server.accept()
print 'server accepted connection request from ', client_address
print 'server waits for data'
data = client.recv(buffer)
print 'server received ', data
server.close()
```
the program tcp_client.py

```python
from socket import *

hostname = 'localhost'  # on same host
number = 11267           # same port number
buffer = 80              # size of the buffer

server_address = (hostname, number)
client = socket(AF_INET, SOCK_STREAM)
client.connect(server_address)

print 'client is connected'
data = raw_input('Give message : ')
client.send(data)

client.close()
```
the program tcp_client.py

```python
from socket import *

hostname = 'localhost'       # on same host
number = 11267              # same port number
buffer = 80                 # size of the buffer

server_address = (hostname, number)
c = socket(AF_INET, SOCK_STREAM)
c.connect(server_address)

print 'client is connected'
data = raw_input('Give message : ')
c.send(data)

c.close()
```
the program tcp_client.py

from socket import *

hostname = 'localhost' # on same host
number = 11267 # same port number
buffer = 80 # size of the buffer

server_address = (hostname, number)
client = socket(AF_INET, SOCK_STREAM)
client.connect(server_address)

print 'client is connected'
data = raw_input('Give message : ')
client.send(data)

client.close()
the program tcp_client.py

```python
from socket import *

hostname = 'localhost'  # on same host
number = 11267  # same port number
buffer = 80  # size of the buffer

server_address = (hostname, number)
client = socket(AF_INET, SOCK_STREAM)
client.connect(server_address)

print 'client is connected'
data = raw_input('Give message : ')
client.send(data)

client.close()
```
from socket import *

hostname = 'localhost'    # on same host
number = 11267           # same port number
buffer = 80               # size of the buffer

server_address = (hostname, number)
client = socket(AF_INET, SOCK_STREAM)
client.connect(server_address)

print 'client is connected'
data = raw_input('Give message : ')
client.send(data)

client.close()
computer networks using sockets in Python

1 Computer Networks
transmission media and network topologies
client/server architecture
layers, protocols, and sockets

2 Network Programming
a simple client/server interaction
the module socket in Python
implementing a simple client/server

3 The SocketServer Module
simplified development of network servers
a server tells clients the time
The SocketServer Module
simplified development of network servers

With the `SocketServer` module we do not need to import the `socket` module for the server script.

Follow these steps:

1. `from SocketServer import StreamRequestHandler, TCPServer`
2. Inheriting from `StreamRequestHandler` define a request handler class. Override `handle()`.
   → `handle()` processes incoming requests
3. Instantiate `TCPServer` with `(address, port)` and an instance of the request handler class.
   → this returns a server object
4. Apply the method `handle_request()` or `serve_forever()` to the server object.
The SocketServer Module
simplified development of network servers

With the **SocketServer** module we do not need to import the **socket** module for the server script.

Follow these steps:

1. from `SocketServer` import `StreamRequestHandler, TCPServer`

2. Inheriting from `StreamRequestHandler` define a request handler class. Override `handle()`. → `handle()` processes incoming requests

3. Instantiate `TCPServer` with `(address, port)` and an instance of the request handler class. → this returns a server object

4. Apply the method `handle_request()` or `serve_forever()` to the server object.
The SocketServer Module
simplified development of network servers

With the SocketServer module we do not need to import the socket module for the server script.

Follow these steps:

1. from SocketServer import StreamRequestHandler, TCPServer
2. Inheriting from StreamRequestHandler define a request handler class. Override handle(). → handle() processes incoming requests
3. Instantiate TCPServer with (address, port) and an instance of the request handler class. → this returns a server object
4. Apply the method handle_request() or serve_forever() to the server object.
The SocketServer Module
simplified development of network servers

With the `SocketServer` module we do not need to import the `socket` module for the server script.

Follow these steps:

1. `from SocketServer import StreamRequestHandler, TCPServer`
2. Inheriting from `StreamRequestHandler` define a request handler class. Override `handle()`.
   → `handle()` processes incoming requests
3. Instantiate `TCPServer` with `(address, port)` and an instance of the request handler class.
   → this returns a server object
4. Apply the method `handle_request()` or `serve_forever()` to the server object.
The SocketServer Module
simplified development of network servers

With the `SocketServer` module we do not need to import the `socket` module for the server script.

Follow these steps:

1. ```python
   from SocketServer import StreamRequestHandler, TCPServer
   ```

2. **Inheriting from** `StreamRequestHandler`
   define a request handler class. **Override** `handle()`.
   → `handle()` processes incoming requests

3. **Instantiate** `TCPServer` **with** `(address, port)`
   and an instance of the request handler class.
   → this returns a server object

4. **Apply the method** `handle_request()` **or**
   `serve_forever()` **to the server object.**
Computer Networks
transmission media and network topologies
client/server architecture
layers, protocols, and sockets

Network Programming
a simple client/server interaction
the module socket in Python
implementing a simple client/server

The SocketServer Module
simplified development of network servers
a server tells clients the time
A Server to tell the Time
an example of using SocketServer

In the window running the server:

$ python clockserver.py
server is listening to 12091
connected at ('127.0.0.1', 49160)
read "What is the time? " from client
writing "Sun Mar 30 20:23:41 2008" to client

In the window running the client:

$ python clockclient.py
client is connected
A Server to tell the Time
an example of using SocketServer

In the window running the server:

```
$ python clockserver.py
server is listening to 12091
connected at ('127.0.0.1', 49160)
read "What is the time? " from client
writing "Sun Mar 30 20:23:41 2008" to client
```

In the window running the client:

```
$ python clockclient.py
client is connected
```
A Server to tell the Time
an example of using SocketServer

In the window running the server:

$ python clockserver.py
server is listening to 12091
connected at ('127.0.0.1', 49160)
read "What is the time? " from client
writing "Sun Mar 30 20:23:41 2008" to client

In the window running the client:

$ python clockclient.py
client is connected
A Server to tell the Time
an example of using SocketServer

In the window running the server:

```
$ python clockserver.py
server is listening to 12091
connected at ('127.0.0.1', 49160)
read "What is the time? " from client
writing "Sun Mar 30 20:23:41 2008" to client
```

In the window running the client:

```
$ python clockclient.py
client is connected
```
Code for the Client

in file clockclient.py

```python
from socket import *

hostname = 'localhost'  # on same host
number = 12091          # same port number
buffer = 25             # size of the buffer

server_address = (hostname, number)
client = socket(AF_INET, SOCK_STREAM)
client.connect(server_address)

print 'client is connected'
data = 'What is the time?'
client.send(data + (buffer-len(data))*' ')
data = client.recv(buffer)
print data

client.close()
```
from socket import *

hostname = 'localhost'    # on same host
number = 12091             # same port number
buffer = 25               # size of the buffer

server_address = (hostname, number)
client = socket(AF_INET, SOCK_STREAM)
client.connect(server_address)

print 'client is connected'
data = 'What is the time?'
client.send(data + (buffer-len(data))*' ')
data = client.recv(buffer)
print data

client.close()
from socket import *

hostname = 'localhost'  # on same host
number = 12091          # same port number
buffer = 25             # size of the buffer

server_address = (hostname, number)
client = socket(AF_INET, SOCK_STREAM)
client.connect(server_address)

print 'client is connected'
data = 'What is the time?'
client.send(data + (buffer-len(data))*' ')  
data = client.recv(buffer)
print data

client.close()
Code for the server
in the file clockserver.py

from SocketServer import StreamRequestHandler
from SocketServer import TCPServer
from time import ctime

port = 12091

class ServerClock(StreamRequestHandler):
    ""
    The server tells the clients the time.
    ""
    def handle(self):
        ""
        Handler sends time to client.
        ""
        ...

ss = TCPServer(('', port), ServerClock)
print 'server is listening to', port
ss.handle_request()
from SocketServer import StreamRequestHandler
from SocketServer import TCPServer
from time import ctime

port = 12091

class ServerClock(StreamRequestHandler):
    """
    The server tells the clients the time.
    """
    def handle(self):
        """
        Handler sends time to client.
        """
        ...

ss = TCPServer(("", port), ServerClock)
print 'server is listening to', port
ss.handle_request()
Code for the server

in the file clockserver.py

from SocketServer import StreamRequestHandler
from SocketServer import TCPServer
from time import ctime

port = 12091

class ServerClock(StreamRequestHandler):
    
    """
    The server tells the clients the time.
    """

def handle(self):
    
        """
        Handler sends time to client.
        """

    ...

ss = TCPServer(('', port), ServerClock)
print 'server is listening to', port
ss.handle_request()
Code for the Handler

```python
def handle(self):
    """
    Handler sends time to client.
    """
    print "connected at", self.client_address
    data = self.rfile.read(25)
    print 'read \"' + data + '\" from client'
    now = ctime()
    print 'writing \"' + now + '\" to client'
    self.wfile.write(now)
```
def handle(self):
    """
    Handler sends time to client.
    """
    print "connected at", self.client_address
    data = self.rfile.read(25)
    print 'read "' + data + '" from client'
    now = ctime()
    print 'writing "' + now + '" to client'
    self.wfile.write(now)
Code for the Handler

def handle(self):
    
    """
    Handler sends time to client.
    """

    print "connected at", self.client_address
    data = self.rfile.read(25)
    print 'read "' + data + '" from client'
    now = ctime()
    print 'writing "' + now + '" to client'
    self.wfile.write(now)
About rfile and wfile attributes in the class StreamRequestHandler

- **rfile** contains input stream to read data from client
  
  example: `data = self.rfile.read(25)`
  
  **client must send exactly 25 characters**!

- **wfile** contains output stream to write data to client
  
  example: `self.wfile.write(data)`
  
  **all data are strings of characters!**
About `rfile` and `wfile` attributes in the class `StreamRequestHandler`

- **`rfile`** contains input stream to read data from client
  
  **example:** `data = self.rfile.read(25)`

  
  `client must send exactly 25 characters!`

- **`wfile`** contains output stream to write data to client
  
  **example:** `self.wfile.write(data)`

  
  `all data are strings of characters!`
About rfile and wfile attributes in the class StreamRequestHandler

- **rfile** contains input stream to read data from client
  - example: `data = self.rfile.read(25)`
  - *client must send exactly 25 characters!*
- **wfile** contains output stream to write data to client
  - example: `self.wfile.write(data)`
  - *all data are strings of characters!*

Network Programming

a simple client/server interaction
the module socket in Python
implementing a simple client/server

The SocketServer Module

simplified development of network servers
a server tells clients the time
About `rfile` and `wfile` attributes in the class `StreamRequestHandler`

- **rfile** contains input stream to read data from client
  
  example: ```
  data = self.rfile.read(25)
  ```

  *client must send exactly 25 characters!*

- **wfile** contains output stream to write data to client
  
  example: ```
  self.wfile.write(data)
  ```

  *all data are strings of characters!*
Alternatives to Simple Example

Instead of `StreamRequestHandler`, we can use `DatagramRequestHandler`.

Instead of `TCPServer`, we can use `UDPServer`, if we want UDP instead of TCP protocol.

On Unix (instead of `TCPServer`): `UnixStreamServer` or `UnixDatagramServer`.

Choice between

1. `handle_request()`: handle one single request, or
2. `serve_forever()`: indefinitely many requests.
Alternatives to Simple Example

Instead of `StreamRequestHandler`, we can use `DatagramRequestHandler`.

Instead of `TCPServer`, we can use `UDPServer`, if we want UDP instead of TCP protocol.

On Unix (instead of `TCPServer`): `UnixStreamServer` or `UnixDatagramServer`.

Choice between

1. `handle_request()`: handle one single request, or
2. `serve_forever()` : indefinitely many requests.
Alternatives to Simple Example

Instead of `StreamRequestHandler`, we can use `DatagramRequestHandler`.

Instead of `TCPServer`, we can use `UDPServer`, if we want UDP instead of TCP protocol.

On Unix (instead of `TCPServer`): `UnixStreamServer` or `UnixDatagramServer`.

Choice between

1. `handle_request()`: handle one single request, or
2. `serve_forever()`: indefinitely many requests.
Alternatives to Simple Example

Instead of StreamRequestHandler, we can use DatagramRequestHandler.

Instead of TCPServer, we can use UDPServer, if we want UDP instead of TCP protocol.

On Unix (instead of TCPServer): UnixStreamServer or UnixDatagramServer.

Choice between

1. handle_request(): handle one single request, or
2. serve_forever(): indefinitely many requests.
Using `serve_forever()`

With `serve_forever()`, we can

1. serve indefinitely many requests,
2. simultaneously from multiple clients.

```python
ss = TCPServer(('',port),ServerClock)
print 'server is listening to', port
try:
    print 'press ctrl c to stop server'
    ss.serve_forever()
except KeyboardInterrupt:
    print ' ctrl c pressed, closing server'
ss.socket.close()
```
Using `serve_forever()`

With `serve_forever()`, we can

1. serve indefinitely many requests,
2. simultaneously from multiple clients.

```python
ss = TCPServer(('',port),ServerClock)
print 'server is listening to', port
try:
    print 'press ctrl c to stop server'
    ss.serve_forever()
except KeyboardInterrupt:
    print 'ctrl c pressed, closing server'
ss.socket.close()
```
Using `serve_forever()`

With `serve_forever()`, we can

1. serve indefinitely many requests,
2. simultaneously from multiple clients.

```python
ss = TCPServer(('',port),ServerClock)
print 'server is listening to', port
try:
    print 'press ctrl c to stop server'
    ss.serve_forever()
except KeyboardInterrupt:
    print ' ctrl c pressed, closing server'
ss.socket.close()
```
Using `serve_forever()`

With `serve_forever()`, we can

1. serve indefinitely many requests,
2. simultaneously from multiple clients.

```python
ss = TCPServer(('',port),ServerClock)
print 'server is listening to', port
try:
    print 'press ctrl c to stop server'
    ss.serve_forever()
except KeyboardInterrupt:
    print ' ctrl c pressed, closing server'
ss.socket.close()
```
Using `serve_forever()`

With `serve_forever()`, we can

1. serve indefinitely many requests,
2. simultaneously from multiple clients.

```python
ss = TCPServer(('',port),ServerClock)
print 'server is listening to', port
try:
    print 'press ctrl c to stop server'
    ss.serve_forever()
except KeyboardInterrupt:
    print 'ctrl c pressed, closing server'
ss.socket.close()
```
Summary + Assignments

Background: §4.1,4 in *Computer Science, an overview.*

Assignments:

1. What is Twisted? What can it do for you?

2. Extend the client/server interaction to simulate a password dialogue. After receiving data from a client, the server returns *access granted* or *access denied* depending on whether the received data matches the password.

3. Describe how you would implement the game of rock, paper, and scissors in a client/server manner.

4. How would you implement a blackjack game using a client/server protocol?

Final exam on Monday 8 December from 1 to 3PM in LC A4.