School of Computer Science

University of St.Gallen

"From insight to impact"

EQUIS ACCREDITED
AACSB ACCREDITED
ASSOCIATION OF MBAS ACCREDITED

Fundamentals and Methods of Computer Science for Business Studies

Week 6: Guided Exercise

Prof. Dr. Stephan Aier
Titularprofessor
stephan.aier@unisg.ch

Prof. Dr. Simon Mayer
Professor für Informatik
simon.mayer@unisg.ch

Prof. Dr. Siegfried Handschuh
Professor für Informatik
siegfried.handschuh@unisg.ch

First: The missing Demo from Yesterday

Remember: HTTP is based on **text messages**

That means that we can “manually” chat with Web servers

Prerequisites for this demo

- A server that is ready to chat in http on port 80 (e.g., example.org)
- The telnet program on your machine (Windows: “Activate Windows Features” -> check “Telnet”)

```
telnet example.org 80
```

- Now, everything you type into the console is sent directly to the server (over TCP)
- To see what you are typing, activate “localecho” in telnet (CTRL+], type “set localecho”, <Enter>, <Enter>

This Week’s Menu

Lecture

Motivation
Computer Networks Basics
The Networking Stack
The World Wide Web

Guided Exercise

Briefing: Security
Stock Exchange
Overview: Food-and-Walls

Warm-up: Differential Privacy Primer

Imagine that we're performing **queries** over a database that holds **personal user data**

How much any individual's data contributes to the result of a database query depends on…

1. How much this individual's **local data influences the result** (e.g., if they are an outlier)
2. How **many other individuals**' data sets are involved in the database query

1 person in the data -> contribution 100% (privacy = 0%)

100 persons in the database -> contribution 1% (privacy > 0%) (if the influence of each individual on the result is similar)

How can we, if queries are **made on the data of fewer and fewer persons**, still produce the **same amount of privacy?**

We can **add noise** to the query result! ☺

Differential Privacy Example: Randomized Response

What's the incidence *p* of an attribute in a given population?

"Have you ever taken performance-enhancing drugs in an academic context?"

Procedure

1. Toss a coin
   a. You're "**Heads**": Toss the coin again. Ignore the outcome. **Answer honestly.**
   b. You're **"Tails"**: Toss the coin again. Keep the outcome private. **Answer "Yes" if heads, "No" if tails.**

We get a fraction of "yes" responses that we call *q*

- Some of these "yes" responses are given by persons who have never taken such drugs
- All other "yes" responses are given by persons who actually have taken them
- We don't know who is who…

How many?

To get a 'false' yes, a person needs two 'tails' events.

The true incidence *p* is therefore approximated by

$\mathbf{q} = 0.25 * (1 - p) + 0.75 * p = \mathbf{0.25 + 0.5 * p}$

$\mathbf{p = 2 * q - 0.5}$

Differential Privacy Primer

https://www.youtube.com/watch?v=gI0wk1CXlsQ

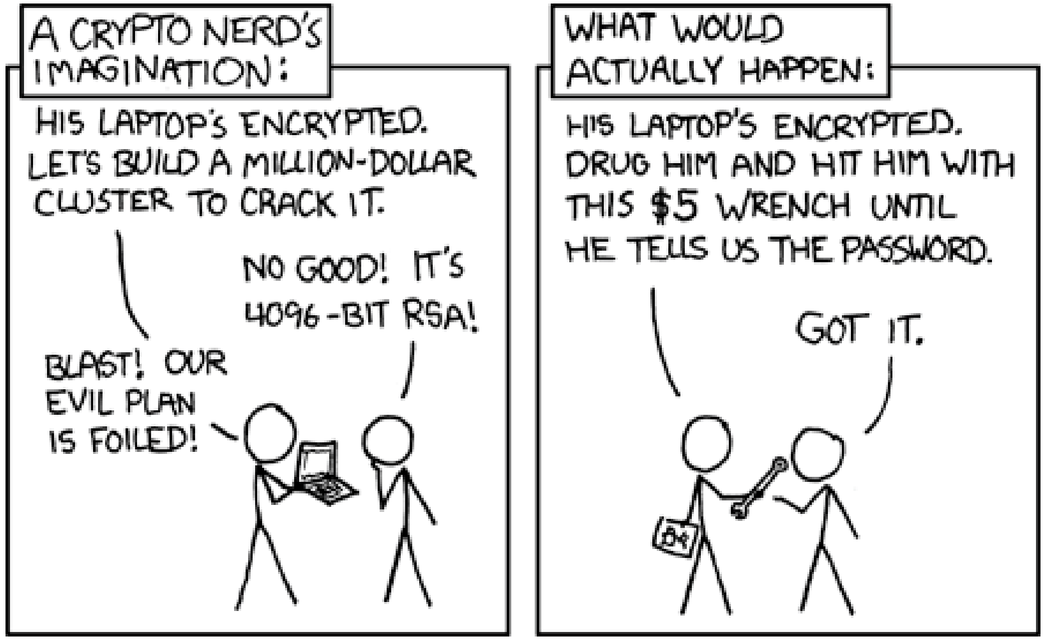
“It allows companies to collect information about their users without compromising the privacy of the individual.”

Is differential privacy really a “free lunch”?

No!

- Concrete example: what if the only drug-taker in the room gets two ‘tails’?
- But differential privacy allows us to **tune the precision/privacy trade-off** in a system ☺

HTTPS and TLS


A CRYPTO NERD'S IMAGINATION:
HIS LAPTOP'S ENCRYPTED. LET'S BUILD A MILLION-DOLLAR CLUSTER TO CRACK IT.
NO GOOD! IT'S 4096-BIT RSA!
BLAST! OUR EVIL PLAN IS FOILED!
WHAT WOULD ACTUALLY HAPPEN:
HIS LAPTOP'S ENCRYPTED. DRUG HIM AND HIT HIM WITH THIS $5 WRENCH UNTIL HE TELLS US THE PASSWORD.
GOT IT.

TLS (Transport Layer Security) is a cryptographic protocol that provides **data privacy** and **data integrity** as well as **authentication** between computer applications

- TLS runs "on top of" a reliable transport (e.g., TCP), so it's treated as a **transport-layer protocol**
- Applications must actively initiate TLS, so it's sometimes classified as an **application-layer protocol**

Website protocol support (Apr. 2021)

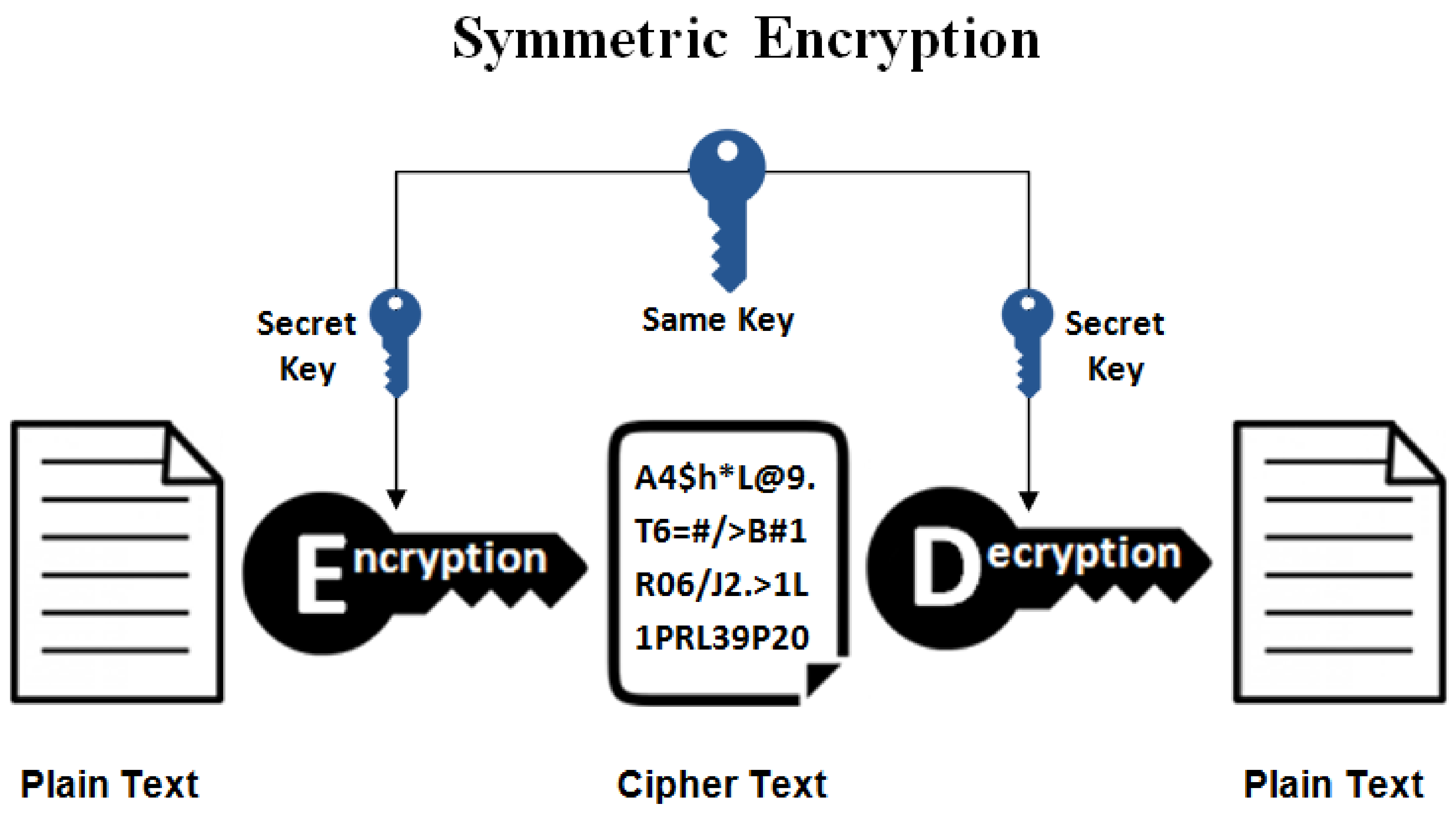
Protocol version	Website support[67]	Security[67][68]
SSL 2.0	0.5%	Insecure
SSL 3.0	3.4%	Insecure[69]
TLS 1.0	46.9%	Deprecated[8][9][10]
TLS 1.1	52.3%	Deprecated[8][9][10]
TLS 1.2	99.4%	Depends on cipher[n 1] and client mitigations[n 2]
TLS 1.3	44.8%	Secure

https://en.wikipedia.org/wiki/Transport_Layer_Security (in 2015)

University of St.Gallen

Primer on Encryption

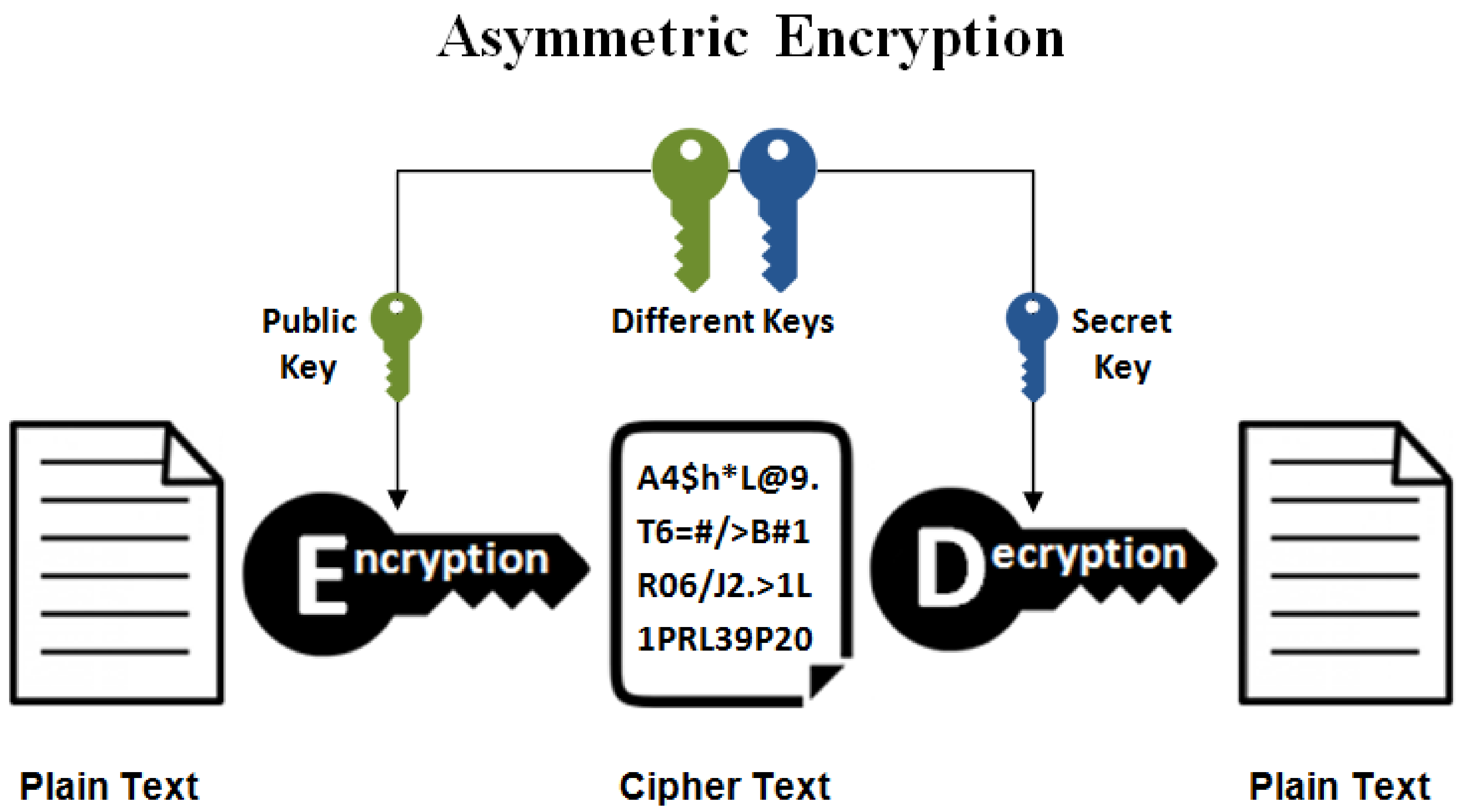
Advantage: Fast! Simple!
Disadvantage: Requires shared key!
Well-known Algorithms: AES, DES



https://www.ssl2buy.com/wiki/symmetric-vs-asymmetric-encryption-what-are-differences

Primer on Encryption

Advantages: No shared secret! Usually also enables digital signatures!
Disadvantages: Slow! How to know that a public key is authentic?
Well-known Algorithms: RSA, DSA, Elliptic curve techniques



https://www.ssl2buy.com/wiki/symmetric-vs-asymmetric-encryption-what-are-differences

«Magic»: How to Establish Shared Secrets?

How to establish a shared secret between two parties over a public channel?

https://www.youtube.com/watch?v=YEBfamv-_do from 2:08

And now, wouldn’t it be great if we could…

…use this **slow** method to **derive a shared secret**

…and then use this shared secret for **fast symmetric encryption**

https://networklessons.com/uncategorized/diffie-hellman-key-exchange-explained/

TLS Features

Here is (quality-controlled) background material about this:
https://en.wikipedia.org/wiki/Public_key_infrastructure

Setup

- The communicating parties use **symmetric encryption** to encrypt the transmitted data
- The **encryption keys** are **generated from a shared secret** and are unique for each connection
- The shared secret, encryption algorithm, and keys are **negotiated** at the start of each session (“TLS Handshake”), based on protocols such as DH
- Each entity has a **private and a public key**; these are mathematically coupled to get the following properties
- Each entity **advertises its public key** and **keeps the private key private**

Main Feature #1: Secure Connections

- The encrypting entity **encrypts messages** with the **receiver’s public key**
- The receiver **decrypts messages** with its **own private key**

Main Feature #2: Authenticated Identities (usually: optional for the client but required for the server)

- The authenticating entity **signs messages** with its **private** key
- The authenticating entity also **advertises** its **public** key

Do you see any problem here?

- How do we know that the advertised public keys are correct?
- If we don’t, so-called **machine-in-the-middle attacks** become feasible!
- Solution: The authenticity of the public keys are certified by a trusted third party called a **certification authority**

Network Security Basics

HTTPS and TLS

Data Unit	Layer
data	application
segments	transport
packets	network
frames	data link
bits	physical

HTTPS ("HTTP-over-TLS") = Security on the **transport layer!**

– The entire request **above transport is encrypted** but nothing at or below the transport layer

So what does that mean **in practice**? When I send an HTTPS request, …

– …can an eavesdropper **modify the request** sent over HTTPS?
– …can he see the **request contents**?
– …can he see the **request URL**?
– …can he see the **host IP address**
– …can he see the **host port number**?
– …can he see the **HTTP request method**?
– …can he see the **amount of data** transferred?
– …can he see the **duration** of the interaction with the host?

Why?

Yes!

Yes!

Yes!

Yes!

Any Questions / Comments / Doubts / Concerns?

This Week’s Menu

Lecture

Motivation
Computer Networks Basics
The Networking Stack
The World Wide Web

Guided Exercise

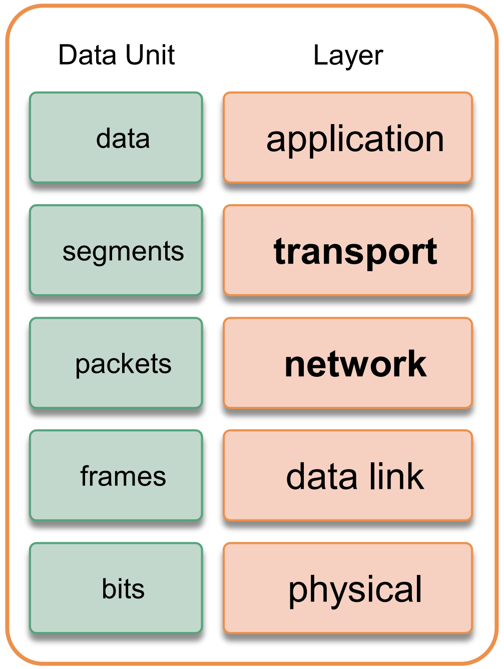
Briefing: Security
Stock Exchange
Overview: Food-and-Walls

Stock Exchange: A Client/Server Application

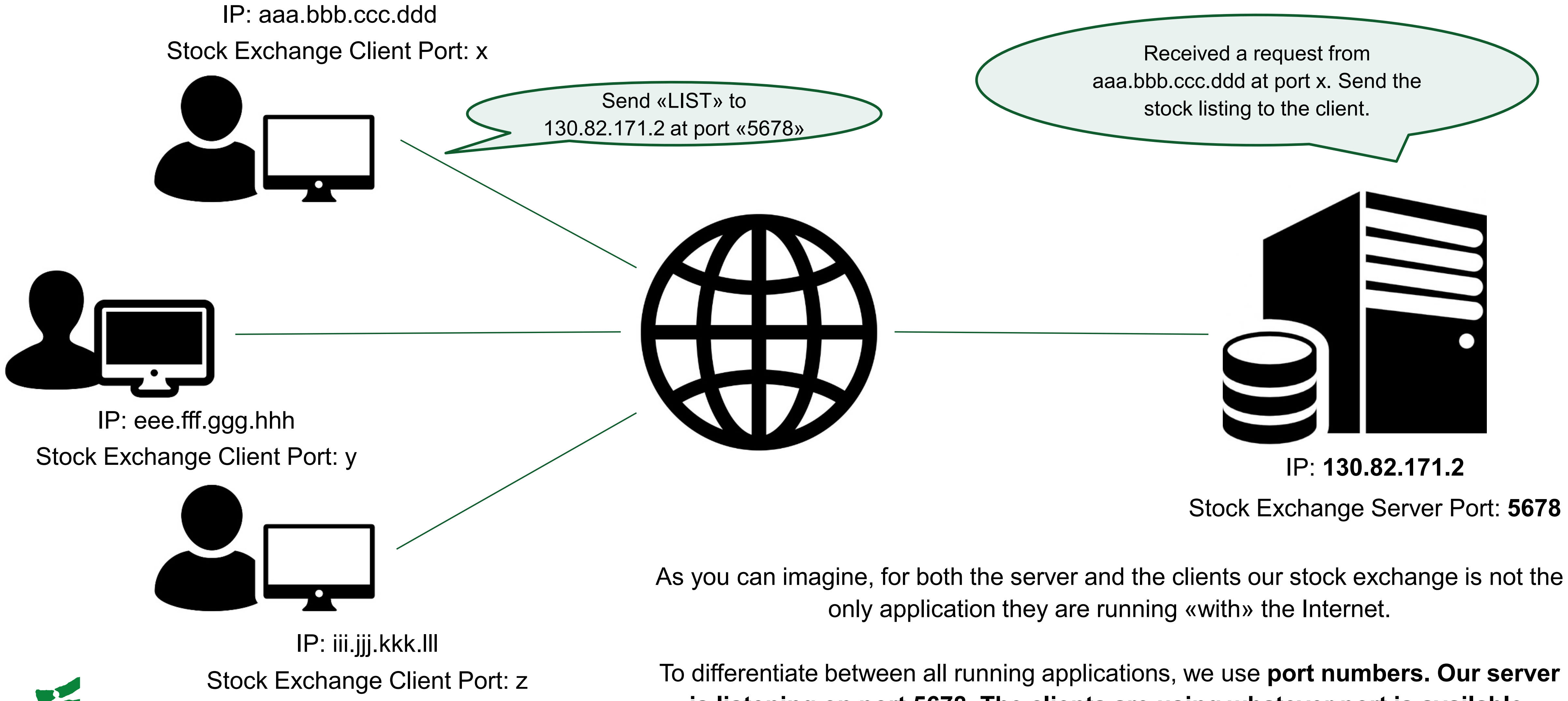
We'll review a **simple stock market server application** and run it on a **publicly reachable server**
You should then be able to run **clients** for the stock market server and interact with it.

Our stock market server communicates with clients via the User Datagram Protocol.
UDP is a very simple **transport-layer** protocol that specifies the ports that two endpoints communicate on.
We use UDP on top of the Internet Protocol (at the **network layer**)
We create the **application-layer** protocol ourselves (i.e., it's 'custom' or 'proprietary').

Our system is **agnostic** of which protocols are used on the lower layer ☺
- It will work through Ethernet cables, fibre-optic cables, and Wi-Fi links
- It will work over mobile (3G, 4G, 5G, etc.) connections
- It would even work with sending physical letters (but with high delays)
- Or by using pigeon carriers (Avian Carriers, https://tools.ietf.org/html/rfc1149)



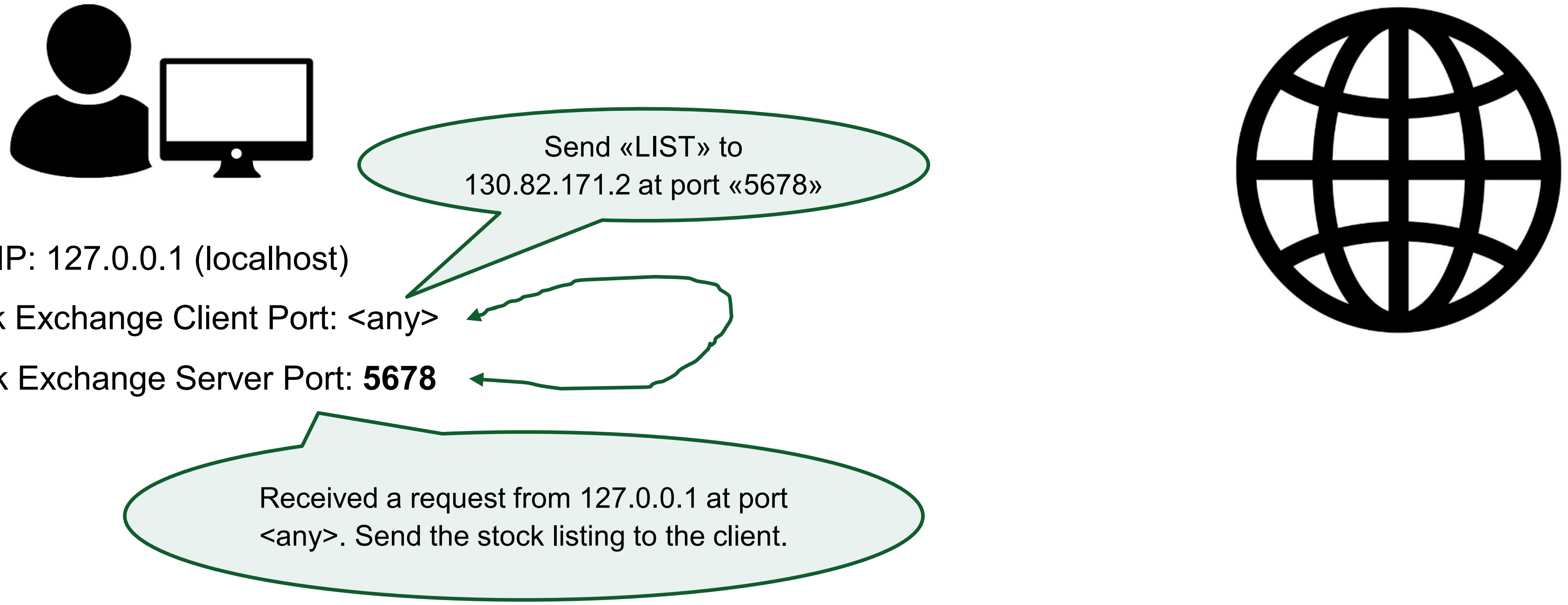
Stock Exchange: A Client/Server Application (Production)


IP: aaa.bbb.ccc.ddd
Stock Exchange Client Port: x
Send «LIST» to 130.82.171.2 at port «5678»
Received a request from aaa.bbb.ccc.ddd at port x. Send the stock listing to the client.
IP: eee.fff.ggg.hhh
Stock Exchange Client Port: y
IP: iii.jjj.kkk.lll
Stock Exchange Client Port: z
IP: **130.82.171.2**
Stock Exchange Server Port: **5678**

As you can imagine, for both the server and the clients our stock exchange is not the only application they are running «with» the Internet.

To differentiate between all running applications, we use port numbers. **Our server is listening on port 5678. The clients are using whatever port is available.**

Stock Exchange: A Client/Server Application (Testing)


Send «LIST» to
130.82.171.2 at port «5678»
IP: 127.0.0.1 (localhost)
Stock Exchange Client Port: <any>
Stock Exchange Server Port: **5678**
Received a request from 127.0.0.1 at port
<any>. Send the stock listing to the client.

Stock Exchange: A Client/Server Application

Specification of the **Stock Market Server**

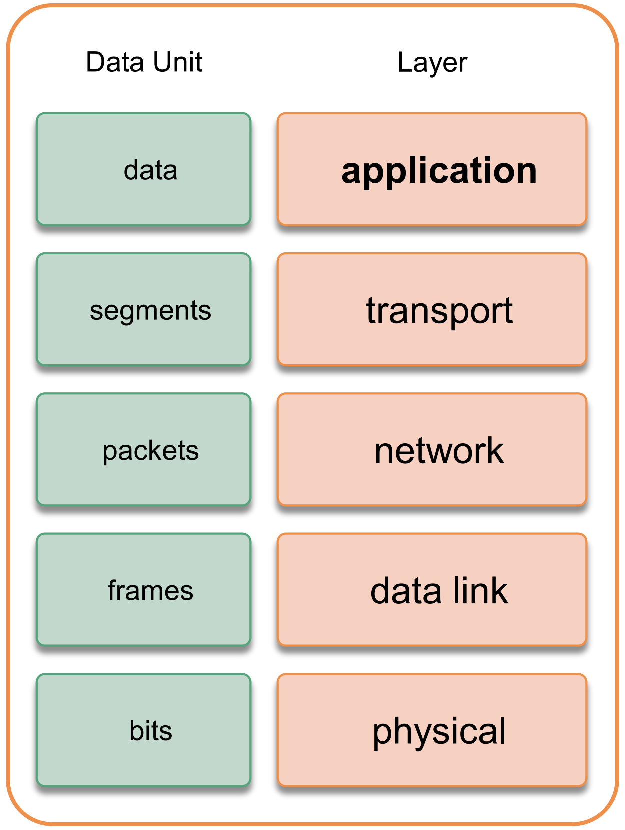
- Keeps a given list of stocks with their current prices
- Performs a continuous auctioning process to match buy and sell orders
- Keeps an order book with current orders
- Receives new orders from clients

Specification of the **Stock Market Client**

- Takes commands (LIST / BUY / SELL) from the user (via the command line)
- Relays these commands to the server
- Displays the server's responses

Specification of our **application-layer protocol**

- List-messages `LIST`
- Sell-messages `SELL <Ticker> <min. Preis>` (e.g., `SELL TSLA 430`)
- Buy-messages `BUY <Ticker> <max. Preis>` (e.g., `BUY LVMH 290`)



Only possible to buy single stocks.

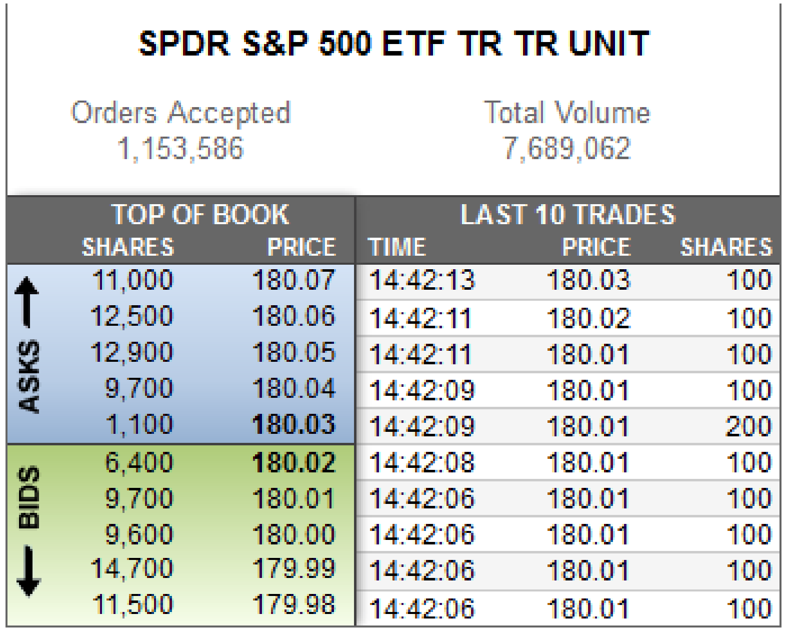
Server Implementation

The server started from this sample code

https://pythontic.com/modules/socket/udp-client-server-example

I extended that with

- The **Stock class**, to introduce an abstraction for stocks
  - `__init__` constructor
  - `ticker` -> ticker symbol of the stock
  - `price` -> current price of the stock
  - `orders_bid` -> list of buy orders / bids for the stock
  - `orders_ask` -> list of sell orders / asks for the stock
  - `buy` method -> accepts bids and executes auctions
  - `sell` method -> accepts asks and executes auctions
- A **dictionary** that holds stock objects and uses Ticker symbols as keys
- The **processClientMessage method**, to handle client messages and thereby implement our specified application-layer protocol
- The **communication** with the client



SPDR S&P 500 ETF TR TR UNIT

Orders Accepted 1,153,586 | Total Volume 7,689,062

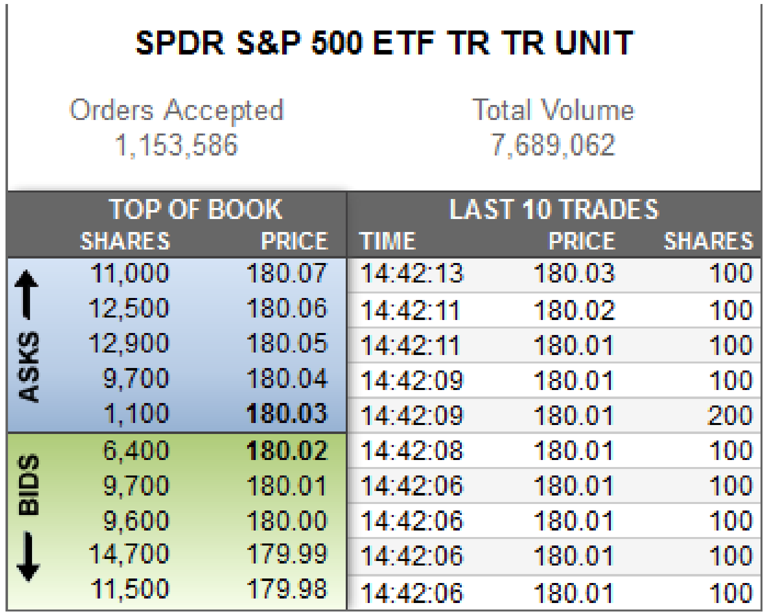
	TOP OF BOOK		LAST 10 TRADES		
	SHARES	PRICE	TIME	PRICE	SHARES
ASKS ↑	11,000	180.07	14:42:13	180.03	100
	12,500	180.06	14:42:11	180.02	100
	12,900	180.05	14:42:11	180.01	100
	9,700	180.04	14:42:09	180.01	100
	1,100	**180.03**	14:42:09	180.01	200
BIDS ↓	6,400	**180.02**	14:42:08	180.01	100
	9,700	180.01	14:42:06	180.01	100
	9,600	180.00	14:42:06	180.01	100
	14,700	179.99	14:42:06	180.01	100
	11,500	179.98	14:42:06	180.01	100

Server Implementation: Order Book

Per stock, the server holds **two sorted lists** with **buy** and **sell** orders

When a new buy order comes in, the **server attempts to match with the lowest** sell-order (and vice versa)

- **Successful**: Order is executed. Matched position is eliminated. New price is assigned.
- **No match**: Order is appended to the appropriate list



SPDR S&P 500 ETF TR TR UNIT

Orders Accepted 1,153,586 | Total Volume 7,689,062

	TOP OF BOOK		LAST 10 TRADES		
	SHARES	PRICE	TIME	PRICE	SHARES
ASKS ↑	11,000	180.07	14:42:13	180.03	100
	12,500	180.06	14:42:11	180.02	100
	12,900	180.05	14:42:11	180.01	100
	9,700	180.04	14:42:09	180.01	100
	1,100	**180.03**	14:42:09	180.01	200
BIDS ↓	6,400	**180.02**	14:42:08	180.01	100
	9,700	180.01	14:42:06	180.01	100
	9,600	180.00	14:42:06	180.01	100
	14,700	179.99	14:42:06	180.01	100
	11,500	179.98	14:42:06	180.01	100

Client Implementation

We create the command-line interface for our client

- Loop until the user enters “BYE”
- All other user inputs are relayed to the server (i.e., the user needs to know about our application-layer protocol)

Overview

This Week’s Menu

Lecture

Motivation
Computer Networks Basics
The Networking Stack
The World Wide Web

Guided Exercise

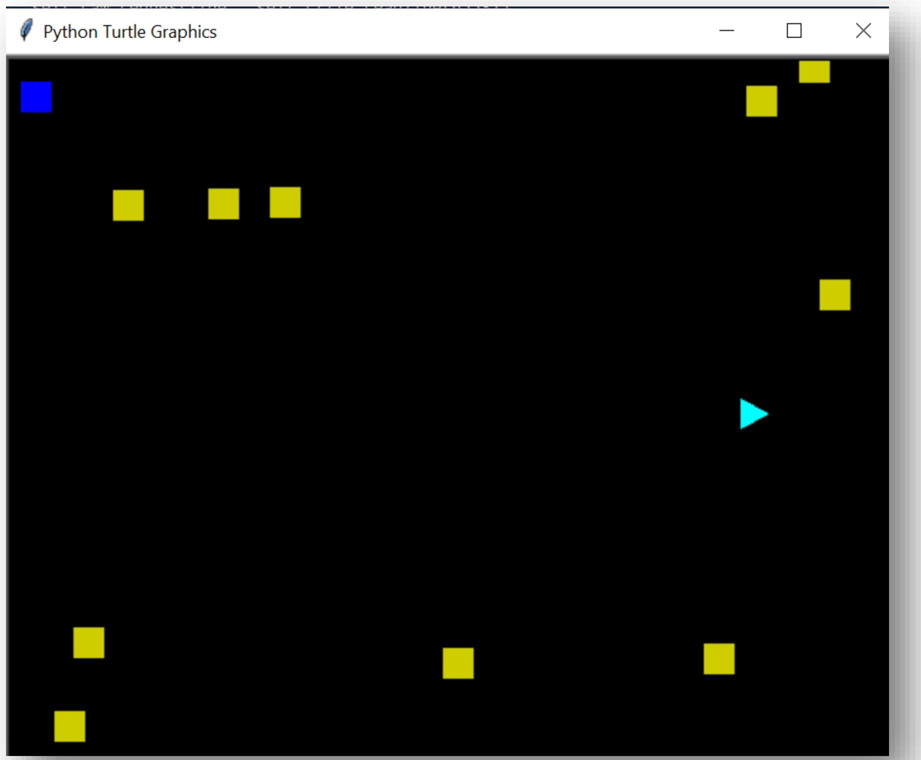
Briefing: Security
Stock Exchange
Overview: Food-and-Walls

Provided for You: Food-and-Walls

Simple 2D-Game, Graphics based on Python Turtle

Basic FaW in `FCS-BWL-6-DistributedSystems-DemoPackage/food-and-walls-basic`

Distributed FaW in `FCS-BWL-6-DistributedSystems-DemoPackage/food-and-walls-server`



Distributed Food-and-Walls with Public Highscores

The FaW Game with an online highscore system that consists of three main components:

After game-over, **Game-Clients** sends the player's name, and the score to the Score-Server

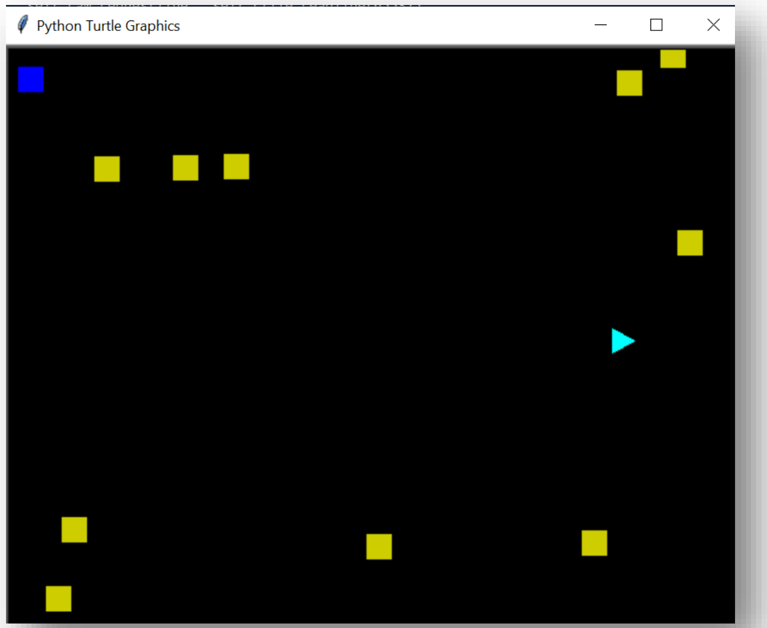
- UDP-based communication on port 5678
- Very similar to the stock exchange client

The **Score-Server** stores all highscores

- Listens for UDP on port 5678
- Appends new highscores to a local file

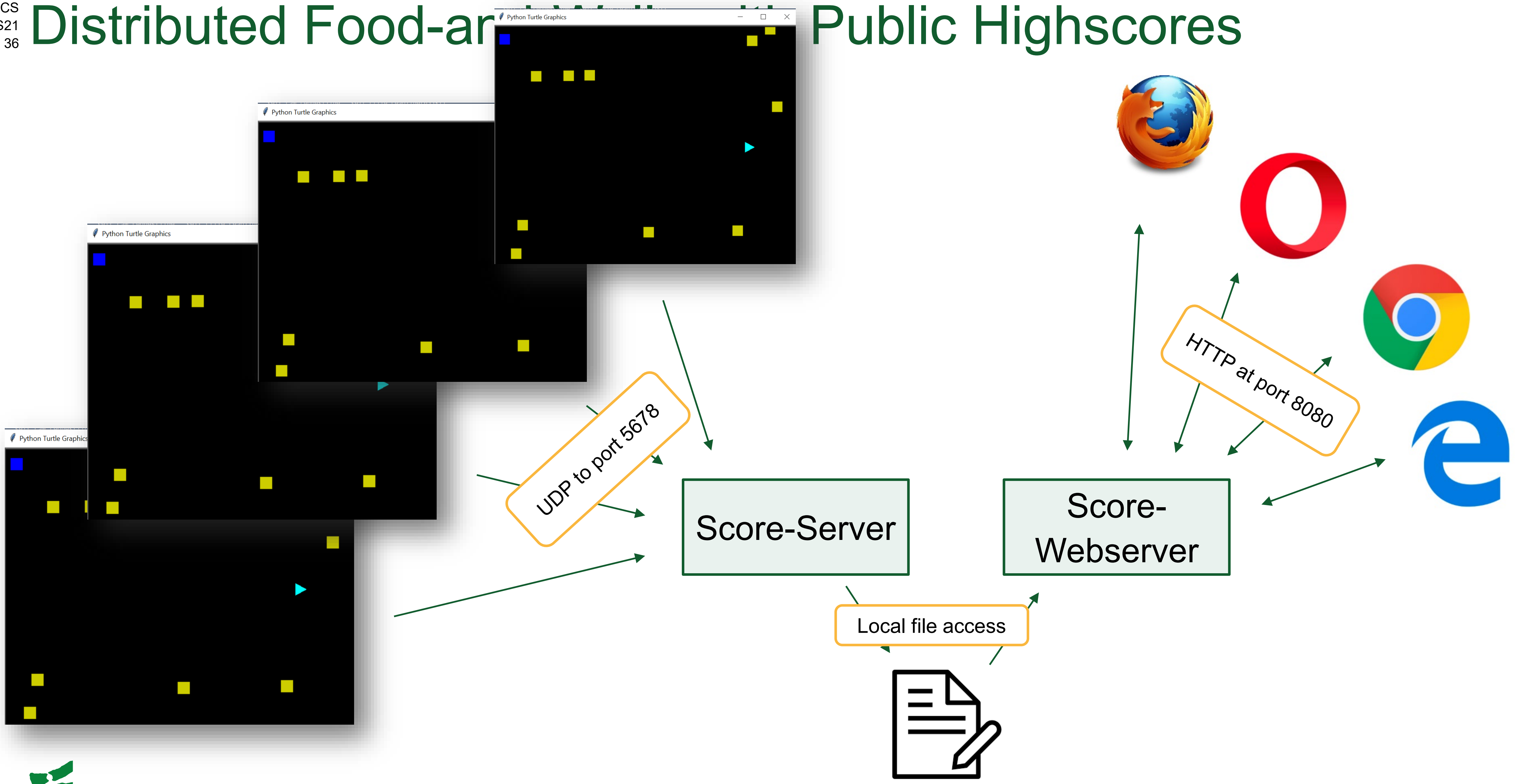
The **Score-Webserver** publishes the highscores on the Web

- Renders a simple Webpage using HTML
- HTTP-based interaction



To try it out on your system, you can run all three components simultaneously.

Distributed Food-ar Public Highscores



Any Questions / Comments / Doubts / Concerns?