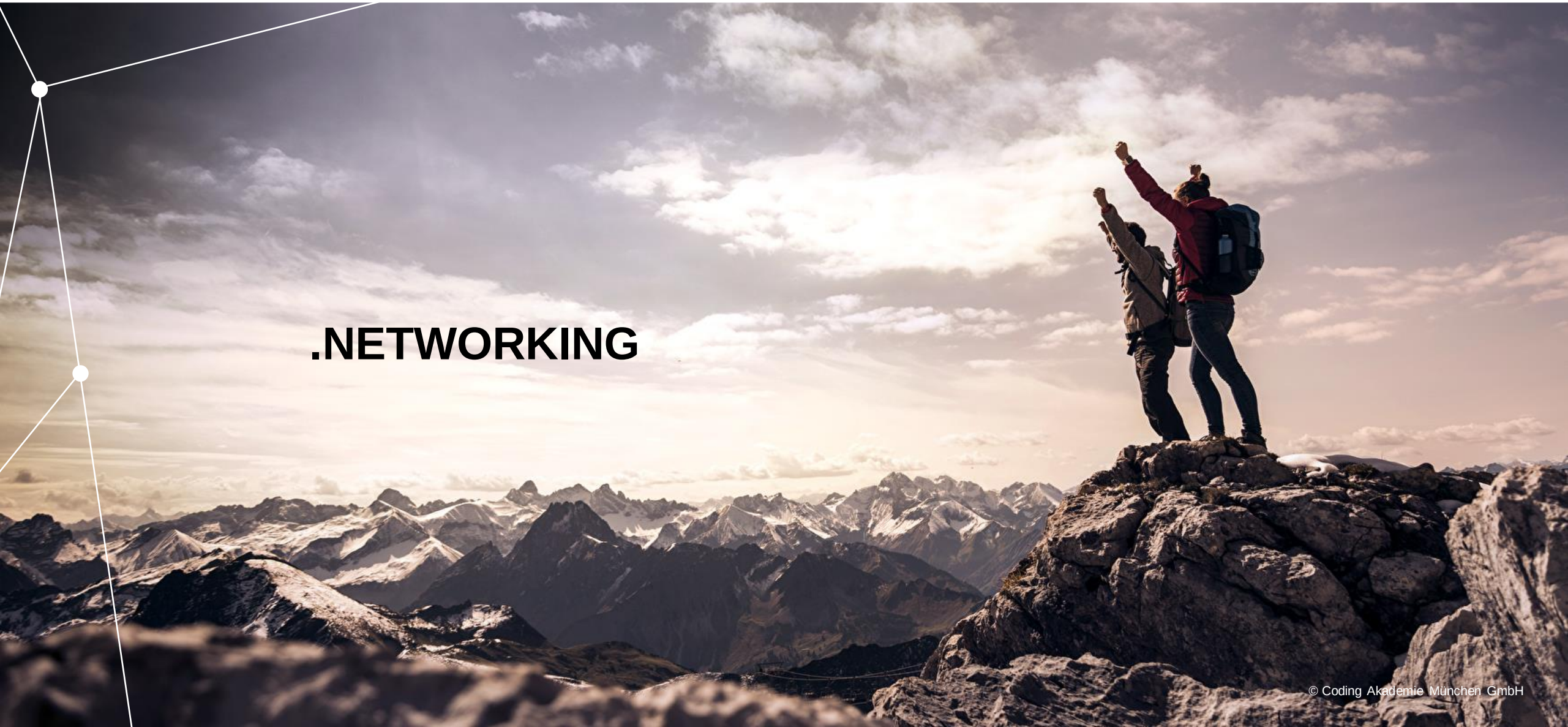
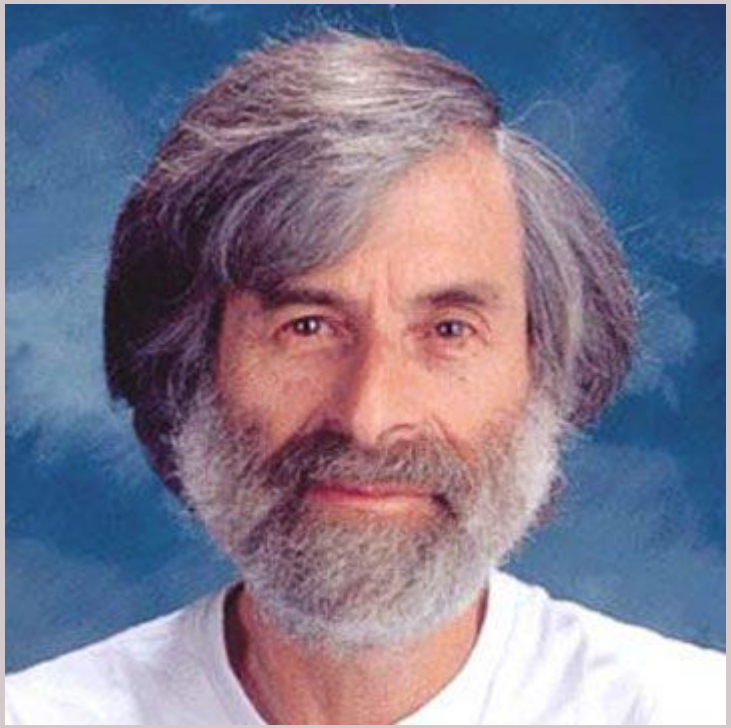
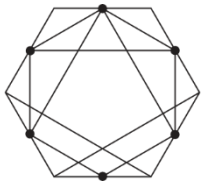




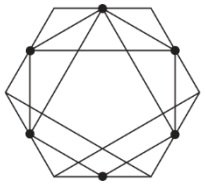
“A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.”

(Leslie Lamport)



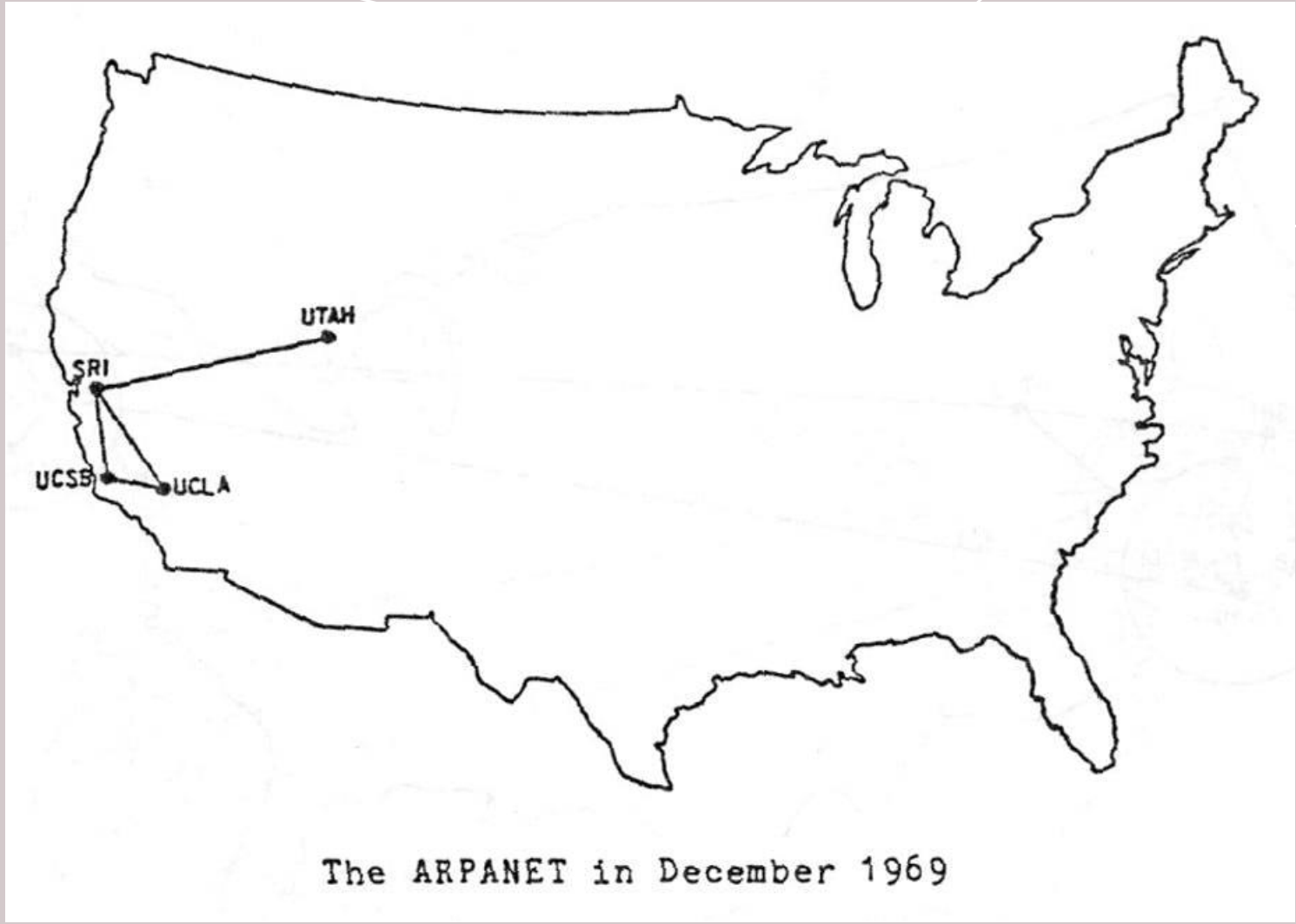
OVERVIEW

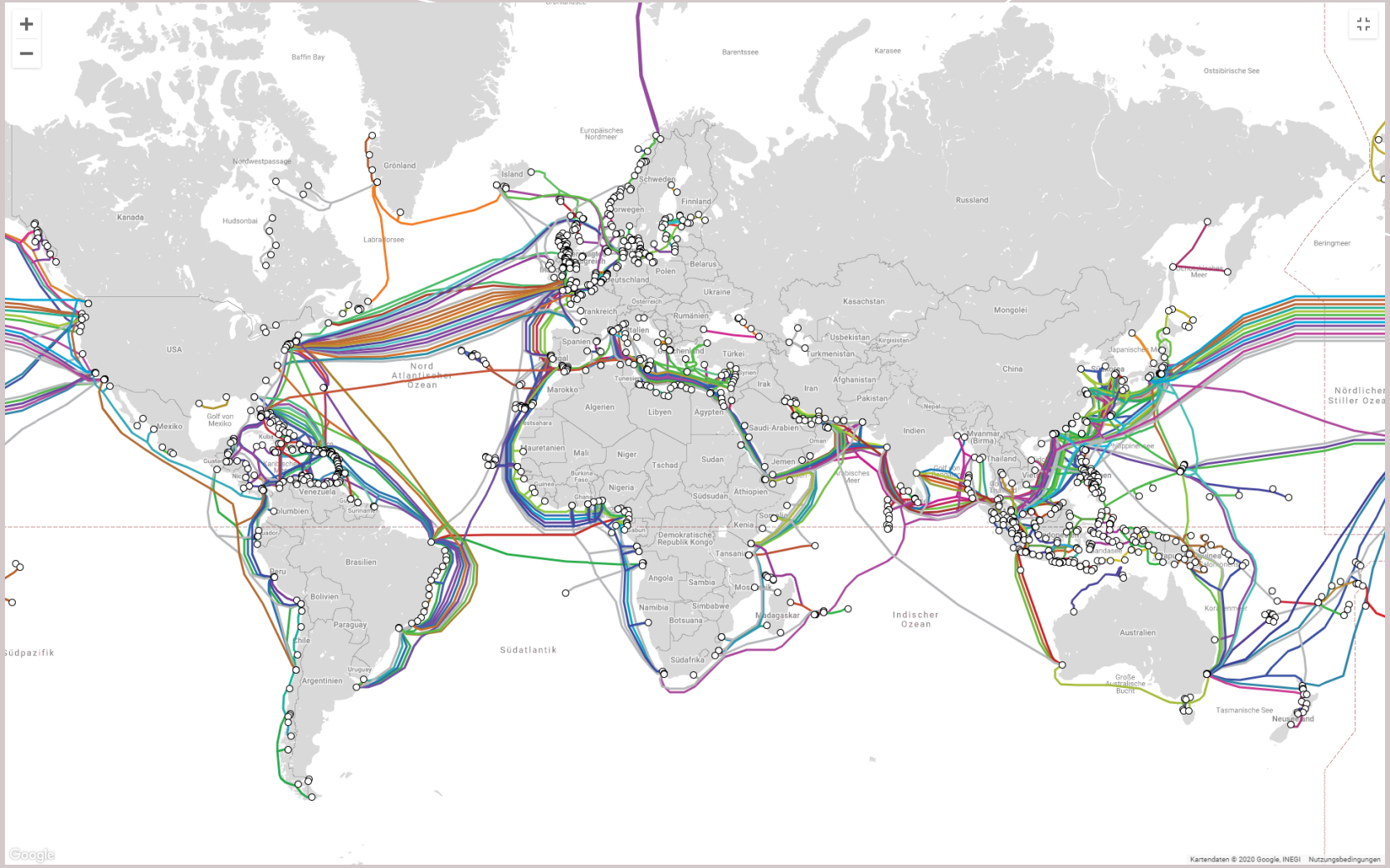
- Distributed Systems
- TCP/IP, HTTP, SOAP, REST/JSON
- Networking with .NET/C#
  - WCF – Windows Communication Foundation
  - .NET Core
- Example Project
- Microservices



DISTRIBUTED SYSTEMS

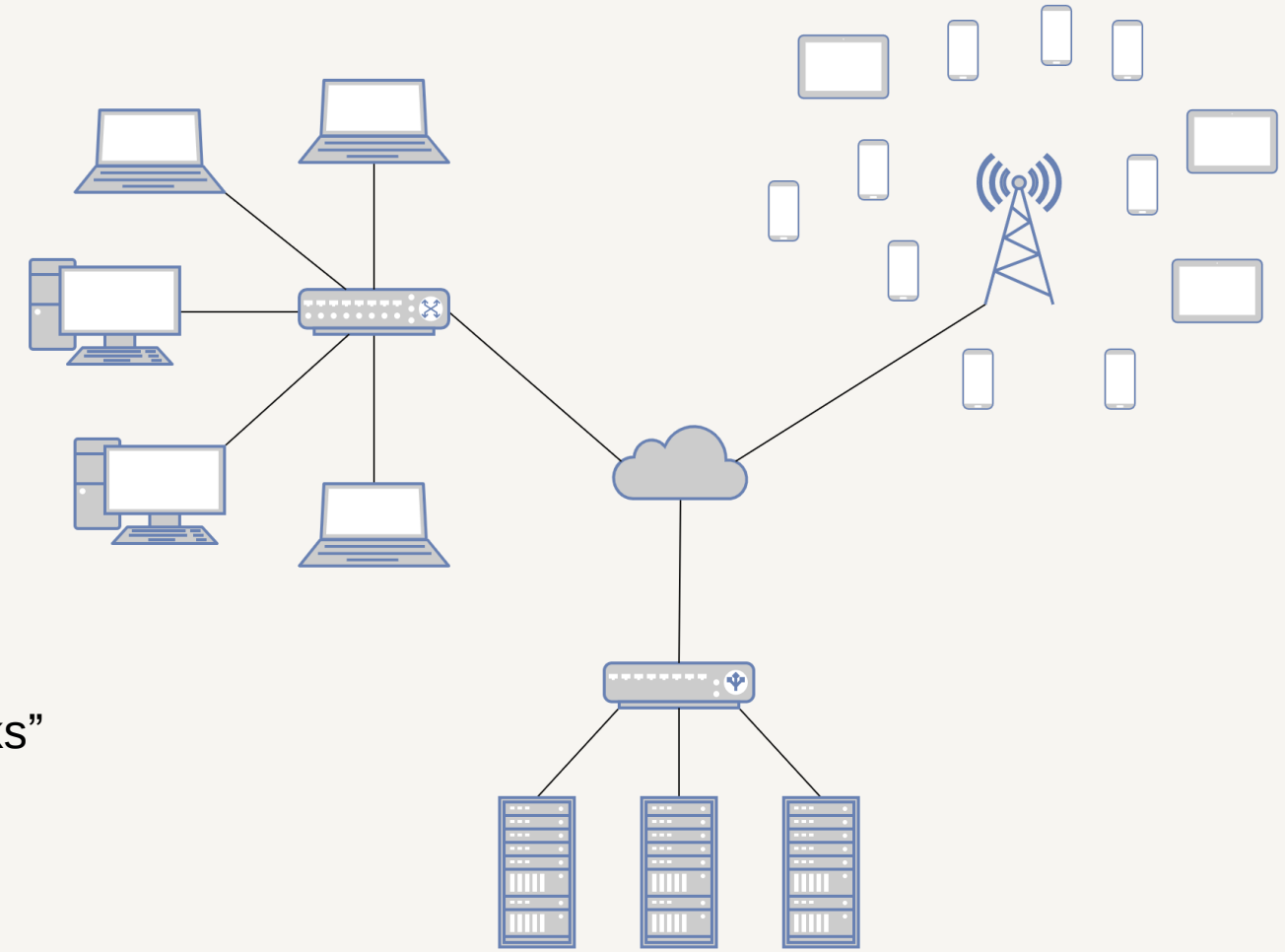
- Collection of multiple autonomous systems
- Collaboration in order to provide services or solve a problem
- Communication with messages

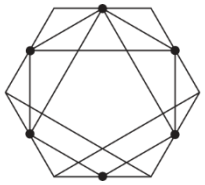




TOPOLOGIES

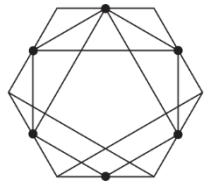
- Bus
- Ring
- Star
- Internet: “Network of Networks”



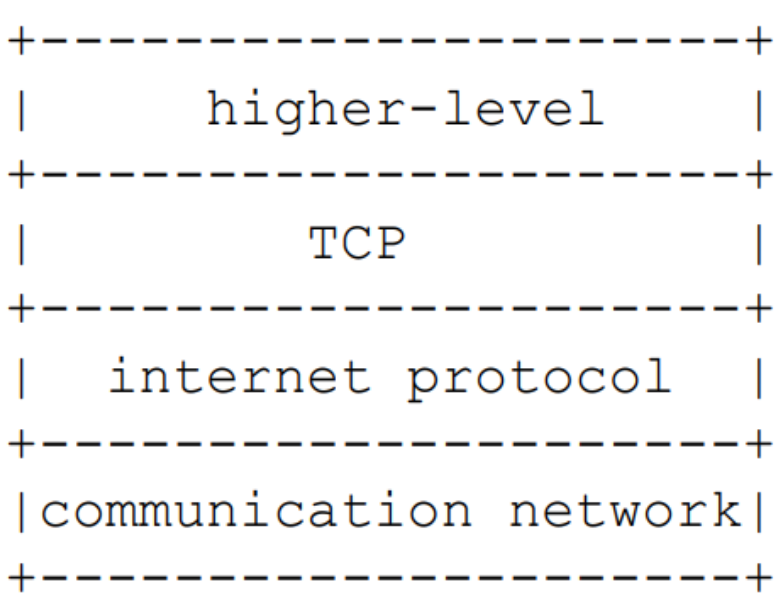


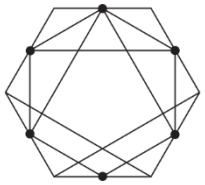
PROTOCOLS

- Protocols define rules how messages are exchanged
- **IP** – Internet Protocol
- **TCP** – Transmission Control Protocol
- **HTTP** – Hypertext Transfer Protocol



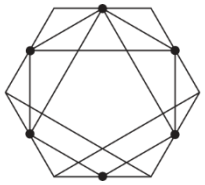
Protocol Layering





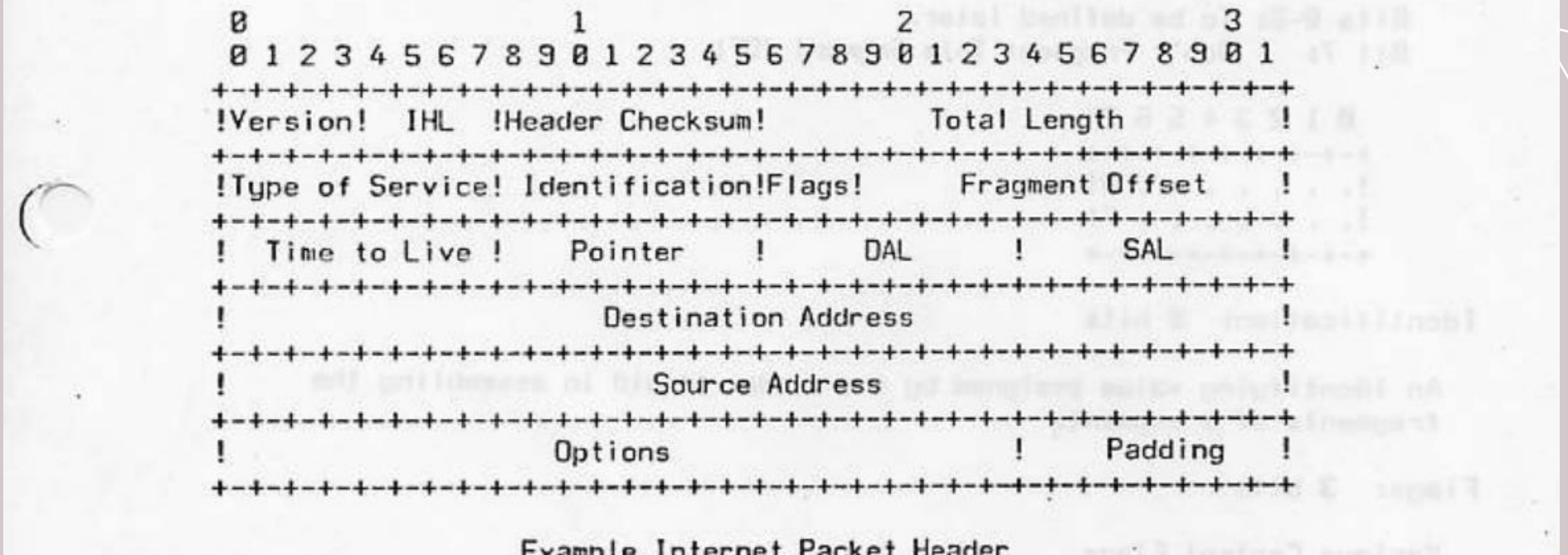
IP – INTERNET PROTOCOL

- Basic transmission of data
- Sending and receiving packets of data
- Supports complex network topologies
- Connection of multiple networks with gateways/routers
- Packets can be sent over multiple network nodes (“hops”)
- Protocol is independent of the actual physical transmission! (electrical, wireless, optical, carrier pigeon…)
- No reliable transmission! (packets can get lost)

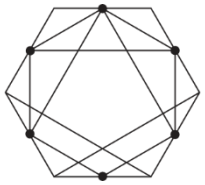


3.3. Internetwork Header Format

A summary of the contents of the internetwork header follows:

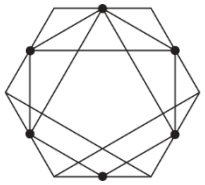


Example Internet Packet Header



TCP – TRANSMISSION CONTROL PROTOCOL

- Addressing of applications over ports
  - 80 – HTTP
  - 21 – FTP
  - …
- Reliable transmission!
  - Segmentation and numbering of packets
  - Received packets are always **ACK**nowledged
  - When no ACK is received, the sender re-transmits the message

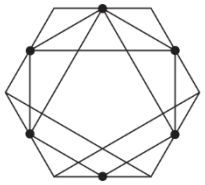


TCP CONNECTIONS

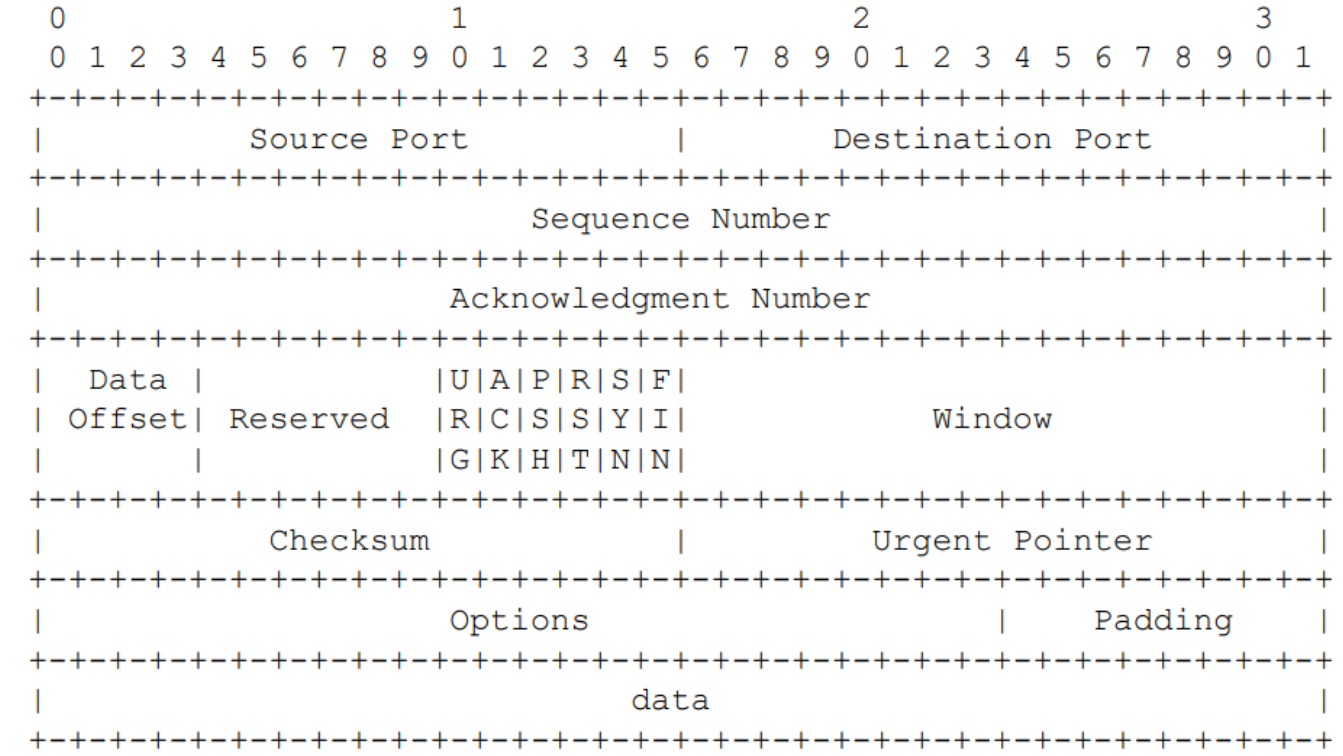
- Connection establishment: “3-Way-Handshake”
  1. Request from client to server (**SYN**chronize)
  2. Response from server (**SYN**chronize **ACK**nowledgement)
  3. Final confirmation from client (**ACK**nowledgement)

- Data Exchange

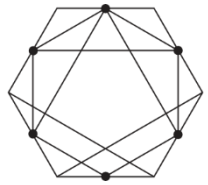
- Connection termination (“Teardown”)



TCP Header Format

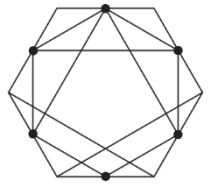


TCP Header Format



DEMO: TCP/IP

- Simple communication directly based on TCP



SOME PROTOCOLS BASED ON TCP…

DHCP

VoIP

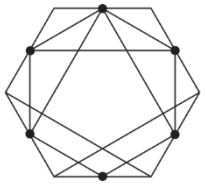
HTTP

DNS

SSH

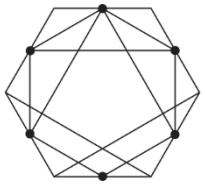
HTTPS

Telnet



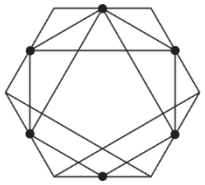
HTTP – HYPERTEXT TRANSFER PROTOCOL

- Application Layer Protocol (on the very top of the ISO/OSI layer model)
- Originally invented in 1989 by Tim Berners-Lee for transferring web pages (“human to machine communication”)
- Nowadays also very widespread for realizing communication between systems (“machine to machine communication”)



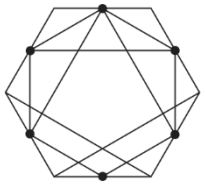
HTTP RESOURCES, URI & URL

- HTTP provides access to resources
- A resource can be uniquely identified with a “Uniform Resource Identifier” (URI)
  (e.g. a GUID, an XML namespace or a web address)
- If an URI provides means of locating the resource, it is called a “Uniform Resource Locator” (URL)
  (e.g. https://en.wikipedia.org/wiki/URL)



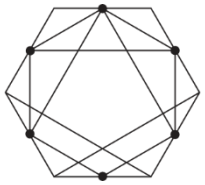
HTTP URL

- An HTTP URL consists of:
  - The “Scheme” followed by “:” (e.g. “http:” or “https:”)
  - An “Authority” consisting of “//” followed by:
    - An optional user info (e.g. “user1@”)
    - A host name (e.g. “wikipedia.org”)
    - An optional port (e.g. “:55555”)
  - A hierarchical path (e.g. “/wiki/URL”)
  - An optional list of URL parameters (e.g. “?id=4&anotherParameter=someText”)



HTTP METHODS

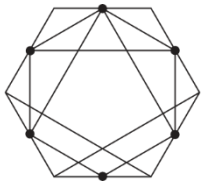
- A URL can be accessed using different methods, the most common ones are:
  - GET – read (used when typing in a URL in a browser)
  - POST – create a new resource in the hierarchy below the specified URL
  - PUT – update (or create) a resource at the specified URL
  - DELETE – remove a source at the specified URL



HTTP – DEALING WITH STATE

- HTTP itself is completely stateless (i.e. doesn’t remember any previous communication)
  - Load balancing and caching is made easy
  - Reduced complexity

- When state is needed (e.g. session IDs or shopping carts), it must be included in every request
  - Storing information on the client side with a “Cookie”
  - Required information is included with each request in the header
  - Alternatives to cookies exist (e.g. “device fingerprinting”, social media IDs…)
    - but don’t really solve the problem of users not being wanted to be tracked

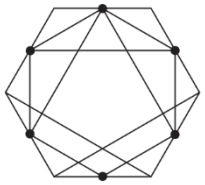


HTTP SERVICES

- HTTP can be used to implement machine-readable services

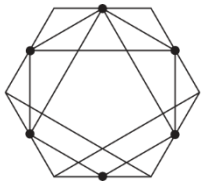
- SOAP – Simple Object Access Protocol
  - Remote method calls based on HTTP POST of XML data
  - Provides a variety of additional standards, e.g. for transactions or formally describing services

- REST – Representational State Transfer
  - Directly incorporates the HTTP philosophy of accessing resources with methods
  - Typically uses JSON data (other formats are possible, e.g. XML)



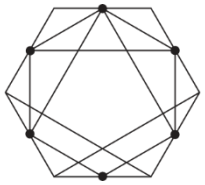
RUNNING EXAMPLE

- A simple service providing Create, Read, Update and Delete operations (CRUD)
- A text can be stored in the service which will assign an ID to it and return both text and ID
- The text stored under an ID can be read, updated and deleted



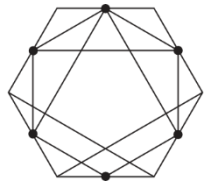
SOAP – SIMPLE OBJECT ACCESS PROTOCOL

- SOAP provides remote method calls by posting XML data to a URL
- One URL is typically used for several operations
  - Operation names and arguments are defined within the XML
- WSDL – Web Service Description Language
  - Provides a formal, machine-readable description of the service
  - Can be used to automatically generate a proxy for accessing the service



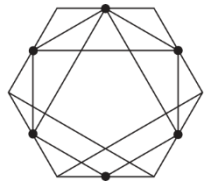
SOAP call of method “Create” with object parameter “data” having an attribute “Text”:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:tem="http://tempuri.org/"
                  xmlns:soap="http://schemas.datacontract.org/2004/07/SoapServiceWCF">
    <soapenv:Header/>
    <soapenv:Body>
        <tem:Create>
            <tem:data>
                <soap:Text>Hello SOAP!</soap:Text>
            </tem:data>
        </tem:Create>
    </soapenv:Body>
</soapenv:Envelope>
```



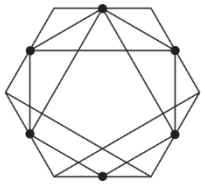
SOAP response, returning the created text and its assigned ID:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
    <s:Body>
        <CreateResponse xmlns="http://tempuri.org/">
            <CreateResult xmlns:a="http://schemas.datacontract.org/2004/07/SoapServiceWCF"
                          xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
                <a:Id>0</a:Id>
                <a:Text>Hello SOAP!</a:Text>
            </CreateResult>
        </CreateResponse>
    </s:Body>
</s:Envelope>
```



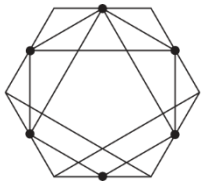
SOAP call of method “Read” with parameter “id”:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:tem="http://tempuri.org/">
    <soapenv:Header/>
    <soapenv:Body>
        <tem:Read>
            <tem:id>0</tem:id>
        </tem:Read>
    </soapenv:Body>
</soapenv:Envelope>
```



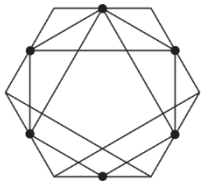
SOAP response:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
    <s:Body>
        <ReadResponse xmlns="http://tempuri.org/">
            <ReadResult xmlns:a="http://schemas.datacontract.org/2004/07/SoapServiceWCF"
                        xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
                <a:Id>0</a:Id>
                <a:Text>Hello SOAP!</a:Text>
            </ReadResult>
        </ReadResponse>
    </s:Body>
</s:Envelope>
```



REST/JSON

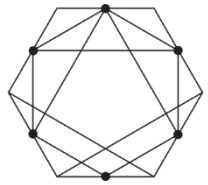
- REST directly incorporates the HTTP philosophy of accessing resources with methods
- The “REST way of thinking” is a bit different to remote method calls
  - Dynamic URLs representing hierarchical resources (instead of one URL for a whole service)
  - HTTP methods specifying access to resources
- Typically based on JSON data (but other formats are possible, e.g. XML)



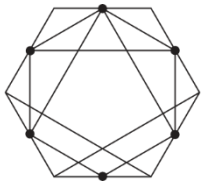
EXAMPLE WITH REST/JSON

- Create operation:
  - HTTP POST to http://localhost:333/main
  - HTTP body: {"text":“Hello REST!"}
  - Response: { "id": 0, "text": "Hello REST!"}

- Read operation:
  - HTTP GET from http://localhost:333/main/0
  - Response again: { "id": 0, "text": "Hello REST!"}
  - Note that the ID is not a parameter but becomes part of the URL!



WCF

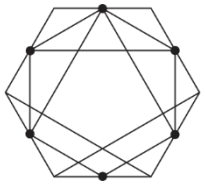


WCF – WINDOWS COMMUNICATION FOUNDATION

- .NET Framework for realizing distributed systems based on services
  - “Standalone Service” (to be deployed and hosted e.g. on IIS or Azure)
  - “Self-hosted Service” (can be hosted within an arbitrary application)

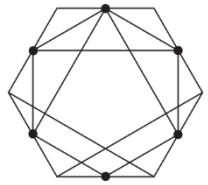
- “ABC-Principle”:
  - Address – location of the service
  - Binding – mode of communication (e.g. protocol, data format or security)
  - Contract – structure of data and definition of operations

- Each can be further specified with so-called “behaviors”



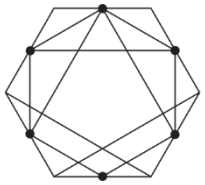
WCF – CONFIGURATION

- WCF separates a service's implementation from its configuration
- Implementation: programming code realizing data structures, interfaces and service logic
- Configuration: definition of operational aspects (hostname, protocols, security…)
  - XML configuration file
  - Configuration with program code also possible



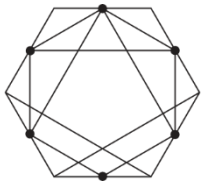
WCF – CONTRACTS

- Both WCF services and clients can use contracts to specify the service interface
- Data Contract: specifies the structure of data that is transmitted over the service
- Service Contract: specifies which methods a service provides



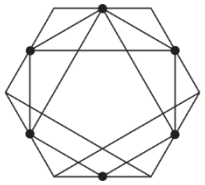
WCF data contract:

```
[DataContract]
15 references
public class MainData
{

    [DataMember]
    5 references
    public int Id { get; set; }

    [DataMember]
    2 references
    public string Text { get; set; }

}
```



WCF service contract:

```
[ServiceContract]
1 reference
public interface IMainService
{

    [OperationContract]
    1 reference
    Task<MainData> Create(MainData data);

    [OperationContract]
    1 reference
    Task<MainData> Read(int id);

    [OperationContract]
    1 reference
    Task Update(bool createIfNotExisting, MainData data);

    [OperationContract]
    1 reference
    Task Delete(int id, bool errorIfNotExisting);

}
```

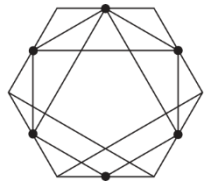


WCF – DATA CONTRACTS

- Per default, the contract's .NET identifiers are also used within the service

- If the service should use a different naming (e.g. for other naming conventions), a mapping can be specified, for example to use lower-case attributes in JSON:

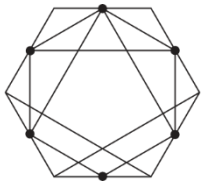
```
[DataMember(Name = "text")]
6 references
public string Text { get; set; }
```



WCF – REST SERVICE CONTRACTS

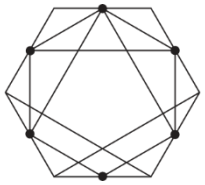
- The mapping to REST methods, data formats and URL parameters can be done like this:

```
[OperationContract]
[WebInvoke(
    Method = "DELETE", ResponseFormat = WebMessageFormat.Json,
    BodyStyle = WebMessageBodyStyle.Bare,
    UriTemplate = "/main/{id}?errorIfNotExisting={errorIfNotExisting}"
    )]
2 references
Task Delete(string id, bool errorIfNotExisting);
```



WCF – BINDINGS

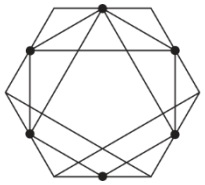
- WCF provides a variety of different bindings:
  - webHttpBinding - REST
  - basicHttpBinding – SOAP 1.1 (“lightweight SOAP”)
  - wsHttpBinding – SOAP 1.2 (SOAP with lots of additional features, e.g. sessions or transactions)
  - wsDualHttpBinding – duplex SOAP binding
  - netTcpBinding – proprietary binding directly based on TCP
  - Several more, e.g. supporting Windows Pipes or MSMQ (Microsoft Message Queueing)



WCF – CONFIGURATION FILES

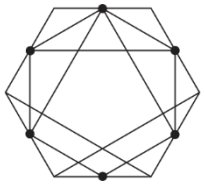
- Configuration files can specify:
  - Mapping of service contract to its implementation
  - Binding
  - Endpoint, address and port (can also be specified “from outside”, e.g. IIS, Azure or VS)
  - Behavior, configuration etc.

- Can get a bit confusing… - often there are multiple ways to do the same thing



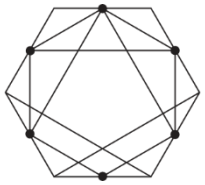
WCF service contract for a standalone SOAP service:

```
<system.serviceModel>
  <behaviors>
    <serviceBehaviors>
      <behavior>
        <serviceMetadata httpGetEnabled="true" httpsGetEnabled="true"/>
        <serviceDebug includeExceptionDetailInFaults="false"/>
      </behavior>
    </serviceBehaviors>
  </behaviors>
  <protocolMapping>
      <add binding="basicHttpsBinding" scheme="https" />
  </protocolMapping>
  <serviceHostingEnvironment aspNetCompatibilityEnabled="true" multipleSiteBindingsEnabled="true" />
</system.serviceModel>
```



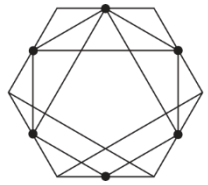
WCF service contract for a standalone REST service:

```
<system.serviceModel>
  <behaviors>
    <serviceBehaviors>
      <behavior name="serviceBehavior">
        <serviceMetadata httpGetEnabled="true"/>
        <serviceDebug includeExceptionDetailInFaults="true"/>
      </behavior>
    </serviceBehaviors>
    <endpointBehaviors>
      <behavior name="web">
        <webHttp/>
      </behavior>
    </endpointBehaviors>
  </behaviors>
  <services>
    <service behaviorConfiguration="serviceBehavior" name="RestServiceWCF.MainService">
      <endpoint behaviorConfiguration="web" binding="webHttpBinding" contract="RestServiceWCFContracts.IMainService"/>
    </service>
  </services>
  <serviceHostingEnvironment multipleSiteBindingsEnabled="true"/>
</system.serviceModel>
```



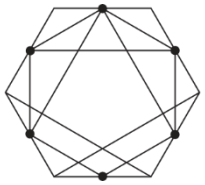
WCF service contract for a self-hosted REST service:

```
<system.serviceModel>
    <behaviors>
        <serviceBehaviors>
            <behavior name="">
                <serviceMetadata httpGetEnabled="true" httpsGetEnabled="true" />
                <serviceDebug includeExceptionDetailInFaults="false" />
            </behavior>
        </serviceBehaviors>
    </behaviors>
    <services>
        <service name="SelfHostedRestServiceWCF.MainService">
            <endpoint address="" binding="webHttpBinding" contract="RestServiceWCFContracts.IMainService">
                <identity>
                    <dns value="localhost" />
                </identity>
            </endpoint>
            <host>
                <baseAddresses>
                    <add baseAddress="http://localhost:333/" />
                </baseAddresses>
            </host>
        </service>
    </services>
</system.serviceModel>
```



DEMO: WCF SERVICES

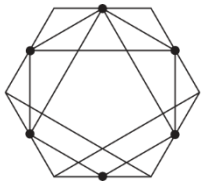
- SOAP
- REST
  - Standalone
  - Self-hosted
  - .NET Core



WCF CLIENTS

- WCF can also be used to implement service clients

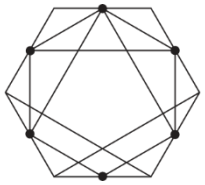
- Multiple approaches possible:
  - “Add Service Reference” – works very well with SOAP/WSDL, often not so good with REST
  - ChannelFactory – simple way to connect to a service when .NET contracts are available (e.g. within a project that is shared between client and server)
  - “Manually” connecting to a REST service



DEMO: WCF CLIENTS

- RestServiceWCFClient (ChannelFactory)

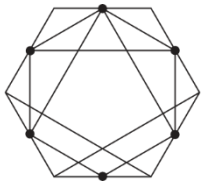
- Client App
  - Manual connection for REST
  - Automatically generated proxy for SOAP (via “Add Service Reference”)



EXERCISE: IMPLEMENT SOME WCF SERVICES

- Convert your “Fruit Stand” project to actual web services based on technology of your choice (REST, SOAP, standalone, self-hosted…)
  - Search Service
  - Product Service
  - Price Service

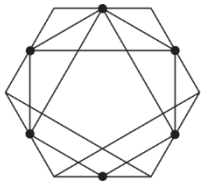
- Implement a simple client based on the Console



EXERCISE: FRUIT STAND HINTS

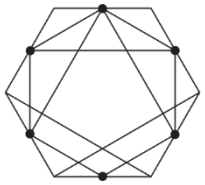
- **To include a .csv file in the service:**
  - **Copy it into the project**
  - **Go to file properties, set build action to “Content” and “Copy to output directory if newer”**

  - **The file can then be accessed like this:**
    - **string filePath = HostingEnvironment.MapPath("~/Fruit Search.CSV");**

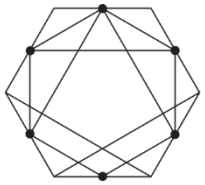


EXERCISE: FRUIT STAND HINTS

- **Product and Price can be either realized with:**
  - **HTTP GET and passing a list of Ids e.g. within the URL or as URL parameters**
    - **The values need to be separated with a special character**
    - **Service needs to parse the list**
  - **HTTP POST and passing either a dedicated request object or a list of IDs directly**
    - **Simple list of values in JSON: [0, 1, 2, 3]**

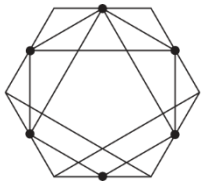


MICROSERVICES



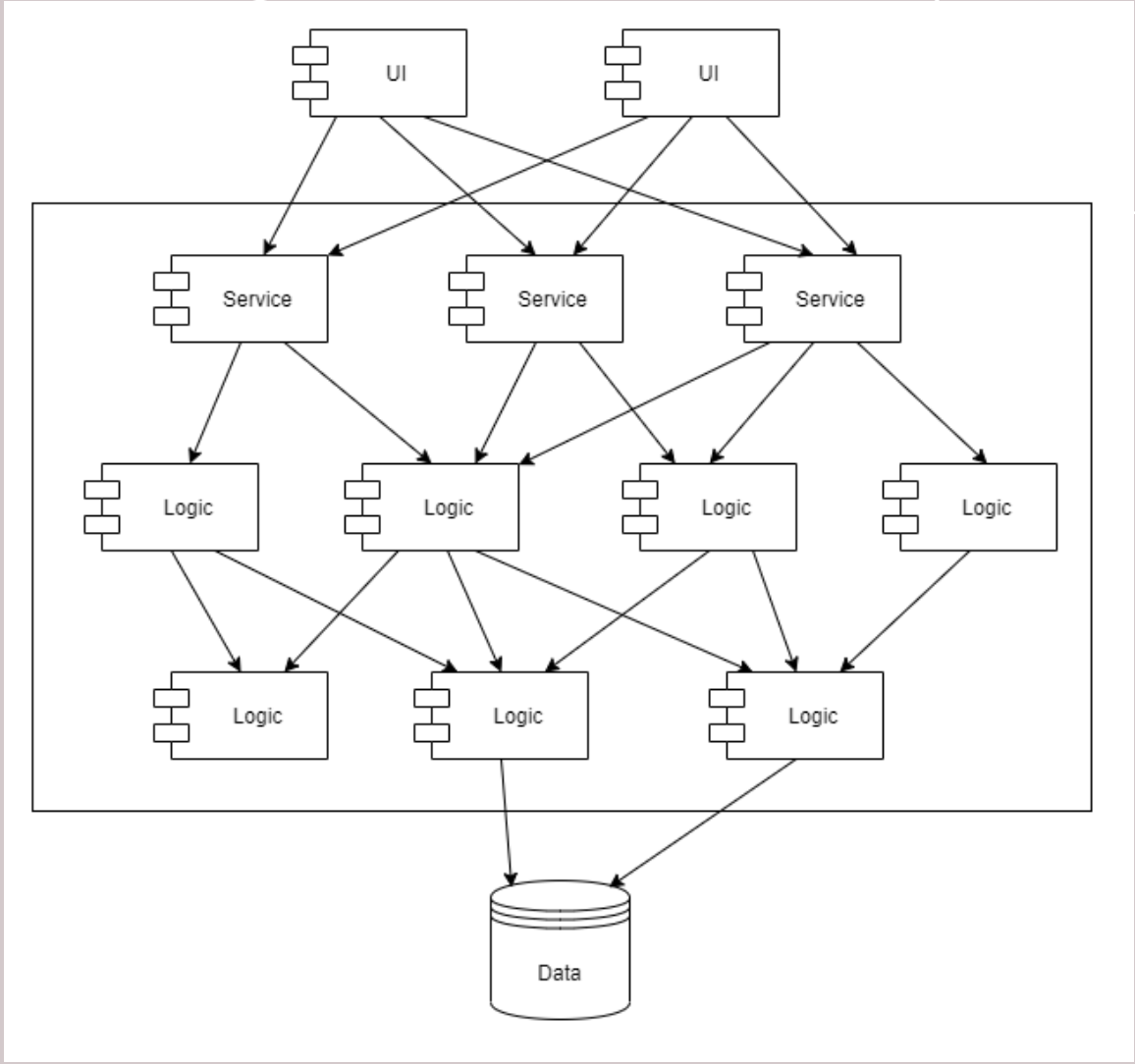
MICROSERVICES

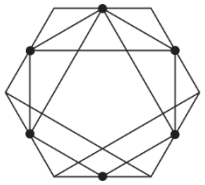
- Architectural style based on “smaller” services (instead of one large monolithic system)
- Loose coupling between services (above all, no shared code!)
- Smaller and more independent teams (~ 5, maximum of 10 people)
- Shorter release cycles
- Works very well with an agile development process



SERVICE ORIENTED ARCHITECTURE

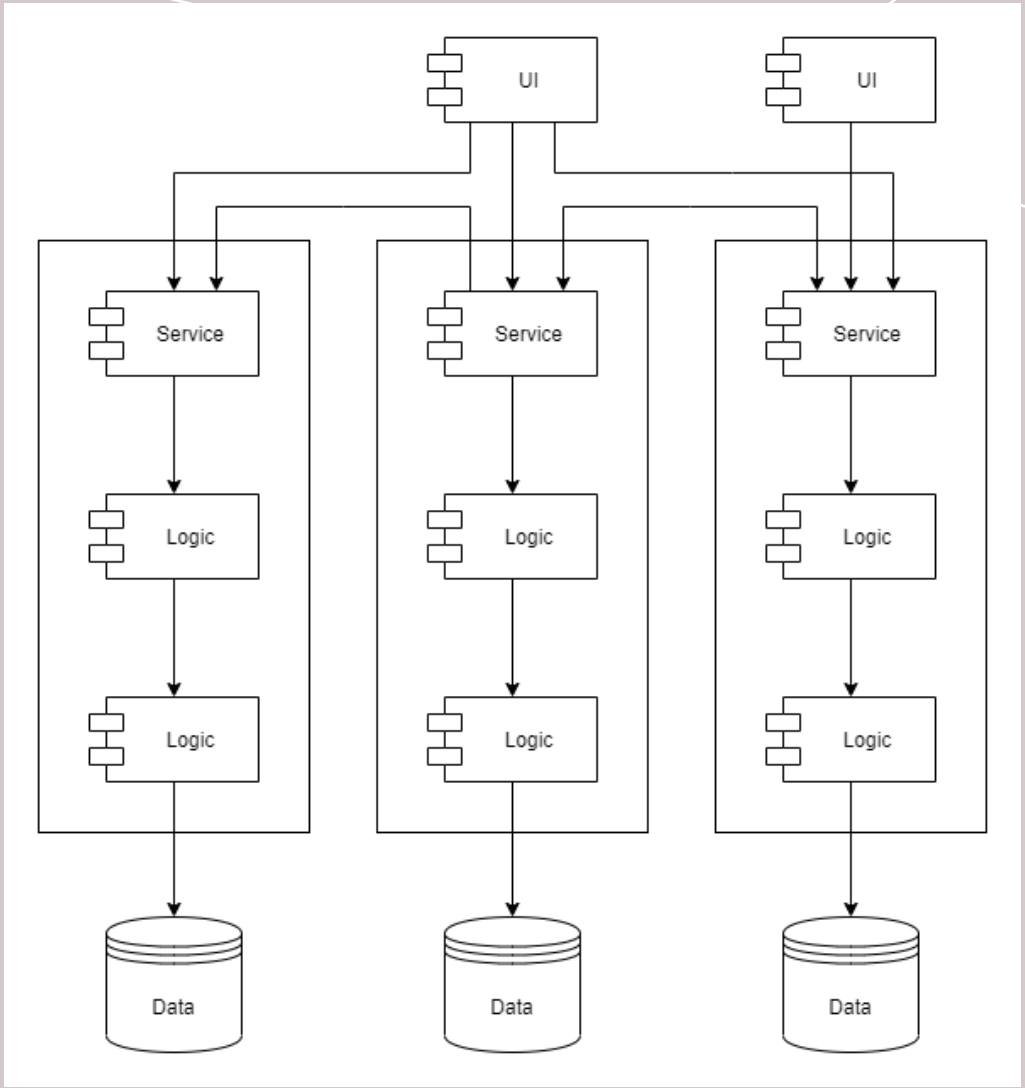
- SOA - Service-oriented Architecture (late 2000’s, early 2010’s):
- Separation between business logic (service) and clients (e.g. UI or other services)
- Communication over established Internet techniques (mostly SOAP over HTTP)
- Clear service interfaces over a well-defined API
- Many backend systems turned out to be rather heavyweight:
  - Big projects with lots of code providing multiple API’s
  - Often no clear separation between different business logic domains
  - Shared code between different logic domains
  - Hard to change only one thing without possibly changing something else
  - Project consists of a single deployment unit

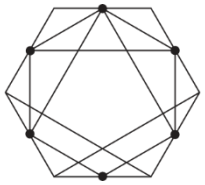




MICROSERVICES

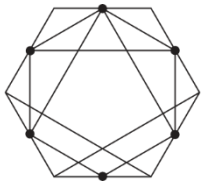
- Microservices “philosophy” has early origins, became widespread in the mid 2010’s
- Smaller services dealing with a defined busines logic domain (“Do one thing and do it well”)
- Self-contained services (no shared code with other services)
- Services can be deployed independently from each other
- Communication between services must be “explicit”:
  - API calls
  - Integration over the UI
  - Synchronization of data (possible but not always ideal as it introduces dependency)





BOUNDED CONTEXT

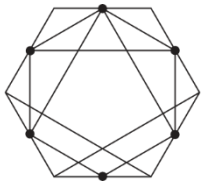
- “Bounded Context” is a term from Domain-Driven Design
- Within a bounded context, an entity has a well-defined meaning
- In another context, the same entity can have a different meaning
  - “User” in context “User Management” will focus on things like “Name”, “Address” or “Login”
  - “User” in context “Search” will focus e.g. on “personal preferences” or “search history”



VERTICAL AND HORIZONTAL SEPARATION

- Vertical separation (“technical”):
  - User Interface
  - Business Logic
  - Data Storage
  - External Services

- Horizontal separation (“business domain”):
  - Separation by bounded context
  - Typically matches the structure of an organization!

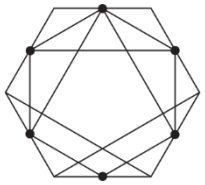


CONWAY'S LAW

- "Any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure." (Melvin Conway)

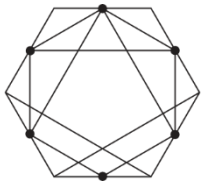
- Not ideal: separation of teams along technical boundaries (e.g. UI team and Backend team)
  - Typically requires lots of communication about technical aspects
  - Everybody must deal with lots of business logic from different domains
  - High risk for building a monolith!

- Better: separation based on business domains
  - System architecture with only low coupling
  - Reduced communication between teams
  - Improved communication between development and stakeholders



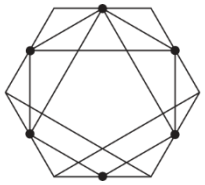
MICROSERVICES - DATA

- In order to achieve a real decoupling, also data storage needs to be decoupled!
- Separation of logic alone will not bring real independence if in the end everything comes together again in one single shared database with many dependencies between logic domains
- Data synchronization between services makes sense in some scenarios (e.g. when migrating existing architectures)
- Event-driven messaging queues can achieve data synchronization without coupling too much



MICROSERVICES - UI

- There are multiple approaches for realizing user interfaces within a microservice architecture
- User interface can be the “glue” between several services
- Approach A: monolithic UI which calls and integrates all the services
  - Communication between services can be handled over the UI
- Approach B: each service also provides its own part of the UI
  - Very high decoupling possible



MICROSERVICES

- Microservices can lead to:
  - A more manageable amount of code for each team
  - Shorter development times
  - Higher code quality
  - Happy developers ☺

- However, they come at a cost:
  - Loose coupling between components creates an overhead in development and communication
  - Integration testing becomes difficult
  - The path from an existing monolithic project to Microservices is typically hard