Bernd Owsnicki-Klewe

# Distributed Software Architectures

## 0. Introduction

***You know you have a distributed system when the crash of a computer you've never heard of stops you from getting any work done.***

LESLIE LAMPORT

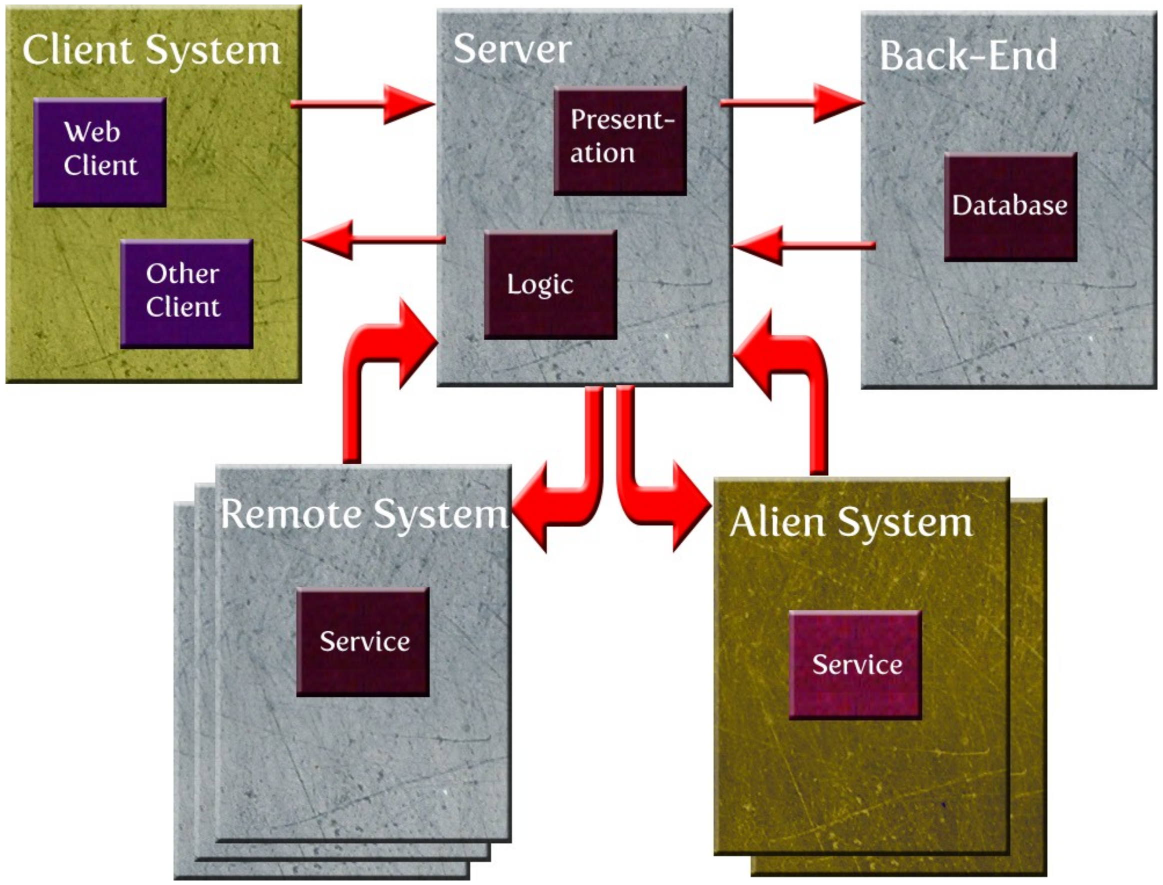
## 1. Course Overview

A first course in Distributed Software Architectures needs some introductory remarks about motivation, contents and extent of coverage. For the moment, we will just assume a general knowledge about modern applications that are not located one private machine anymore. We refer to applications that utilize more than computer system as **Distributed Applications**.

Examples for applications like these are **Web Applications**, i. e. applications that are available over a WWW connection. In order to gain a first insight into the development problems and architectures, let's study an example architecture that is fairly typical for these kinds of applications.

### 1.1 Example Architecture

A closer look into these applications reveals a division into five different usages for computer resources:



**Target Distributed Architecture**

In this course we will assume the role of an application developer, that is someone that is responsible for the application as a whole including logic, user interface, database connections, availability, reliability, etc.

### 1.1.1 Components

For us as developers, the following system types are relevant:

- *The Client System*: We don't have too much control over the properties of the client system. It could be running weird operating systems, etc. The popularity of web clients (web browsers) stems from the fact that most clients can rely on the properties of web connections/browsers; these client types also allow for the transport of parts of the application from the server, like Applets, Flash, JavaScript, etc.

- *Alien Systems*: Other systems we don't have control over but we would like to use anyway. Examples are search engines, shopping servers, identification/authentication servers or any other computer that can offer a service that can be used for our application.

- *"Our" System*: These are computers located under our control – that doesn't mean they're homogenous. But we can install software on them, start processes, monitor and tune performance, reconfigure them and and and... These are the ones shown in gray and these are the ones we're concentrating on in this course.

Our system can actually comprise a number of physical computers, like web servers, application servers, database servers, remote machines for special services, redundant system to secure availability, etc.

### 1.1.2 Connections

Choices for the connections involved of course depend on technological and administrative criteria. We will have a look at the ones believed to be the most widely used:

*Client ←→ Server*: A multitude of connection models are available, but in the most versatile setting you could assume that one of the many Internet-related transmissions are used, like TCP/IP,, UDP, http, ftp, or similar ones. The main objective behind this choice is that these protocols define a common layer and there's already a lot of client-side technology available in the form of web browsers which not only are concerned with rendering issues, but also with the possibility to employ more dynamic interfacing technologies, like applets or Flash. These browsers also offer client-side programming (scripting) technology, like JavaScript.

Apart from deciding on a (low-level) protocol, such as TCP/IP plus http, it's almost equally important to find a information structure that allows for the transmission of complex data between client and server. In many cases the protocol offers a way to encode this information – such as the http request/response structure, but the exact formalism is frequently imposed by singular design decisions rather than general considerations. Here XML can be used to fill this gap – JSON could be an alternative, too. We will deal with XML in depth in the first part of this course.

*Server ←→ Back End*: The choices here are a bit more involved, because there are quite a few options to connect application software on the server with a database. Most common are mixes based on PHP and an SQL database with an integration through the programming language, general-purpose programming languages with a language-specific DB connectivity or with an ODBC connection. Java-based technology provides a mean to connect an application to a database via JDBC.

*Server ←→ Remote System*: If you have perfect control over the remote system, direct connections can be used. Java technology allows to connect software over Java-RMI, heterogenous software would require a language-independent interface, like RMI-IOOP. CORBA is a more abstract and more protable way of accessing applications on another machine.

*Server ←→ Alien System*: These connections rely on whatever mechanism is available on both sides. Since we are in control of the Server, we should be able to implement everything that is required by the alien machine. If the alien machine is equipped with something like CORBA, such a connection could be used – if it's a machine with a lot of "legacy" applications, other ways should be established. Lately a message-based technology came into focus that communicates by sending complete messages back and forth, rather than trying to access any kind of middleware on the other system. This technology is know as **Web Services** and will be treated on an introductory level here and in depth in the follow-up course.

## 2. Course Outline

These notes will sketch a first course in Distributed Software Architectures (DSA) and give an outline of the following areas:

- *XML*
  Assuming that XML – with all the perceived criticism, of course – is still a valid design decision for a communication structure, we'll cover the most important aspects theoretically and explore the embedding of XML into a software architecture.

- *Client-side Processing*
  Even though many aspects of client-side processing (mainly html and CSS) is generally covered in an introductory Web Design course, the overall architecture of the client side is of great importance. We will see, that there are different ideas how to distribute activities between server and client. In many cases the decision how to distribute these activities can be tricky and a developer should be familiar with a multitude of options. The main tool in this section will be Javascript and – for simplicity reasons – a closer look at a particular library that is used to overcome the problem of “browser peculiarities” and to simplify the coding of routine tasks. AJAX (Asynchronous Javascript and XML) will be another topic covered in this section. JSON is another interesting aspect that gained a lot of importance of the years.

- *DB-Persistence Models*
  Since databases form the backbone of modern applications, especially distributed and web-based applications, a basic understanding of their structure, use and limitations is necessary for the following considerations. We will look at different DB models quickly and get some working knowledge about data models, SQL, transactions, DB management and especially DB connectivity in terms of ODBC/JDBC.

  This is not meant as a full-fledged database course; in fact having attended a database course before DSA1 would be very desirable, so this could work more as a refresher.

  Java EE supports a multitude of different persistence models – apart from JDBC, SQL can be embedded into JSP/JSF environments , it offers coupling of DB technology with the Java Object Model as well as approaches that go beyond the “standard” relational model under the “buzzword” of non-SQL databases.

  A modern treatment of Database connectivity is the automation of mapping between the relational DB model and Object-Oriented methods. Object-Relational Mappers (ORM) and advanced APIs, such as the Java Persistence API (JPA) are now in the foreground.

- *Web-based architectures*
  We will have a quite detailed look at the architectures behind modern web-based applications, especially the roles of web servers, application servers and the software architectures behind them. Examples will be Apache/Tomcat and Glassfish together with Java. The technologies we will study will be Servlets and JSP/JSF.

  Of course, there are alternatives to these models and we will briefly discuss some of them in the larger context of the Model-View-Controller paradigm.

- *Service-Oriented Architectures*
  The fast rise of these architectures will be reflected in a chapter about RESTful architectures and their implementation in JAX-RS.

## 3. This Course in the Context of Computer Science

### 3.1 The role of Distributed Systems

Looking at the underlying assumptions about what a Computer Science curriculum is all about, studying “Distributed Computing” - used here in a very general sense, rather than a more specific one like “Distributed Systems” - reveals a lot about the workings of modern software systems, as well as about the methodology of constructing systems like this on a conceptual level.

In this general sense, distributing applications can take many forms – like

- *parallel computing* – i. e. the development of systems that execute algorithms in parallel, mostly for purposed of enhanced execution speed

- *multi-processing* – i. e. a form of computing found in Operating Systems to execute different tasks quasi-simultaneously. Multi-threading is a form similar to multi-processing

- *networked computing* – this somehow puts the emphasis on the technological aspect of the connection of different systems over a network architecture, like ISO/OSI or TCP/IP. It is true, that networks play a very important role in studying distributed systems, but the main emphasis is on communication and design rather than the technology under it.

So, before trying to answer that are “Distributed Software Architectures”, let's first find out what makes distributed systems, in general, unique and interesting.

- *No shared memory*: As opposed to parallel computing where communication can take place over memory locations which can be read or written to by different parts of the software, Distributed Systems rely on different communication mechanisms, like Remote Method Calls or Message Passing techniques.

- *Activities can be carried out in parallel*: The implies the need for coordination and/or synchronization mechanisms, e. g. In the form of suitable protocols, transaction mechanisms, etc.

- *Failures are possible*: Actually, this is one of the more important issues in distributing software. By transparently deploying software components on different pieces of hardware, you should be able to come up with architectures that are more resilient to failure than a monolithic single-machine system.

- *Latency is an issue:* Since the transport of data over the network takes time, since the service used to respond to a remote request might take some time to even start executing, and furthermore since the execution of remote services might take some time, immediate responses are rarely the case.

- *Hardware and software components are heterogeneous*: This in fact is a severe issue, meaning that parts of a distributed syst6em can actually be implemented on quite incompatible hardware, different operating systems and different programming languages.

- *Security is an issue*: Networks are carriers for all kinds of attacks from viruses and malware over non-legitimate use of services to intrusion, spy attacks and disruption/destruction attacks. Measure must be taken to defend against these attacks.

- *The Topology is variable*: Topology meaning the way components are located and conneted to each other. The variability over time requires using particular designs that shield the topology from the application – like the Enterprise Service Bus (ESB), discovery and naming services (JNDI/LDAP) or trading services like in the CORBA/ORB architecture.

- *Scalability is an issue*: The fact that distributed architectures might put varying - generally increasing – load on one or more components puts the question of how this component reacts to high loads into focus. Means of scaling include upgrading the hardware or adding more hardware in forms of placing components on different machines, clustering, load balancing, etc.

## 3.2 And why are Distributed Systems interesting?

Software and hardware components can be joined together to facilitate one or more of the following principles:

- *Communication unit*: Transfer of information from one place to another (e. g. E-mail, ...)

- *Information unit*: Distribute data to multiple interested parties (e. g. WWW)

- *Data unit*: Store data in different places (load balancing, availability, security, ...)

- *Load unit*: Sudden increase or decrease in the number of requests for a service can be managed by distributing components over more than one computer system.

- *Performance unit*: Tasks can be divided into subtasks and assigned to different systems, resulting in faster response.

- *Maintenance unit*: Hardware and software problems can be monitored and removed from a central supervision system.

- *Functional unit*: Special hardware or software can be integrated into the application even if they are only available in a different location.

- *Interoperability unit*: Subsystems can be connected and integrated regardless of their internal workings if they share a certain communication infrastructure.

### 3.2.1 Transparency

Transparency describes the fact that certain aspects of a distributed system are not visible to the user or the application programmer. This results in the system being perceived as a coherent unit and not as a collection of separate components. The following aspects of a distributed system are generally supposed to be hidden:

1. *Location*: It is not necessary to know where a certain service is located, it's only important to find it.
2. *Access*: The use of a service should be uniform, independent from the internal construction or function of the service.
3. *Migration* and *relocation:* If a service can be moved or actually is moved to another location, this is not noticeable.
4. *Replication*: If a service is replicated over more than one component, this should not be noticeable.
5. *Concurrency*: The concurrent use of a service should not influence its behavior for a particular user.
6. *Failure*: Failure and recovery behavior of a service is not visible.

### 3.2.2 Consequences for Software Development

From a software point of view all this requires additional aspects that are not in the general focus of programming non-distributed applications:

| | | | | |
|---|---|---|---|---|
| Spatial separation | | Sequential Processes | | |
| Heterogeneity | | Concurrent Processes | | Synchronization |
| Openness | ▶ | Non-Determinism | ▶ | Programming more difficult |
| Complexity | | Distributed States | | |
| Security | | Communication | | Testing more difficult |

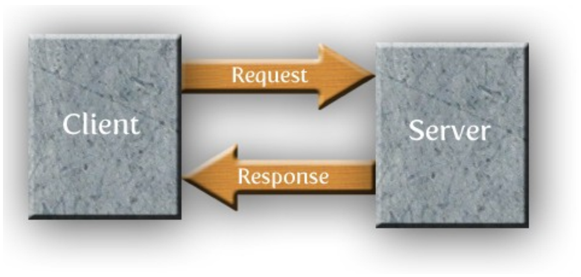
With all these constraints and problems the main focus on the practical level is to develop architectures that

- allow for a comfortable development process that frees the developer from low-level issues like communication, concurrency, security, etc.
- are stable with respect to failure, load problems, etc.
- allow for integration of heterogeneous software and hardware,
- are integrative in the sense that all (or almost all) aspects of an application can be developed without changing from one paradigm into another.

Here, we understand a software architecture as

1. A set of *basic components*,
2. *Design rules* to structure and combine systems to fulfill the criteria above,
3. *Recipes* to help the designer to build systems that behave as intended and
4. *Guidelines* about penalties for breaking the rules, trade-offs, ...

## 3.3 Generic Distributed Architectures



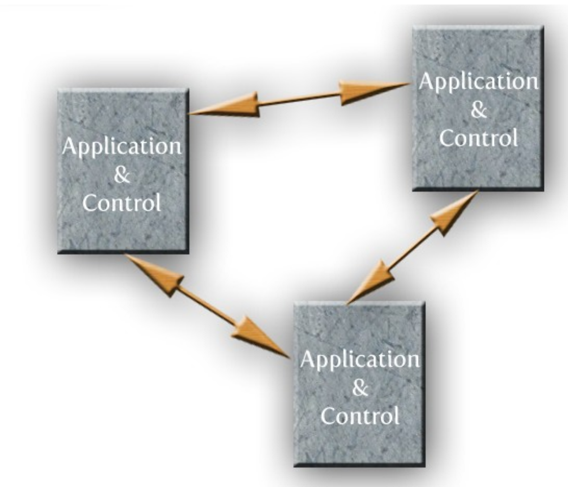
For distributed systems there are a couple of example architectures that are prominent in practice, the most important being of course the **Client-Server Architecture**.

One dedicated server (machine + software) accepts requests from one or more clients and sends back a response indicating the successful execution of the service that has been requested.

The distribution of an application under the client/server model can take different forms; we will look at this in the next section.

**Peer-to-Peer (P2P) Architectures** (see illustration below) treat no service node as special – they are all equivalent.

This means that P2P applications have to use a built-in control mechanism to execute properly. This mechanism controls the exchange of requests and responses, data transfer, transactions (if necessary), security, etc.



P2P networks are in use for message board systems, like *usenet*, where news servers are connected to each other propagating new messages into the network; recent uses are P2P **File Sharing** services.

P2P systems are considered more efficient than client/server systems, since load and storage requirements are distributed over a larger number of service nodes; they are also more robust, since the failure of a node could be compensated by other nodes (except for the lost storage, of course).

Besides these standard models, alternatives can be found in the overall network architecture in terms of size, organization and outreach.

- *Ad-hoc networks*, i. e. networks that integrate nodes upon their appearance. The components can then be located and respond to requests. For example, Jini [http://www.jini.org and http://incubator.apache.org/river/RIVER/index.html] is such an architecture.

- *Agent platforms*, viewing distributed software as a collection of interacting, autonomous **agents**, i. e. pieces of software equipped with (software) sensors and/or actuators, based on intent or goals, able to communicate in a given formal language. An example of such an architecture is the Open Agent Architecture [http://www.ai.sri.com/oaa/].

- *Mobile architectures*, based on cellphones, PDAs and other portable or wearable components. Usually organized in PANs (Personal Area Networks, based on USB/Firewire connections or Bluetooth piconets).

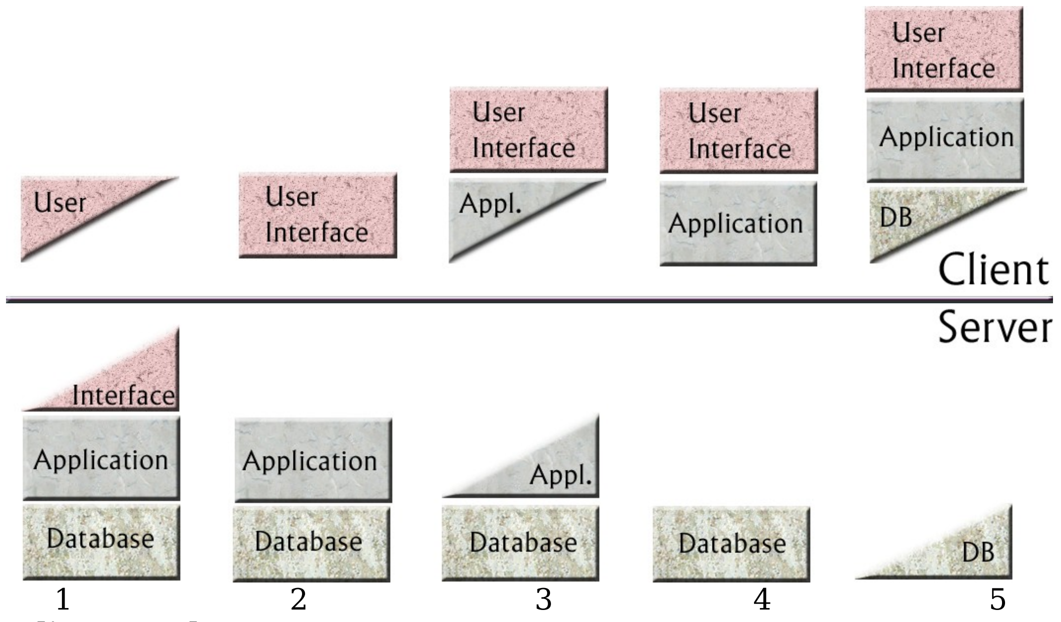
### 3.3.1 Clients/Servers in more detail

Since Client/Server architectures are the most common ones and since they are in the focus of this course, it's appropriate to get acquainted with a deeper insight into the many facets of them.

The first distinction is to look at the distribution of an application into a client/server setting, by assuming an application consisting of

- a User Interface that interacts with a human user and shields the implementation details by an interface component, like a GUI,
- an Application component which consists of the application logic and
- a Database component which is responsible for data persistence and related services.

A “local” (not distributed) application would locate all three components on one machine; bringing a second machine into play would result in one of the five scenarios shown below:

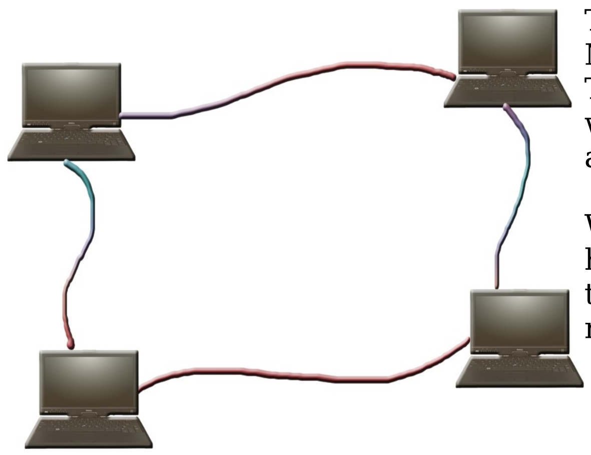


**Thin/Fat Clients and Servers**

1. A part of the User Interface resides on the client, some other parts as well as the whole application including the back-end database sits on the server. It's somewhat unusual to have parts of the UI on the server – an example could be an interface using XML/XSLT server-side rendering.
2. The complete UI sits on the client, everything else is on the server. You see this a lot of times, since the client software could be a browser and the transfer would be done over http, html etc.
3. The client contains a part of the application, as well. Also not unusual, considering client-side scripting mechanisms, like JavaScript and Ajax, which are distributed from the server to the client, but executed on the client to speed up the response time or take load off the server. The first three forms are generally referred to as **Thin Clients**.
4. and 5. are rare cases where a large part or almost all of the application resides on the client. Those cases are found in online gaming environments or complex windows-based applications. These clients are then called **Fat Clients**.

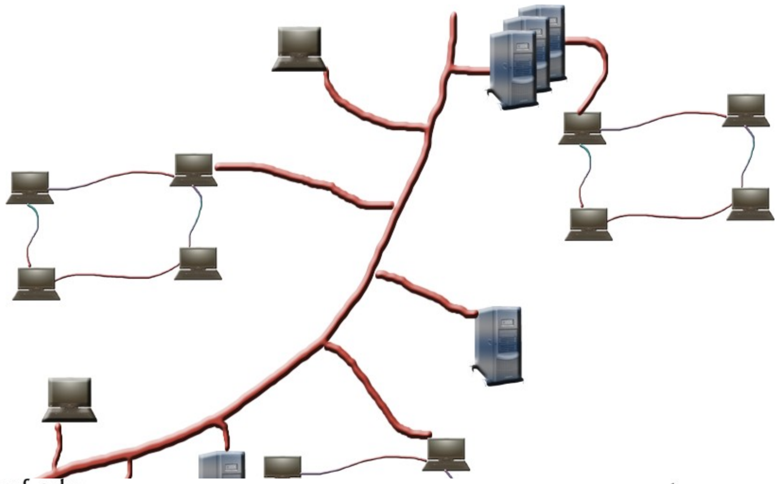
## 3.4 Distribution Infrastructure

Given a certain architecture, the problem is how to physically connect service nodes to each other. As usual, there are a couple of different ways, depending on the number, availability and spatial distribution of the components.



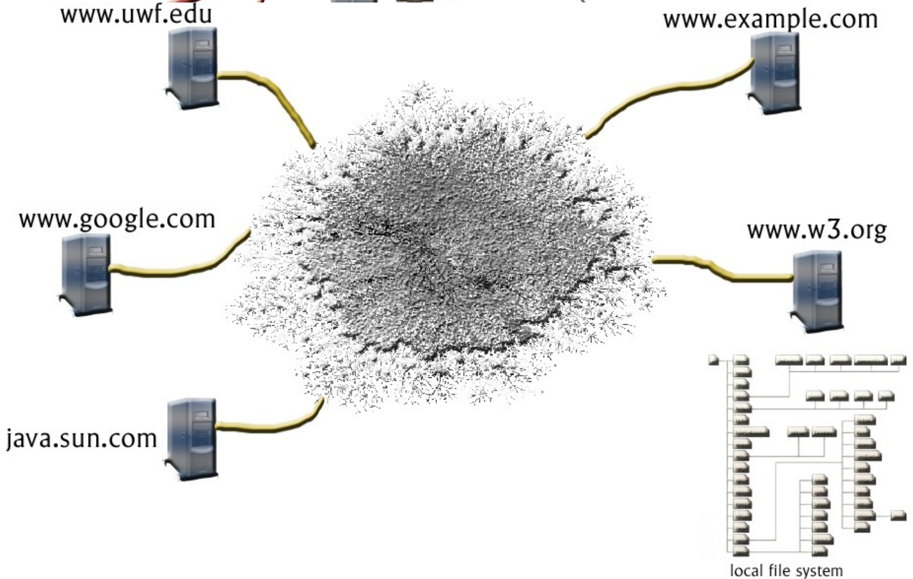
The simplest way would use a Local Area Network (LAN) to connect different machines. This is relatively simple and can be accomplished with little investment into hardware, software and know-how.

Wireless LANs are very common in private households and can be used as an infrastructure to demonstrate distributed architectures with relative ease.



Isolated computers and LANs can communicate over Wide Area Networks (WANs) in which backbones connect clients, servers and LANs over a common protocol.

The best-known carrier for a WAN nowadays is the **Internet** with its associated protocols, like http, ftp etc.



A sub-structure of the Internet, the World Wide

Web (WWW) can also be used as a carrier for distributing components.

Protocols are the same as for the Internet, but the main way to locate services is based on the idea of a Uniform Resource Locator (URL) and the use of HTML or XHTML as a coding method.

## 4. Software Development Blueprints

Starting Software Development in such an environment requires a couple of considerations that have to be added to the usual Software Engineering strategies:

- Choice of a set of technologies that supports the goals of a distributed application.
- Concerns about scalability, i. e. the behavior of the system under a unpredictable number of client requests. This is especially important for web-based applications which might become more popular than previously expected. A recent example is the rescaling of the Twitter platform [http://highscalability.com/scaling-twitter-making-twitter-10000-percent-faster].
- Concerns about security. Whenever computer systems are connected over a physical connection, new attack patterns may evolve. These patterns are based on well-know risks in networked systems together with attacks against the server infrastructure or attacks against the software itself which is easier accessible that in a local setting.
- Underlying development models. The Software Engineering discipline develops development strategies that guide the development of large software systems. In general, these models are based on waterfall or spiral disciplines, guided by a notational framework for the respective artifacts, like UML in the object-oriented case. This seems to be different especially in the area of web-based applications, where methods like “Agile Software Development” are the most popular ones.
- Development roles: As will be seen during this course, the development of distributed (or “Enterprise”) architectures is a process that requires managing a lot of different aspects [cf. http://java.sun.com/javaee/5/docs/tutorial/doc/bnaca.html], from product/tool providers, web designers, component developers, client developers, business logic developers etc. to application administrators. These roles can be filled by different people in the same organization or different organizations, they could be supported by sophisticated tools or they could be played by just one individual – this is more or less how we're going to do it in this course.

This course will take a relative simple attitude towards technologies. Starting from the basic architecture described in 1.1, especially the server-database connection, some choices could be

- CGI-based technology, using C or C++ and ODBC,
- PHP-based technology,

- .NET technology,
- Python + Django
- Java EE technology,

• CGI and PHP technology appear to be limited to server-side programming and seem to offer little support for issues of security, concurrency control and transaction security.

Furthermore, issues of transparency – as sketched above – as well as ease of development and reuse place a focus on object-oriented development, even though there is criticism against certain aspects of object orientation [http://www.agiledata.org/essays/impedanceMismatch.html] in settings like the ones this course aims at.

• ASP.NET offer basically everything needed for this course; the only point against it is that it's tied to a certain platform , but the mono project (http://www.go-mono.com) might lead a way out of this problem. It would also require to introduce a new programming language, like VBScript or C#. On the plus side the platform-dependence might give ASP.NET the edge in terms of efficiency.[1] Generally, ASP.NET is considered easier to learn and to support faster development cycles.

• We'll have a look at Python and Django

• Java  Enterprise Edition was the final choice, not because it's intrinsically superior to comparable technologies, but because it is built around the Java Core language and extends it towards the course objectives by APIs, like SAX, JDBC, JPA, JAX-RS, ... It also integrates with web development formalisms, like Java Server Pages and visual development frameworks, like Java Server Faces and facilitates component-based development in the form of Enterprise Java Beans (EJBs).

One of the interesting developments in Java EE is a the development of a variety of different frameworks, extensions, etc.[2] Depending on your point of view, you might consider this a great chance to find the development style that suits your projects best or you might be seriously appalled by the idea of spending a lot of project time to figure out, *what* to actually use.

Development environments and server software are available for J2EE developments and since the introduction of IDEs that make the notoriously difficult deployment process for distributed applications easier, a course can now concentrate on the important conceptual properties and the development strategies without getting lost in technical details.

1 Actually, I know of no “hard” figures comparing ASP.NET and J2EE. The use of native code in ASP.NET should lead to better execution times, but a lot of other factors could play a role in this.

2 JSF, Struts, Spring MVC, AppFuse, jdon, Dinamica, Expresso, Jspresso... and there are some more, see http://java-source.net/open-source/j2ee-frameworks

## 5. How should a course like this be structured?

The main problem in organizing a course like this is to spot the role a technology-oriented course has to play in the era of "information on the web" in general and Wikipedia in particular.

Yes, you can find almost all relevant information online and in fact the glossary (online) is organized as a collection of links into Wikipedia. But there's more to it than just that.

The general layout will be somewhat like this:

- The **lecture** will attempt to organize the relevant material in terms of what is important, in which sequence should the material be considered, how are these terms related and what are the conceptual issues involved.
- **Finger exercises** will let you walk through a technology on the "Hello World" level, i. e. you will do them to gain trust in it and see that it really works.
- **Readings** will let you study selected documents accompanying the course that are not directly covered in the lecture. These readings are essential and will confront you with the "lingo" of the respective area.
- **Projects** are more challenging and will make you apply the things you learned in the other phases of the course in a different context. Expect some serious work there.

### 5.1 What you should know

This course cannot possibly deliver all knowledge necessary to do practical work in this area; you should bring in working knowledge in the following areas:

- **Java programming**: to an intermediate level, i. e. knowledge about classes, inheritance, interfaces, packages, exceptions, collections, ...
  - Java packages
    - Description: http://en.wikipedia.org/wiki/Java_package
    - Use in Java: http://www.jarticles.com/package/package_eng.html
  - Java Archives (jar files): http://en.wikipedia.org/wiki/JAR_%28file_format%29
  - The Java Classloader: http://en.wikipedia.org/wiki/Classloader

- **(X)HTML, CSS and JavaScript**: since we-based applications require these techniques to manage the client side, some working knowledge is helpful.

- **Operating Systems**: basic knowledge about the workings, especially notions of processes and threads.
  - Java Threads: http://www.javaworld.com/javaworld/jw-04-1996/jw-04-threads.html

- **Tools:** Knowledge in linux and git.

- **Networks**: layered architectures, protocols, security, ...
  - OSI: http://en.wikipedia.org/wiki/OSI_model
  - TCP/IP: http://en.wikipedia.org/wiki/TCP/IP_model
  - Sockets
    - Description: http://en.wikipedia.org/wiki/Internet_socket
    - Sockets in Java: http://www.devarticles.com/c/a/Java/Socket-Programming-in-Java/