# Distributed Computing Systems Professor Ustijana Rechkoska-Shikoska, Ph.D.

Architectures



# Architectures

- Distributed systems are often complex pieces of software of which the components are by definition dispersed across multiple machines. To master their complexity, it is crucial that these systems are properly organized. There are different ways on how to view the organization of a distributed system, but an obvious one is to make a distinction between, on the one hand, the logical organization of the collection of software components, and on the other hand the actual physical realization.
- These software architectures tell us how the various software components are to be organized and how they should interact.
- An important goal of distributed systems is to separate applications from underlying platforms by providing a middleware layer. Adopting such a layer is an important architectural decision, and its main purpose is to provide distribution transparency.
- The final instantiation of a **software architecture** is also referred to as a **system architecture**.

# Architectural Styles

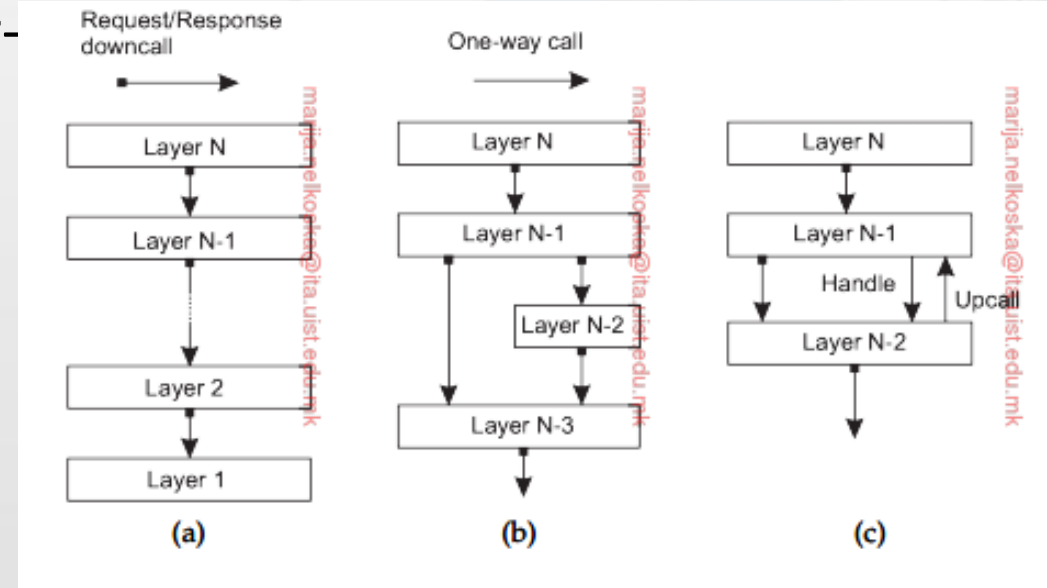
- Architectural style is very important. Such a style is formulated in terms of components:
  - the way that components are connected to each other;
  - The data exchanged between components;
  - How this elements are jointly configured into a system.
- A **component** is modular unit with well-defined required and provided interfaces that is replaceable within its environment.
  - Can be replaced if its interfaces remain untouched.
  - **Connector** (allows for the flow of control and data between components).

# Architectural Styles

- Several styles have by now been identified, of which the most important ones for distributed systems are:
  - **Layered architectures**
  - **Object-based architectures**
  - **Event-based architectures**

# Layered Architectures

- **Basic idea**: components are organized in **layered fashion** where a component $L_j$ can make a downcall to a component at a lower-level layer $L_i$ (with $i < j$) and generally expects a response. Only in exceptional cases will an upcall be made to a higher-



# Layered Communication protocols

- **Communication-protocol stacks**
  - In communication-protocol stacks, each layer implements one or several communication services allowing data to be sent from a destination to one or several targets. To this end, each layer offers an interface specifying the functions that can be called. In principle, the interface should completely hide the actual implementation of a service.
  - Another important concept of that is **(communication) protocol,** which describes the rules that parties will follow in order to exchange information.
    - consider a reliable, connection-oriented service, which is provided by many communication systems.
    - This kind of service is realized in the Internet by means of the **Transmission Control Protocol(TCP).**

# Application layering

- Considering that a large class of distributed applications is targeted toward supporting user or application access to databases, many people have advocated a distinction between three logical levels, essentially following a layered architectural style:
  - The application-interface level
  - The processing level
  - The data level
- In line with this layering, we see that applications can often be constructed from roughly three different pieces: a part that handles interaction with a user or some external application, a part that operates on a database or file system, and a middle part that generally contains the core functionality of the application. This middle part is logically placed at the processing level. In contrast to user interfaces and databases, there are not many aspects common to the processing level. Therefore, we shall give a number of examples to make this level clearer.

# Application layering contd.

- As a first example, consider an Internet search engine. Ignoring all the animated banners, images, and other fancy window dressing, the user interface of a search engine can be very simple: a user types in a string of keywords and is subsequently presented with a list of titles of Web pages. The back end is formed by a huge database of Web pages that have been perfected and indexed.
- As a second example, consider a decision support system for stock brokerage. Analogous to a search engine, such a system can be divided into the following three layers:
  - A front end implementing the user interface or offering a programming interface to external applications
  - A back end for accessing a database with the financial data
- As a last example, consider a typical desktop package, consisting of a word processor, a spreadsheet application, communication facilities, and so on.
- The data level contains the programs that maintain the actual data on which the applications operate. An important property of this level is that data are often **persistent** , that is, even if no application is running, data will be stored somewhere for next use. In its simplest form, the data level consists of a file system, but it is also common to use a full-fledged database.

# Object-based and service-oriented architectures

- A far more loose organization is followed in object-based architectures,

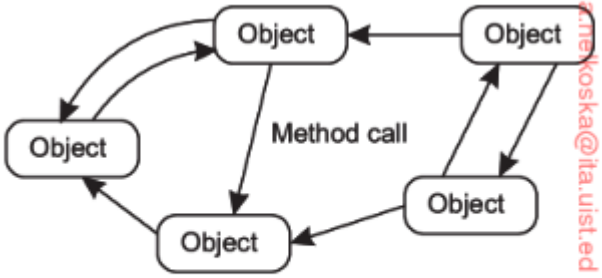



**Figure 2.5:** An object-based architectural style.

- In essence, each object corresponds to what we have defined as a component, and these components are connected through a procedure call mechanism. In the case of distributed systems, a procedure call can also take place over a network, that is, the calling object need not be executed on the same machine as the called object.
- Object-based architectures are attractive because they provide a natural way of encapsulating data (called an object's state) and the operations that can be performed on that data (which are referred to as an object's **methods)** into a single entity. The interface offered by an object conceals implementation details, essentially meaning that we, in principle, can consider an object completely independent of its environment.
- When a client **binds** to a distributed object, an implementation of the object's interface, called a **proxy** , is then loaded into the client's address space. A proxy is analogous to a client stub in **RPC systems.**

- The only thing it does is marshal method invocations into messages and un marshal reply messages to return the result of the method invocation to the client.

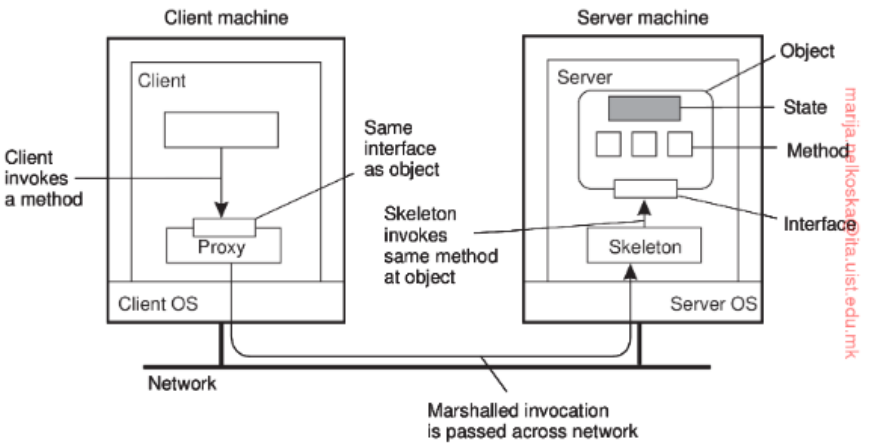



**Figure 2.6:** Common organization of a remote object with client-side proxy.

- A characteristic, but somewhat counterintuitive feature of most distributed objects is that their state is not distributed: it resides at a single machine. Only the interfaces implemented by the object are made available on other machines. Such objects are also referred to as remote objects.
- **Encapsulation** is the keyword here: the service as a whole is realized as a self-contained entity, although it can possibly make use of other services. By clearly separating various services such that they can operate independently, we are paving the road toward **service-oriented architectures** , generally abbreviated as **SOAs**.
- In a service-oriented architecture, a distributed application or system is essentially constructed as a composition of many different services. Not all of these services may belong to the same administrative organization.

# Resource-based architecture

- As an increasing number of services became available over the Web and the development of distributed systems through service composition became more important, researchers started to rethink the architecture of mostly Web-based distributed systems.
- Resources may be added or removed by (remote) applications, and likewise can be retrieved or modified.
- This approach has now been widely adopted for the Web and is known as Representational State Transfer (REST). There are four key characteristics of what are known as **Restful** architectures.
  - Resources are identified through a single naming scheme
  - All services offer the same interface, consisting of at most four operations,
  - Messages sent to or from a service are fully self-described
  - After executing an operation at a service, that component forgets everything about the caller

| Operation | Description |
|---|---|
| PUT | Create a new resource |
| GET | Retrieve the state of a resource in some representation |
| DELETE | Delete a resource |
| POST | Modify a resource by transferring a new state |

**Figure 2.7:** The four operations available in RESTful architectures.

- The RESTful architecture has become popular because of its simplicity.
- However, holy wars are being fought over whether RESTful services are better than where services are specified by means of service-specific interfaces.
- In particular, the simplicity of RESTful architectures can easily prohibit easy solutions to intricate communication schemes. One example is where distributed transactions are needed, which generally requires that services keep track of the state of execution.

# Publish-subscribe architectures

- As systems continue to grow and processes can more easily join or leave, it becomes important to have an architecture in which dependencies between processes become as loose as possible
- A large class of distributed systems have adopted an architecture in which there is a strong separation between **processing** and **coordination**.
  - **Coordination** encompasses the communication and cooperation between processes.

| | **Temporally coupled** | **Temporally decoupled** |
|---|---|---|
| **Referentially coupled** | Direct | Mailbox |
| **Referentially decoupled** | Event-based | Shared data space |

**Figure 2.9:** Examples of different forms of coordination.

# Publish-subscribe architectures contd.

- When processes are temporally and referentially coupled, coordination takes place in a direct way, referred to as **direct coordination.** The referential coupling generally appears in the form of explicit referencing in communication.
- A different type of coordination occurs when processes are temporally decoupled, but referentially coupled, which we refer to as **mailbox coordination**.
- The combination of referentially decoupled and temporally coupled systems form the group of models for **event-based coordination**. In referentially decoupled systems, processes do not know each other explicitly. The only thing a process can do is **publish a notification** describing the occurrence of an event

- A well-known coordination model is the combination of referentially and temporally decoupled processes, leading to what is known as a **shared data space**.
  - Shared data spaces are thus seen to implement an associative search mechanism for tuples.
- Shared data spaces are often combined with event-based coordination: a process subscribes to certain tuples by providing a search pattern; when a process inserts a tuple into the data space, matching subscribers are notified. In both cases, we are dealing with a **publish-subscribe architecture**, and indeed, the key characteristic feature is that processes have no explicit reference to each other.
  - The difference between a pure event-based architectural style, and that of a shared data space is shown in Figure 2.10.

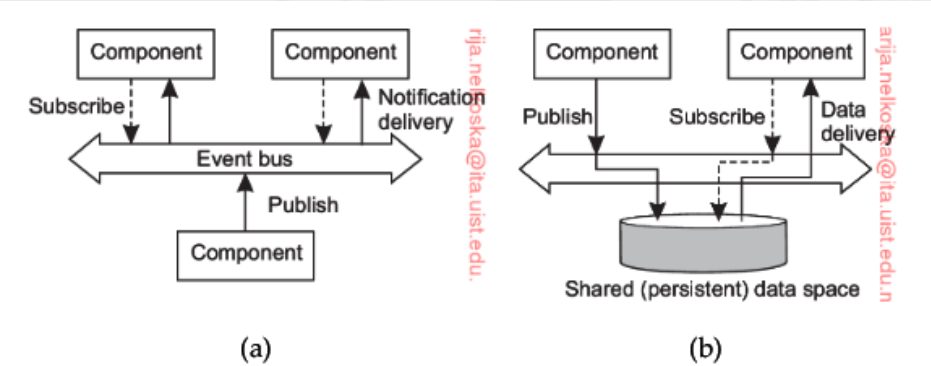



**Figure 2.10:** The (a) event-based and (b) shared data-space architectural style.

- Assume that events are described by a series of **attributes**. A notification describing an event is said to be **published** when it is made available for other processes to read. To that end, a **subscription** needs to be passed to the middleware, containing a description of the event that the subscriber is interested in. Such a description typically consists of some (**attribute, value**) pairs, which is common for so-called **topic-based publish-subscribe** systems.
- In content-based publish-subscribe systems a subscription may also consist of (attribute, range) pairs. In this case, the specified attribute is expected to take on values within a specified range. Descriptions can sometimes be given using all kinds of predicates formulated over the attributes, very similar in nature to SQL-like queries in the case of relational databases.

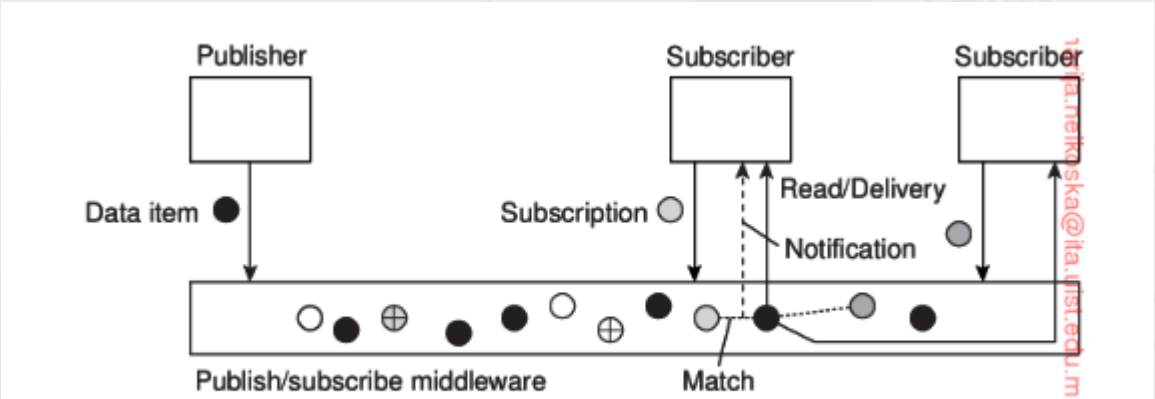



**Figure 2.12:** The principle of exchanging data items between publishers and subscribers.

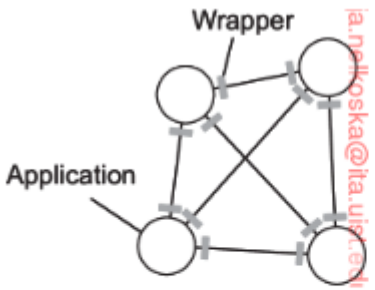
# Middleware organization

- Let us now zoom into the actual organization of middleware, that is, independent of the overall organization of a distributed system or application. In particular, there are two important types of ***design patterns*** that are often applied to the organization of middleware: wrappers and interceptors.
  - goal for middleware:  achieving openness
- It can be argued that the ultimate openness is achieved when we can compose middleware at runtime.

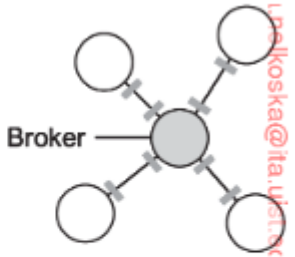
# Wrappers

- A wrapper or adapter is a special component that offers an interface acceptable to a client application, of which the functions are transformed into those available at the component. In essence, it solves the problem of incompatible interfaces. In the context of distributed systems wrappers are much more than simple interface transformers.
- An **object adapter** is a component that allows applications to invoke remote objects, although those objects may have been implemented as a combination of library functions operating on the tables of a relational database.

- Wrappers have always played an important role in extending systems with existing components. Extensibility, which is crucial for achieving openness, used to be addressed by adding wrappers as needed.
- Facilitating a reduction of the number of wrappers is typically done through middleware. One way of doing this is implementing a **so- called broker** , which is logically a centralized component that handles all the accesses between different applications. An often-used type is a **message broker.**




(a)




(b)

**Figure 2.13:** (a) Requiring each application to have a wrapper for each other application. (b) Reducing the number of wrappers by making use of a broker.

# Interceptors

- A **remote-object** invocation is carried out in three steps:
- An **interceptor** is nothing but a software construct that will break the usual flow of control and allow other (application specific) code to be executed.
- Interceptors are a primary means for adapting middleware to the specific needs of an application.
- The basic idea is simple: an object A can call a method that belongs to an object B, while the latter resides on a different machine than A.

# Interceptors contd.

- A remote-object invocation is carried out in three steps:
  - Object A is offered a local interface that is exactly the same as the inter- face offered by object B. A calls the method available in that interface.
  - The call by A is transformed into a generic object invocation, made possible through a general object-invocation interface offered by the middleware at the machine where A resides.
  - Finally, the generic object invocation is transformed into a message that is sent through the transport-level network interface as offered by A's local operating system.



# Interceptors contd.

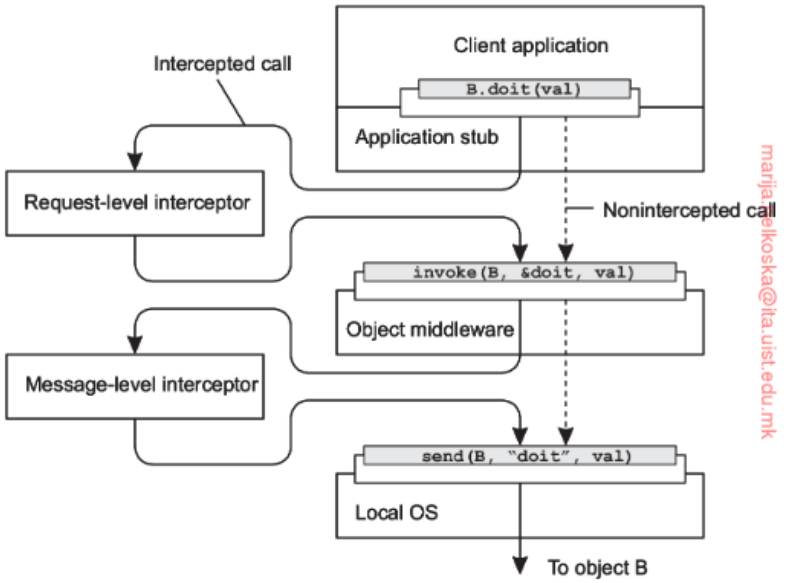



**Figure 2.14:** Using interceptors to handle remote-object invocations.



- In the end, a call to a remote object will have to be sent over the network. In practice, this means that the messaging interface as offered by the local operating system will need to be invoked. At that level, a **message-level interceptor** may assist in transferring the invocation to the target object.

# Modifiable middleware

- What wrappers and interceptors offer are means to extend and adapt the middleware. The need for adaptation comes from the fact that the environment in which distributed applications are executed changes continuously. Changes include those resulting from mobility, a strong variance in the **quality-of-service** of networks, failing hardware, and battery drainage, amongst others.
- These strong influences from the environment have brought many designers of middleware to consider the construction of *adaptive software*.

# Modifiable middleware contd.

- Component-based design focuses on supporting modifiability through composition. A system may either be configured statically at design time, or dynamically at runtime. The latter requires support for late binding, a technique that has been successfully applied in programming language environments, but also for operating systems where modules can be loaded and unloaded at will.
- The bottom line is that in order to accommodate dynamic changes to the software that makes up middleware, we need at least basic support to load and unload components at runtime. In addition, for each component explicit specifications of the interfaces it offers, as well the interfaces it requires, are needed. If state is maintained between calls to a component, then further special measures are needed. **By-and-large,** it should be clear that organizing middleware to be modifiable requires very special attention.

# Summary

- Distributed systems can be organized in many different ways. We can make a distinction between software architecture and system architecture. The latter considers where the components that constitute a distributed system are placed across the various machines. A keyword when talking about architectures is architectural style
- Important styles include: **layering, object-based styles, resource-based styles, and styles in which handling events are prominent**.
- In hybrid architectures, elements from centralized and decentralized organizations are combined. A centralized component is often used to handle initial requests, for example to redirect a client to a replica server, which, in turn, may be part of a peer-to-peer network as is the case in BitTorrent-based systems.