2. Basic Concepts

Distributed computing is the field of computer science that studies distributed systems.

> **Distributed System** A distributed system is one in which components located at networked computers communicate and coordinate their actions only by passing messages[1].

Distributed systems have always been important in computing, but the World Wide Web, pioneered by Tim Berners-Lee in the early 1990's[2], made such system widely available and accessible to a large audience.

Traditionally, distributed systems have been designed with two purposes in mind:

- To increase the computational power, that is, to allow complex calculations to be performed faster by utilizing more than a single computer.
- To share large amounts of data, that is, by storing data on a single central computer, a large number of other computers and thus users can access and modify common information.

It is fair to say today that the latter aspect is by far the most important, which the world wide web is a striking example of. The ability to share pictures and thoughts with friends on social media, the ability to handle orders and inventories globally for a company, the ability to book flights, hotel rooms, car rental, etc., all over the world are all examples of how shared data allows fast communication and coordination across people and continents.

Distributed systems can be organized in many ways but I will restrict myself to discuss client-server architectures.

> **Client-server architecture** Two components need to communicate, and they are independent of each other, even running in different processes or being distributed in different machines. The two components are not equal peers communicating with each other, but one of them is initiating the communication, asking for a service that the other provides. Furthermore, multiple components might request the same service provided by a single component. Thus, the

[1] George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair, "Distributed Systems – Concepts and Design, Fifth Edition", Pearson Education Limited, 2012.

[2] Tim Berners-Lee, "WWW: Past, present, and future.", *Computer* 29, Oct 1996.

component providing a service must be able to cope with numerous requests at any time, i.e. the component must scale well. On the other hand, the requesting components using one and the same service might deal differently with the results. This asymmetry between the components should be reflected in the architecture for the optimization of quality attributes such as performance, shared use of resources, and memory consumption.

The CLIENT-SERVER pattern distinguishes two kinds of components: clients and servers. The client requests information or services from a server. To do so it needs to know how to access the server, that is, it requires an ID or an address of the server and of course the server's interface. The server responds to the requests of the client, and processes each client request on its own. It does not know about the ID or address of the client before the interaction takes place. Clients are optimized for their application task, whereas servers are optimized for serving multiple clients[3].

In this part of the book, we will discuss the issues involved in developing distributed systems. Distributed computing is a large and complex subject as is evident from the discussion in the next section. I therefore have to limit myself to discussing core aspects that relate to those of the main themes of the book: the responsibility-centric perspective on designing object-oriented software, patterns for distributed computing, and how to develop flexible and reliable systems.

2.1 The Issues Involved

If you compare the definition of distributed systems with that of object-oriented programming by Budd (2002), they are actually strikingly similar:

Object-orientation (Responsibility): An object-oriented program is structured as a community of interacting agents called objects. Each object has a role to play. Each object provides a service or performs an action that is used by other members of the community[4].

If you replace the word "object" in Budd's definition with "components located at networked computers" they are nearly identical. Indeed, a web server is just an object that provides a service, namely to return web pages when your browser requests so. Essentially, your web browser is invoking

[3]Paris Avgeriou and Uwe Zdun, "Architectural patterns revisited - a pattern language", In 10th *European Conference on Pattern Languages of Programs* (EuroPlop), Irsee, 2005.
[4]Timothy Budd, "An Introduction to Object-Oriented Programming", Addison-Wesley, 2002.

the method of the web server object located at the computer given by the host name part of the URL.

However, there are some fundamental differences between invoking methods on objects in a standalone program, and invoking methods on remote objects.

The first problem in the programming model is that networked communication is implemented at a low level of abstraction, basically the only thing you can, is to send an array of bytes to a remote computer and receive an array of bytes from the network:

```java
void send(Object serverAddress, byte[] message);
byte[] receive();
```

Moreover, `send()` is an asynchronous function, which just puts the byte array into a buffer for the network to transmit and then returns immediately without waiting for a reply.

This is of course far from a high level object-oriented method call. Much of the following discussions, and the chapter on the Broker pattern in particular, deal with solutions to this problem.

Even when we do manage to bridge the gap between invoking methods on remote objects on one hand, and the low-level send/receive support on the other, there are still numerous issues to deal with.

- Remote objects do not share address space. Thus, references to local objects on my machine passed to a remote object on a remote machine are meaningless.
- Networks only transport bits and bytes with no understanding of classes, interfaces, or types which are fundamental structuring mechanisms in modern programming languages. In essence, a network is similar to a file system: just as a program's internal state needs to be converted to allow writing and reading from a disk—the internal state must be converted to allow sending it to a remote object.
- Remote objects may execute on different computing architectures—different CPUs and instruction sets. The problem is that different processors layout bits differently for for instance integers.
- Networks are slow. Invoking a method on a remote object is between 10 to 250 times slower than if the object is within the same Java virtual machine. You can find some numbers at the end of the chapter.
- Networks fail. This fundamental property means methods called on remote objects may never return.
- Remote computers may fail, thus the remote object we need to communicate with may simply not be executing.
- Networks are unsafe. Data transferred over networks can be picked up by computers and people that we are not aware of, that we can not trust.

- Remote objects execute concurrently. Thus, we face all the troubles of concurrent programming at the very same moment we program for distributed systems.
- If too many clients invoke methods on a server, it will become overloaded and respond slowly or even crash - typically because memory is exhausted.

These issues are complex, and several of them are architectural in nature, dealing with quality attributes like performance, availability, and security[5]. Treating each aspect in great detail is well beyond the scope of this book.

In this book, I will present architectural patterns and programming models that handles the fundamental aspects of invoking methods on remote objects. The emphasis is on "normal operations", that is, the context in which networks are stable, and the computers that host the remote objects are online and working correctly. This is of course the fundamental level to master before venturing into the more complex architectural issues of high performance and high availability distributed computing.

At the end of the chapter I will point towards central books that I have found helpful in handling the "failed operations" scenario, in which there are issues with slow network, broken servers, etc.

2.2 Elements of a Solution

Returning to our case study, TeleMed, we can state the challenges faced more concretely. On one hand we would like the source code that handles TeleMed story 1 to look something like this

```java
public void makeMeasurement() {
  TeleObservation teleObs;
  teleObs = bloodPressureMeterHardware.measure();
  TeleMed server = new RemoteTeleMedOnServer(...);
  String teleObsId = server.processAndStore(teleObs);
}
```

On the other hand, we only have operating system functions to send and receive byte arrays over a network.

Bridging this gap requires us to consider (at least) four aspects.

1. How to make two distributed objects, the one on the client side and the one on the server side, simulate a synchronous method call when they only have send/receive at their disposal ?

[5]Len Bass, Paul Clements, and Rick Kazman, "Software Architecture in Practice, 3rd ed.", Addison-Wesley, 2012.

2. How to convert structured objects, like the `TeleObservation`, into byte arrays suitable for transport on a network and back again?
3. How to keep, as best possible, our object-oriented programming model using method calls on objects?
4. How to locate the remote object to call?

These four challenges are answered by four programming techniques, that I will sketch below: The request-reply protocol, the marshalling technique, the *Proxy* pattern, and name services, which together form the backbone of the *Broker* pattern.

Request-Reply

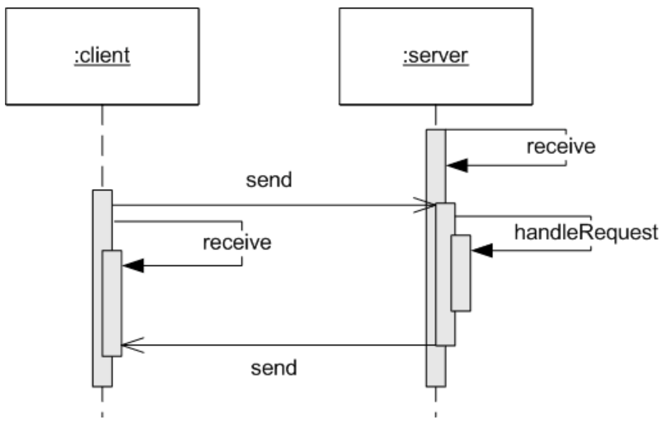
A normal method call at run-time in an object-oriented program is a so-called **synchronous method call** — the caller waits (blocks) until the object has computed an answer and returns the answer using the return statement.

In contrast a call to `send()` returns immediately once the operating system has sent the message on the network — which is not the same as the server having received the message, processed it, and computed an answer.

To simulate synchronous method calls over a network between a client object and a remote server object, a communication pattern called the **request-reply protocol** has evolved.

> **Request-Reply Protocol**. The request-reply protocol simulates a synchronous call between client and server objects by a pairwise exchange of messages, one forming the request message from client to server, and the second forming the reply message from the server back to the client. The client sends the request message, and waits/blocks until the reply message has been received.

In this protocol, the client makes the request by calling `send(server, request)` and then immediately calls `reply = receive()` which waits until it receives a reply back from the server. The server does the same but in the opposite order: It invokes `receive()` and awaits until some client sends it a message. Once received, it computes an answer, and then calls `send(client, reply)` to send the reply back. The UML sequence diagram in the figure below illustrates this protocol.



Request reply protocol.

The request-reply protocol is well known from browsing web pages. You enter some URL in the address field (that is, the server address and a path) and hit the 'Go' button, and then the browser sends the request, and waits until it receives a reply (usually showing some kind of "waiting for…" message or icon) which is the web page contents, after which it renders that page in the browser.

Marshalling

Methods take parameters that range from simple data types like an integer to complex object types like a HashMap or a class structure. Moreover, a remote object must be identified, and it must be told the exact identity of the method to call. All this data has to be converted into a format suitable for transmission over a network—that basically can only transport sequences of bytes.

This process is called **serialization** or **marshalling**.

> **Marshalling** is the process of taking a collection of structured data items and assembling them into a byte array suitable for transmission in a network message.

> **Unmarshalling** is the process of disassembling a byte array received in a network message to produce the equivalent collection of structured data items.

Obviously, the marshalling and unmarshalling processes have to agree on the format used, the *marshalling format*.

In the days before the world wide web changed everything, binary formats were often used, as they produce small packages on the network and are thus faster to transmit. In domains where speed and latency are very important, like online gaming, binary formats are still preferred. However, binary has its disadvantages. First, CPUs may vary in their encoding of bits into e.g. integers, which requires low level conversions.

Find resources on little-endian and big-endian byte ordering.

Second, they have the downside of not being easy to read for humans as well as less easy to adapt.

Extensible Markup Language (XML) became a widely used markup language to define open standard formats during the late 1990. It is defined by the World Wide Consortium (W3C) which is a main international standards organization. XML is both machine-readable as well as (relatively) human-readable. XML allows data values to be expressed in XML documents as *elements* using *tags* that are markup constructs that begin with the character ‹ and end with ›. Elements are surrounded by start-tags and end-tags.

As an example, consider a `TeleObservation` object with the values

```
teleObservation:
  patientId: ``251248-1234''
  systolic: 128.0
  diastolic: 76.0
```

Such an object could be represented in XML as

```
<TeleObservation>
  <patientId> 251248-1234 </patientId>
  <systolic> 128.0 </systolic>
  <diastolic> 76.0 </diastolic>
</TeleObservation>
```

XML has a lot of advantages, mainly due to the fact that it is a well known format, standardized, and supported by a large number of tools and software libraries. One of its main disadvantages is that it is verbose which makes it harder to read, and puts a lot of requirements on the hardware in terms of bandwidth to carry the large amounts of text.

JavaScript Object Notation (JSON) is an alternative whose main advantage is that it is much more compact. It is a human-readable open standard format for representing objects consisting of key-value pairs. It includes definition of data types, like strings, numbers, booleans, and arrays, and is thus closer to a description of objects than XML. As an example, the above TeleObservation object may look like

```
{
  patientId:  ``251248-1234'',
  systolic:   128.0,
  diastolic:  76.0
}
```

The examples in XML and JSON above represent a tele observation object, but I also need to encode the method to call—for instance the `TeleMed` interface has two methods, `processAndStore` and `getObservationsFor` . Thus, one plausible marshalling in JSON of an invocation of `processAndStore` would be:

```
{
  methodName : "processAndStore_method",
  parameters : [
    {
      patientId:  ``251248-1234'',
      systolic:   128.0,
      diastolic:  76.0
    }
  ]
}
```

Note that marshalling formats are fine for encoding simple data types like strings and numbers, as well as record types (that is: classes that only have simple data types as instance members), but they cannot easily represent references to other objects. I will return to this issue in Chapter Broker Part II.

Marshalling is a universal issue in distributed computing and is well supported by numerous open source libraries, see the sidebar.

JSON Libraries

JSON Libraries Marshalling and demarshalling JSON is supported by numerous libraries. Here I will demonstrate how Google Gson easily marshalls an object into JSON and back again. The code looks like

```
public static void main(String[] args) {
  Gson gson = new Gson();
  TeleObs a = new TeleObs("251248-1234", 128, 76);
  String aAsJson = gson.toJson(a);

  System.out.println("Original: " + a);
  System.out.println("As JSON: " + aAsJson);

  TeleObs b = gson.fromJson(aAsJson, TeleObs.class);
  System.out.println("Demarshalled: " + b);
```

```
    }
```

This code produces

```
Original: [id = 251248-1234: (128.0, 76.0)]
As JSON: {"patientId":"251248-1234","systolic":128.0,
            "diastolic":76.0}
Demarshalled: [id = 251248-1234: (128.0, 76.0)]
```

Note how the demarshalling method requires a type argument. Thereby Gson can itself create an object from the indicated type.

Proxy

With the request-reply protocol, we can convert a synchronous method call into a pair of calls: send data and wait until reply is received. However, this is obviously tedious to do at every place in the client code where we want to invoke methods on the remote object. Also we create a hard coupling from our domain code to the network implementation which lowers maintainability and testability.

Luckily, we have already a design pattern that solves this problem, namely the *Proxy* pattern (Chapter 25 in *Flexible, Reliable Software*).

Remember that a *Proxy* object implements the same interface as the **RealSubject** but is a placeholder that controls access to it. In our networked environment we would thus create a proxy on the client side, in which all methods are coded using the request-reply protocol to make the computation occur on the real object on the server.

In pseudo code for our TeleMed system, the TeleMedProxy's method `processAndStore` method may look like this:

```java
public class TeleMedProxy implements TeleMed, ClientProxy {

  public String processAndStore(TeleObservation teleObs) {
    byte[] requestMessage = marshall(teleObs);
    send(server, requestMessage);
    byte[] replyMessage = receive(); // Blocks until answer received
    String id = demarshall(replyMessage);
    return id;
  }
```

We achieve two important properties by using the *Proxy*. First, our domain code only interacts with `TeleMed` using its interface so there is no hard coupling to the network and distribution layer. Second, it supports dependency injection - for instance wrap the proxy by a *Decorator* which caches values locally, etc.

Note that every method in every proxy will follow a similar template: marshal, send, receive, demarshall. It can therefore be abstracted into a new role, the **Requestor**, which I will introduce in the *Broker* pattern in the next chapter.

Name Services

Finally, we need to refer uniquely to the remote object, that we want to call a method on. In ordinary Java code, you refer to an object through its object reference:

```java
TeleObservation teleObs =
  new TeleObservation("251248-1234", 126.0, 70.0);
String s = teleObs.toString();
```

Here `teleObs` refers to the object, and is actually a pointer to the part of the computer memory that holds the object's data: It is a memory address. Having this object reference allows us to invoke methods on the object, like the `teleObs.toString()` method invokation above.

However, if the object is located on some remote computer, not in our own computers memory, things are more complicated. First, we have to known *which* computer hosts the object, and next, we need some way to identify exactly which object on the remote computer, we need to invoke a method on. So basically a remote object reference needs to encode two pieces of information: the identity of the remote computer, and the identity of the object on the remote computer.

The solution requires two pieces: First, we have to define a **naming scheme** and second we need **name services**[6]. The naming scheme is a standardized way to identify/name remote references, and the name service is a dictionary/yellow pages/directory that allows us to translate from a name/identity of the remote object to the actual computer that hosts the object, as well as the actual reference to it on the remote computer.

The naming scheme must ensure that each name/identity of a remote object is unique. One scheme is a string with a hierarchical template, like file paths: "www.baerbak.com/telemed/251248-1234/2019-26-04-08-57-22"; which follows a template like "computername/system/person/timestamp". The benefit is that a human can actually read the string and get an impression of what object we are talking about. You will note that URLs follow this naming

[6]George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair, "Distributed Systems - Concepts and Design, Fifth Edition", Pearson Education Limited, 2012.

scheme. Another approach is simply to machine generate unique identities, like universally unique identifier (UUID).

The role of a name service is then to store and lookup names/identities of remote objects and translate them into their real (computer, reference) counter parts. This can be implemented in many ways.

At it simplest level, a name service is simply a Java Map data structure: Once a server receives a remote object name, it can fetch the relevant object reference:

```java
// Declaration of the name service
Map<String, TeleObs> nameService = ..

// Server has received a remote object name from client
TeleObs teleObs = nameService.get(remoteObjectName);
String s = teleObs.toString();
```

Of course, this simple example above assumes that the client already knows which remote computer, the object is located on. This is actually a quite feasible case, as I will discuss below.

In the more general case, name services are standalone services/processes.

This is similar to DNS servers on the internet, which allow symbolic names, like www.baerbak.com[7] to be looked up to get their actual physical IP address, like 87.238.248.136. The great advantage of a registry is that if location changes (say, my web server moves to a new IP address), you just change in the registry and everybody can still find it.

Java RMI and similar remote method invocation frameworks provides a separately running service, the **registry**, where you can bind a name/identity with a remote object reference.

Frameworks based on HTTP can be said to have solved the look up problem by reusing the DNS look up servers. Thus usually they simply know the name of the server with the object by using the URL of the server. Thus instead of such libraries having their own built-in registry, they use hard coded server names and let the DNS system handle the problem of finding the correct server to call.

2.3 Tying Things Together

The *Broker* pattern discussed in the next chapter can be said to combine the elements above into a cohesive whole that provides a strong programming model for building client-server architectures.

[7]www.baerbak.com

It uses the *Proxy* pattern on the client side to 'mimic' the normal interface oriented programming model so client objects communicate with the remote object in an almost normal way. The methods of the client side proxy are all following a template that uses marshalling to encode parameters, object id, and method name into a byte array; uses the low level send()/receive() methods to implement a request-reply protocol call to a remote server; and then demarshalls the returned byte array into a valid return value.

Similarly the server side uses demarshalling and marshalling to decode the received byte array into information on which object's method to call with which parameters; does the call (often called the "up call") to the real implementation of the object; and finally marshals the return value into a byte array to be returned to the client.

2.4 Summary of Key Concepts

Distributed systems are computer systems that provides services to their users by having multiple computers communicating and coordinating by exchanging messages over a network. One of the main advantages of distributed systems is the ability to access large amounts of shared data.

A common way to structure distributed systems in many domains is the **client-server architecture** in which a large number of *clients* sends requests to a central *server* which processes the request and return an answer to the clients. Typically, the server maintains a large data set, that clients create, read, and update.

Such distributed systems resemble object-oriented system in that client objects invoke methods on server objects, and thus supporting *remote method calls* is a natural way to express message passing in distributed systems.

However, remote method calls are fundamentally different from ordinary methods calls in a single program. Key differences are

- Networks can only send and receive byte arrays
- Client and server objects are executing concurrently
- Network send() is asynchronous, thus the client will not await the answer being returned from the server
- Object references are meaningless across computer boundaries
- Network communication is slower, may fail, and transmissions can be intercepted by unwanted processes

To mitigate these differences, we can apply methods and practices

- The Request-Reply protocol defines a template algorithms in which a client will send a request to the server, and then wait/block until an answer has been returned, thereby mimicking normal OO method calls.

- Marshalling is a technique to translate normal programming types, like strings, integers, and booleans, into byte arrays and back. Thereby we can convert back and forth between the OO programming level and the network level.
- Proxy is a pattern that *provide a surrogate for another object* and we can use this pattern to implement a surrogate for the server object on the client side, having the same interface, but whose implementation uses request-reply and marshalling to communicate in an OO method call manner with the real object on the server.
- Name service is the role of a storage that can bind object names or identities to the real objects. Thereby we can transmit object identities instead of object references between clients and servers and use the name service to resolve which actual object to communicate with.

The aspects of failure handling, security, and performance are outside the scope of this book.

Distributed Systems—Concepts and Design by Colouris et. al[8] is comprehensive treatment of distributed system.

2.5 Review Questions

Explain what a distributed system is? Explain what a client-server architecture is - and what clients and servers can and can not.

What are the challenges of implementing distributed system? How does the techniques of *request-reply*, *marshalling*, *proxy*, and *name service* handle aspects of a solution to these challenges? Explain each of the four techniques in detail.

Performance of Network Calls

Network calls are slower than ordinary method calls, but the slowdown is highly affected by how geographically separated the client and server are. Below I give some numbers that I have measured. I used the TeleMed system and did five times 10,000 uploads of a single blood pressure measurement for each experiment. The time measured below is in milliseconds for 10,000 uploads. The tabel outlines a configuration, the average run time for the 5 times 10,000 uploads, the maximum time for any of the 5 runs, and finally the average slow down factor, using local call as baseline. Thus, remote calls to another machine on the same network switch was a factor 12.7 times slower than doing in-memory method calls.

[8]George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair, "Distributed Systems – Concepts and Design, Fifth Edition", Pearson Education Limited, 2012.

Configuration	**Average time (ms)**	**Max time (ms)**	**Factor**
Local call	1,796	3,366	1.0
Localhost	9,731	12,806	5.4
Docker	17,091	35,873	9.5
On switch	19,690	22,025	11.0
Frankurt	494,966	513,411	275.6

I ran the server in a virtual machine (4GB memory, 2 processers, Lubuntu Linux) on a small business server (4 core 3.1 GHz Xeon E3) running VMWare ESXi 6.5. The Frankfurt case ran the server in a Docker container.

The configurations were:

- Local call: The TeleMed client made local method calls to "the server object" within the same Java program.
- Localhost: The TeleMed client made remote socket calls to a TeleMed server hosted on the same machine.
- Docker: The TeleMed server ran in a docker container on the same machine as the TeleMed client.
- On switch: The TeleMed client ran on a separate machine, connected to the same network switch as the TeleMed server.
- Frankfurt: The TeleMed server ran on a DigitalOcean virtual machine located in Frankfurt (Germany), while the TeleMed client was located in Aarhus (Denmark).

As is evident, network calls are slower but as long as your calls are between machines in close proximity, the slowdown is not overwhelming. However, once you get to production environments the slowdown is much higher: a factor of 275 compare to local calls, and a factor of 11 times slower than if the server was on the same network switch.

Disclaimer: These values are from a small experiment, and there are different characteristics on the hardware of the machines that run clients and server, so take the values with a grain of salt…