Distributed Systems And SOAs

Ivan Lanese

(Original slides from Fabrizio Montesi)

Distributed Systems

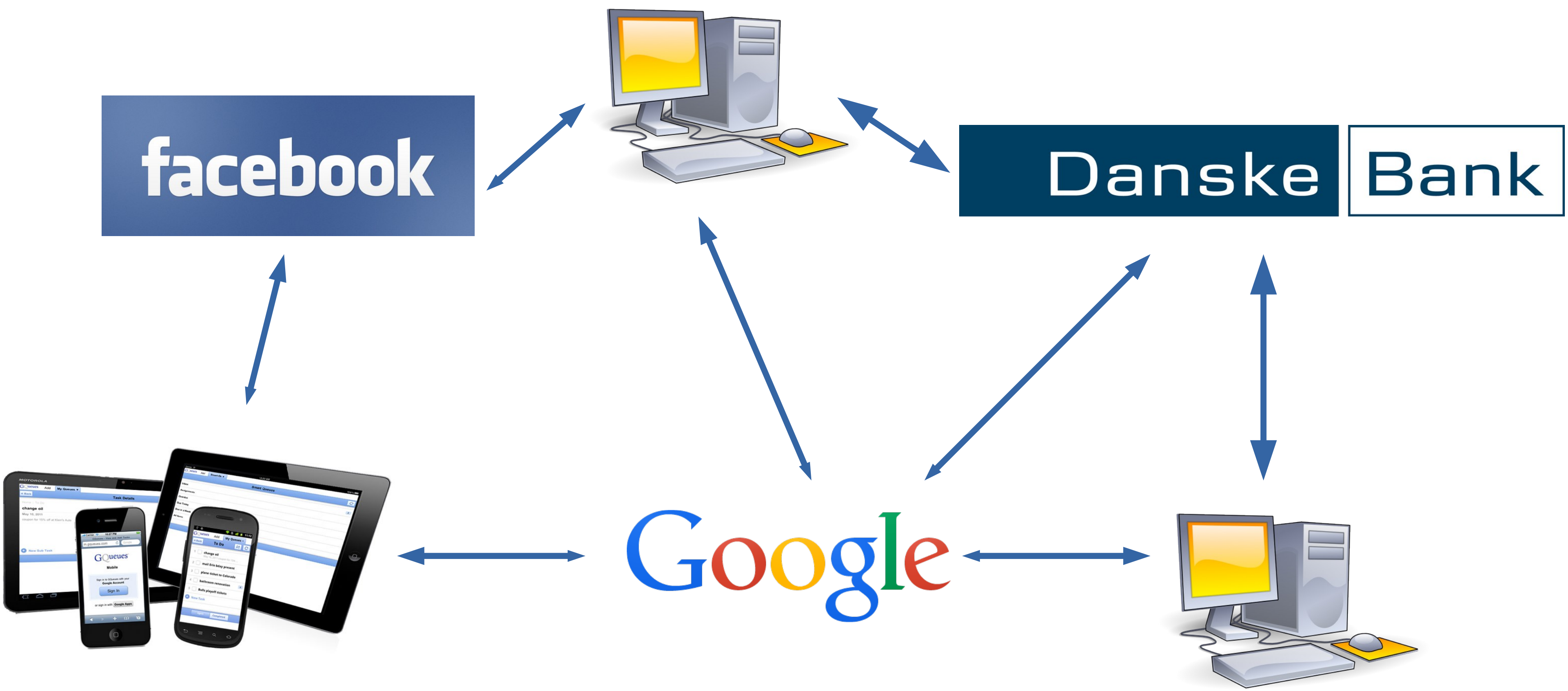
- **Distributed system:**
  a network of endpoints that communicate by exchanging messages

- The natural environment to execute business processes

- But not only! Let's see some examples...

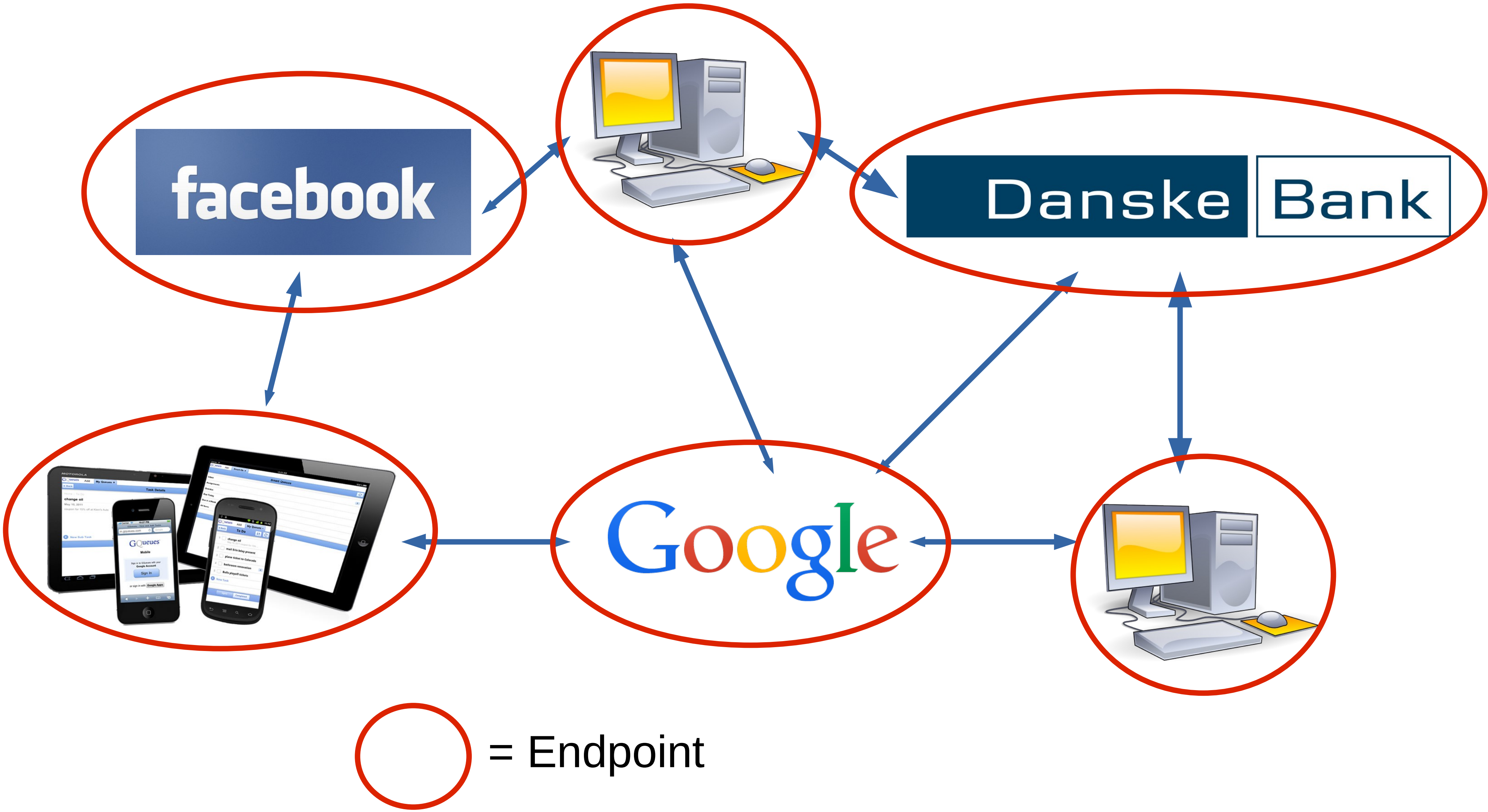
The Internet

- The Internet is a distributed system:



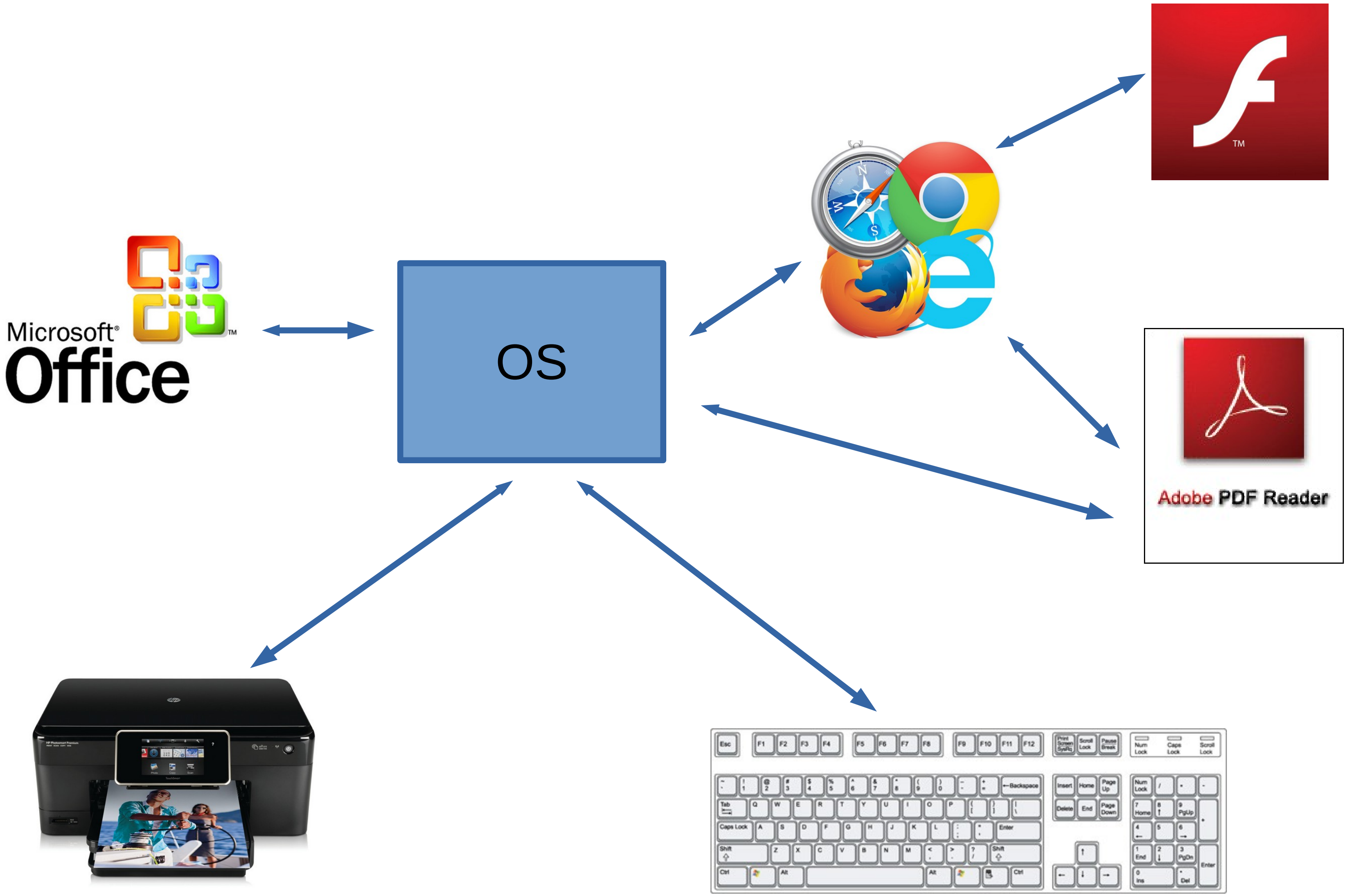
The Internet

- The Internet is a distributed system:



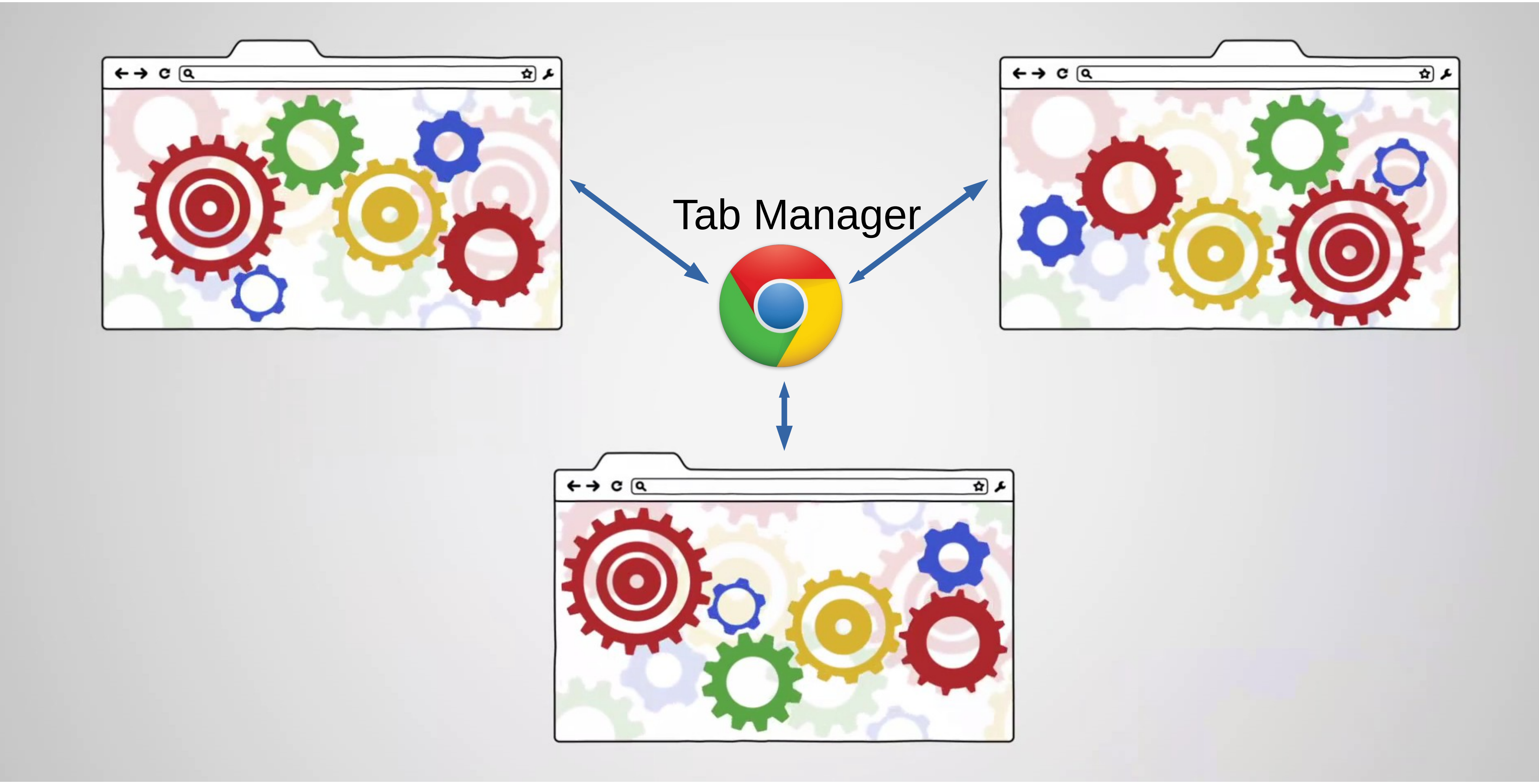
Your Computer

- The OS and apps in your computer (or phone):



Google Chrome

- Even applications can be distributed systems. Google Chrome:



Complexity

- Distributed systems are big! Some numbers:

System	Number of Endpoints
My computer	260
A house	Hundreds
A company	Thousands (or millions)
The Internet	At least 40 billions

Endpoint Programming

- How do we program all these endpoints?

- We write a program for each.

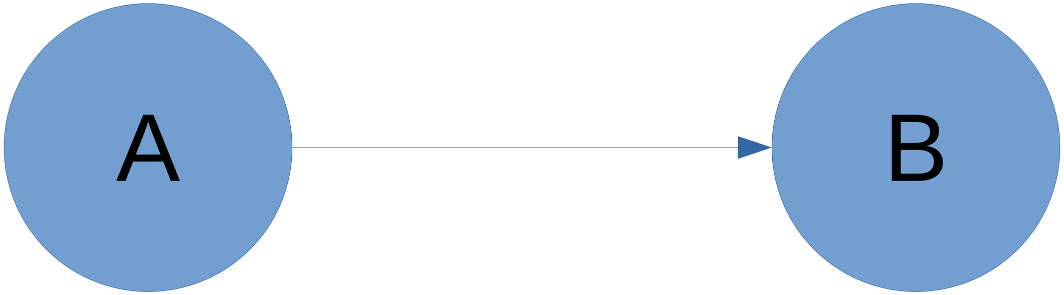
- Programs interact by sending and receiving messages.

Not so easy...

- Programming distributed systems is usually harder than programming non distributed ones.

- Some problems are:
    - handling communications;
    - handling heterogeneity;
    - handling faults;
    - handling the evolution of systems.

Not so easy... - Communications



- The basic feature for any distributed system.
- Let us look at how Java does it. We open a TCP/IP socket and we send some data:

```
SocketChannel socketChannel = SocketChannel.open();
socketChannel.connect(new InetSocketAddress("http://someurl.com", 80));

Buffer buffer = . . .; // Create a byte buffer with data to be sent.

while( buffer.hasRemaining() ) {
    socketChannel.write( buffer );
}
```

Not so easy... - Communications

- That is not good Java code.
- We need to remember to:
  - handle exceptions;
  - close the channel.
- Better version:

```
SocketChannel socketChannel = SocketChannel.open();
try {
  socketChannel.connect(new InetSocketAddress("http://someurl.com", 80));
  Buffer buffer = . . .; // Create a byte buffer with data to be sent.

  while( buffer.hasRemaining() ) {
    socketChannel.write( buffer );
  }
}
catch( UnresolvedAddressException e ) { . . . }
catch( SecurityException e ) { . . . }
/* . . . many catches later . . . */
catch( IOException e ) { . . . }
finally { channel.close(); }
```

Not so easy... - Communications

- Phew...! Are we done?
- No! The server-side code can be much more complicated!

Not so easy... - Communications

- A “simple” example that listens to events on an existing channel... and does not even handle exceptions!

```
Selector selector = Selector.open();

channel.configureBlocking(false); //ensures the channel can be used

SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
//registers the channel for reading

while(true) {
  int readyChannels = selector.select();
  if(readyChannels == 0) continue;

  Set<SelectionKey> selectedKeys = selector.selectedKeys();
  Iterator<SelectionKey> keyIterator = selectedKeys.iterator();
  while(keyIterator.hasNext()) {
    SelectionKey key = keyIterator.next();
    if(key.isAcceptable()) {
        // a connection can be accepted.
    } else if (key.isConnectable()) {
        // a connection was established with a remote client.
    } else if (key.isReadable()) {
        // a channel is ready for reading
    } else if (key.isWritable()) {
        // a channel is ready for writing
    }

    keyIterator.remove();
} }
```

Not so easy... - Heterogeneity

- In the real world, distributed systems can be heterogeneous.

- Different applications that are part of the same system could...
  - use different **communication mediums** (Bluetooth? TCP/IP?, …);
  - use different **data protocols** (HTTP? SOAP? X11?);
  - use different **versions** of the same data protocol (SOAP 1.1? 1.2?);
  - and so on...

Not so easy... - Faults

- Applications in a distributed system can perform a *distributed transaction*.

- Example:
  - a client asks a store to buy some music;
  - the store opens a request for handling a payment on a bank;
  - the client sends his credentials to the bank for closing the payment;
  - the store sends the goods to the client.

- Looks good, but a lot of things may go **wrong**, for instance:
  - the store (or the bank) could be offline;
  - the client may not have enough money in his bank account;
  - the store may encounter a problem in sending the goods.

Not so easy... - Evolutions

- Distributed systems usually *evolve over time*.
- Each application could be made by a different company.
- A company may update its application.
- Again, many possible pitfalls:
  - the updated version may use a **new data protocol**, unsupported by the clients;
  - the updated version may have a **different interface**, e.g. before it took an integer as a parameter for a functionality, now a string;
  - the updated version may have a **different behaviour**, e.g. before it did not require clients to log in, now it does.

How to simplify?

- Things can be made easier by hiding the low-level details
  - Finding suitable abstractions hiding these details
  - Programming using these abstractions

- Two main approaches:
  - make a library/tool/framework for an existing programming language;
    - e.g., zeep or suds in python, JAX-WS in Java, ...
  - make a new programming language
    - many programming languages (e.g., Erlang, Go, ...) now provide abstractions for communication (but not for heterogeneity...)

Language example: Erlang

- Erlang processes can easily send messages to remote Erlang processes.

- We can spawn a local process and send a message to it:

```
Pid = spawn(pong, []),
Pid!pong.
```

- We can spawn a remote process and send a message to it:

```
Pid = spawn(distrPong, pong, []),
Pid!pong.
```

- When we have the pid of the target process, remote communication is identical to local communication
- Easy since communication is between Erlang processes

Service-oriented Computing (SOC)

- A design paradigm for **distributed systems**.
- A **service-oriented** system is a network of **services**.
- Services communicate through message passing.

- Messages are tagged with **operations** (similar to method names in OO).
- Services are typed with **interfaces**, which define **message data types** for operations.
- Original reference technology: Web Services.
  - Based on XML;
  - WS-BPEL (BPEL for short) for programming composition.

Why SOC? A few reasons...

- Everybody was using custom solutions for distributed computing.
- We need to **integrate** different applications, possibly belonging to different companies
  - They rely on existing software.
  - Programs using different data protocols cannot interact.
- We need support for more **dynamicity**.
  - **Service Discovery:** we can discover where services are located at runtime.
- We need support for **structured interactions**.
  - Many web applications implement logical orderings between actions.
  - Example: in a newspaper web portal, a user may need to log in *before* reading the news.

SOA Basics

- A Service-Oriented Architecture (SOA) is composed by **services**.
- A **service** is an application that offers **operations**.
- A service can invoke another service by sending messages to one of its **operations**.
- Recalling Object-oriented programming:

Service-oriented	Object-oriented
Services	Objects
Operations	Methods

- Services can be based on different technologies, languages, …
- Services can belong to different organizations
- Services need to rely on standards (XML, SOAP, WSDL, …) to ensure interoperability

The SOA Dream

- Creating new applications and business processes by **integrating** existing services which are:
  - Independent
  - Available on the net
  - Provided by different organizations
  - **Dynamically discovered**

- The SOA Dream is not yet reality
  - And there is now less emphasis on it
  - There is instead increasing emphasis on scalability and evolvability

- However SOAs have produced relevant results
  - Standardizing many aspects of interaction
  - Enabling integration and interoperation of systems provided by different companies

The SOA descendants

- SOC is evolving, and many of the original technologies are less and less used
- The main concepts are still valid, combined with new ones and giving rise to new technologies
- REST (Representational State Transfer) services
  - Software architecture based on resources accessed via HTTP and stateless services
  - Emphasis on simplicity and scalability
- Microservices
  - Software architecture based on many small services which can be independently deployed and scaled
  - Relies on container technologies such as Docker
  - Emphasis on scalability and evolvability (continuous deployment, continuous integration)