Microservice Programming

2

Fabrizio Montesi
<fmontesi@imada.sdu.dk>

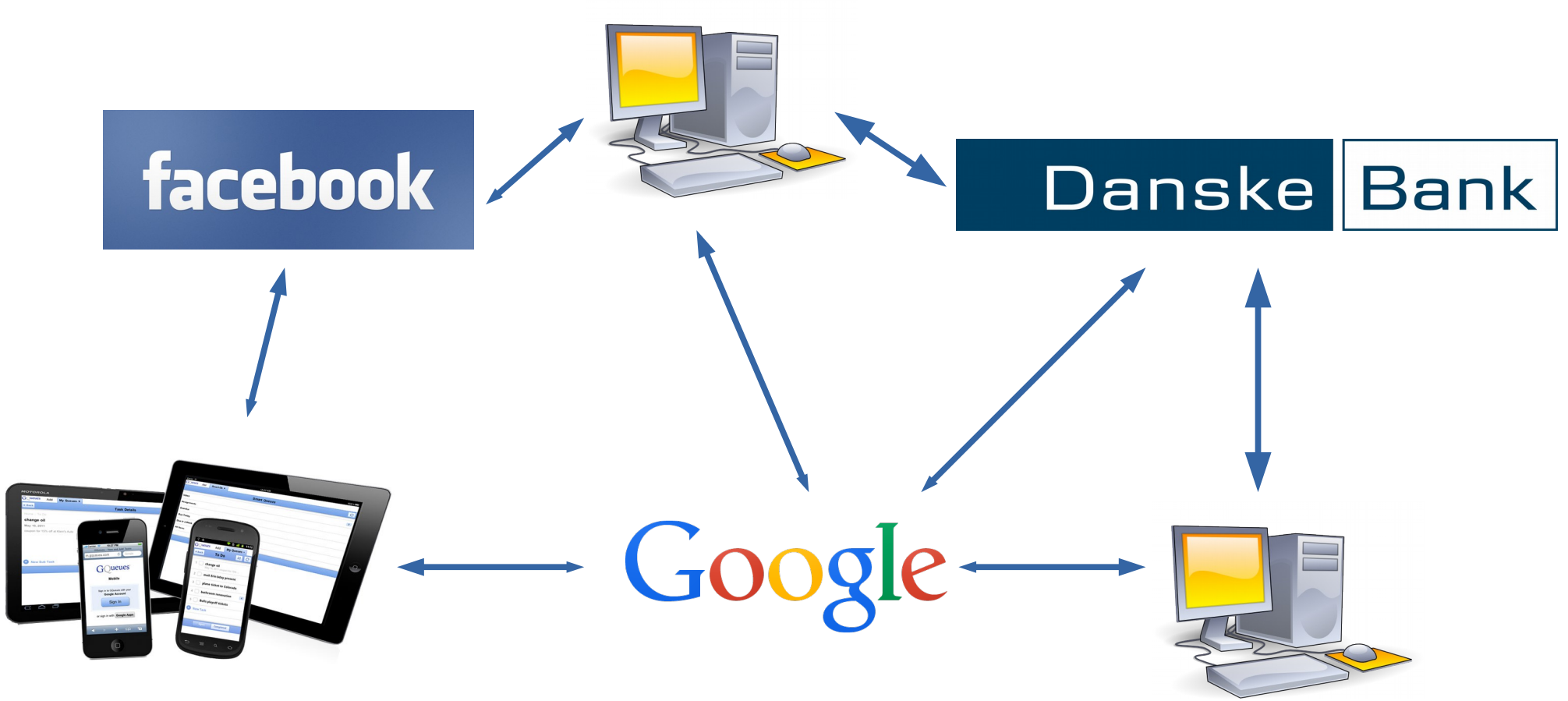
Distributed Systems

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

- Widespread! Let's see some examples...

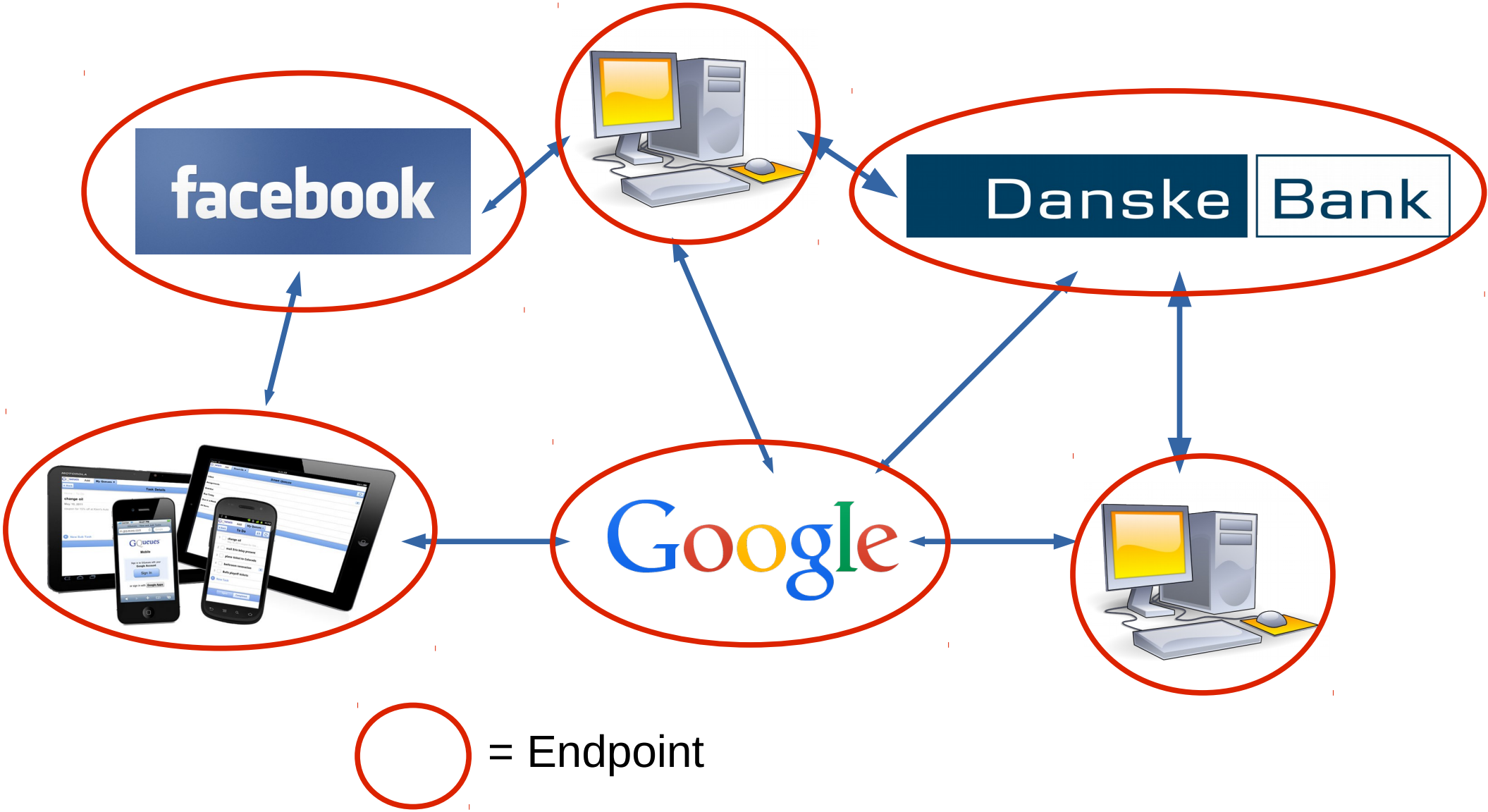
The Internet

- The Internet is a distributed system:



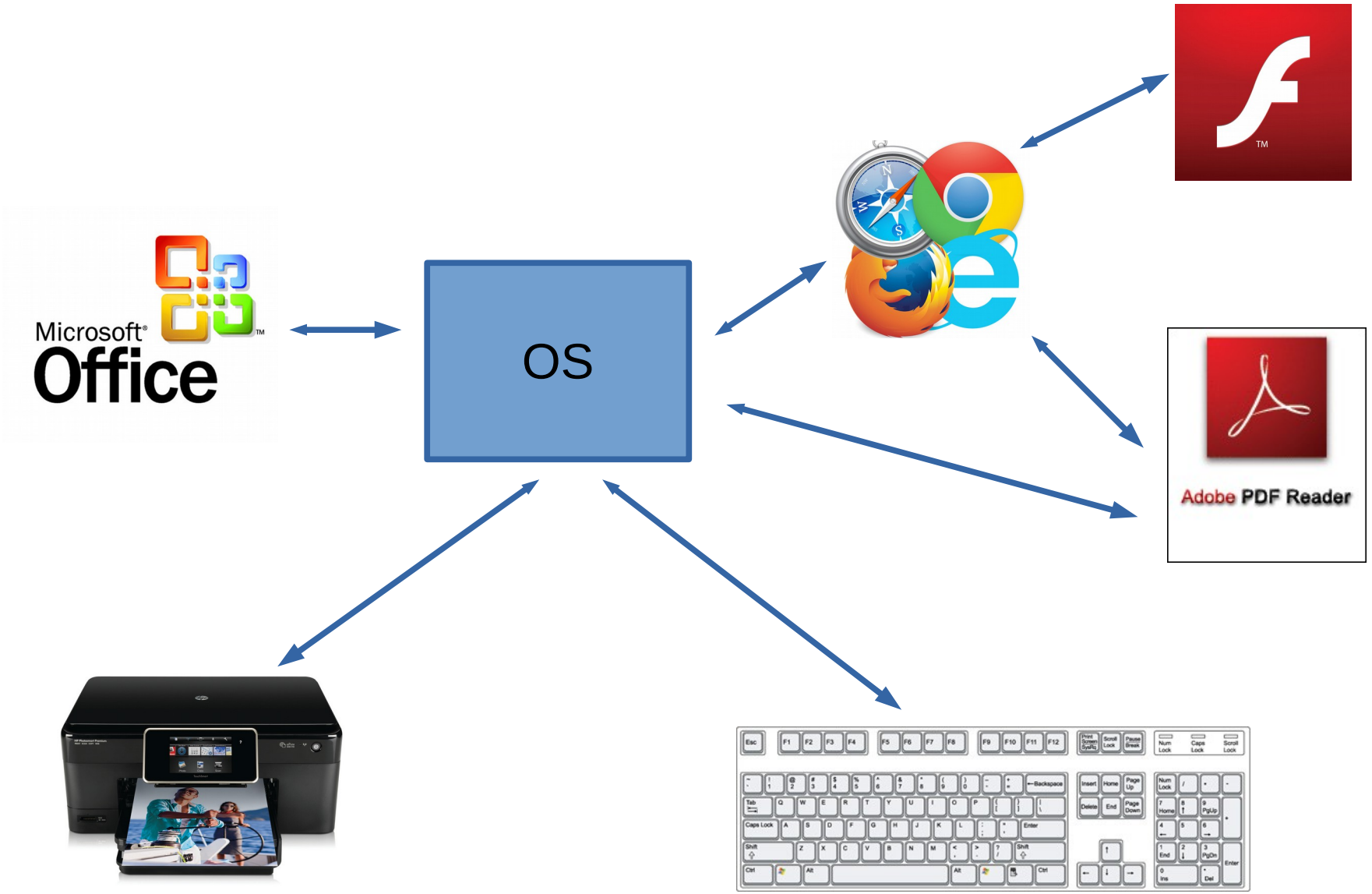
The Internet

- The Internet is a distributed system:



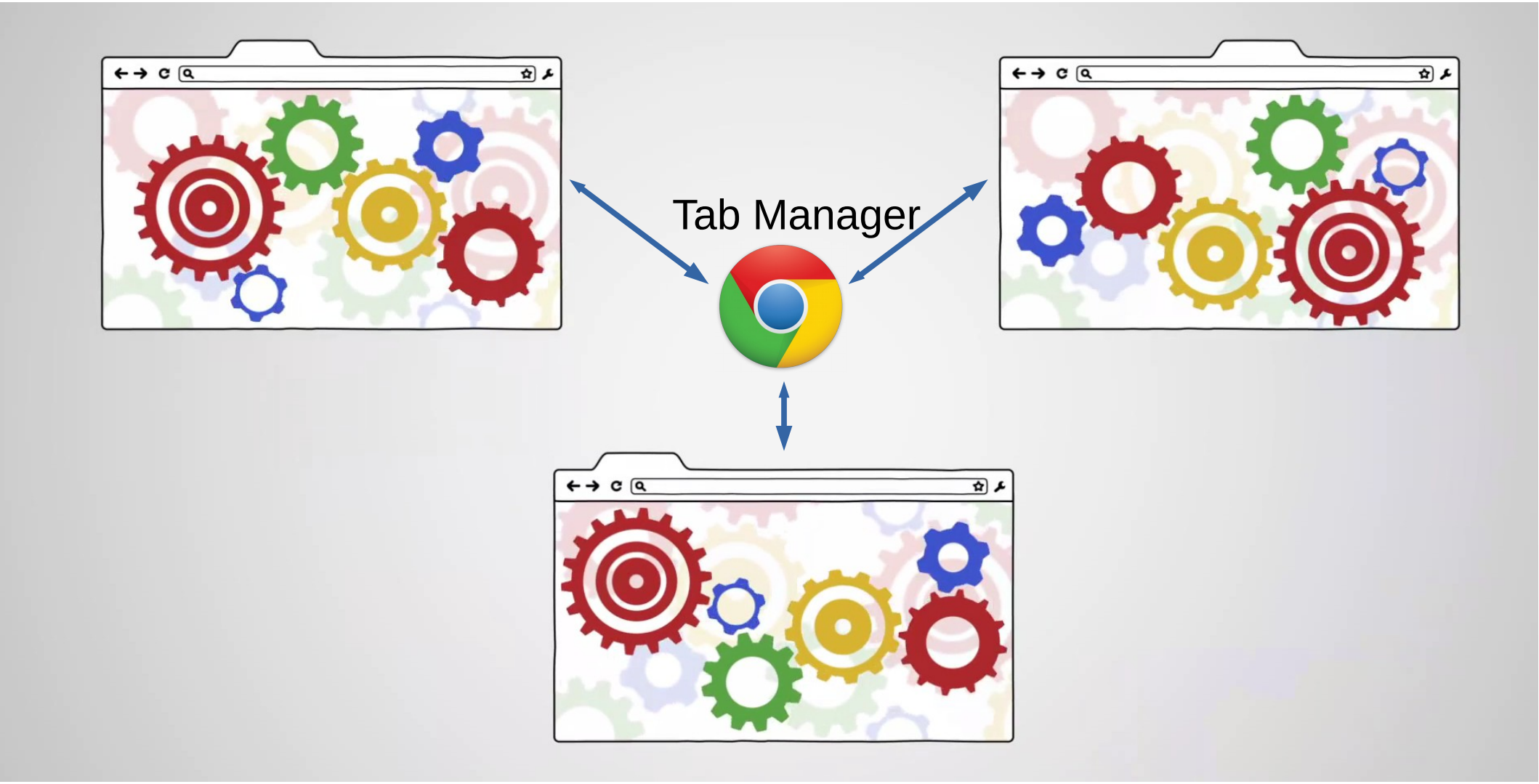
Your Computer

- 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	160
A house	Hundreds
A company	Thousands (or millions)
The Internet	At least 20 billions

Endpoint Programming

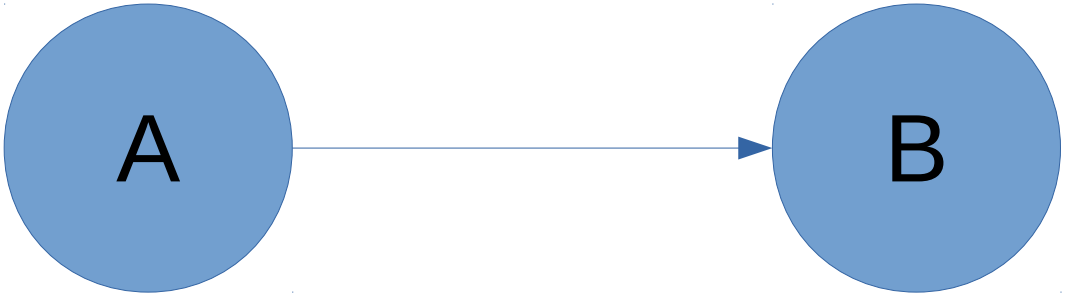
- How do we program all these endpoints?
- We write a program for each.
- Programs interact by sending and receiving messages.

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() ) {
    channel.write( buffer );
}
```

Not so easy... - Communications

- That is not good Java code.
- We need to remember to:
  - handle eventual exceptions;
  - remember to 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() ) {
    channel.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!
- Also: what if we want to reuse previously opened channels?

Not so easy... - Communications

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

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

channel.configureBlocking(false);

SelectionKey key = channel.register(selector, SelectionKey.OP_READ);

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 was accepted by a ServerSocketChannel.
    } else if (key.isConnectable()) {
        // a connection was established with a remote server.
    } 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. first it took an integer as a parameter for a functionality, now a string;
  - the updated version may have a **different behaviour**, e.g. first 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.
- Two main approaches:
  - make a library/tool/framework for an existing programming language;
  - make a new programming language.
- **Question:** What is the difference between the two approaches?

Framework example: Java RMI

- Objects can bridge to remote objects executing in other applications.
- Local execution:

```
Calculator calculator = new Calculator();
int sum = calculator.sum( 11, 2 );
System.out.println( sum );
```

- With Java RMI we can use a calculator from another remote application:

```
Registry registry = LocateRegistry.getRegistry(host);
Calculator calculator =
            (Calculator) registry.lookup("Calculator");

int sum = calculator.sum( 11, 2 );
System.out.println( sum );
```

- **Question:** what is nice about Java RMI?

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.
- 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 more **integration** with 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.