Module 626-1

# ARCHITECTURE DU SYSTÈME D'INFORMATION ET DESIGN PATTERNS

Dr. Stefan Behfar

Session 5

## Fallacies of distributed computing

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. *Transport cost is zero (time and money).*

[Wikipedia, based on a Peter Deutsch's paper in 1997]

## Consequence: prepare for failures

- Timeout
- Circuit breaker
- Use references
- Tolerant reader

## Timeout pattern

**Problem**: avoid waiting for ever if the service does not respond to request either because the connection is not available or the server is busy.

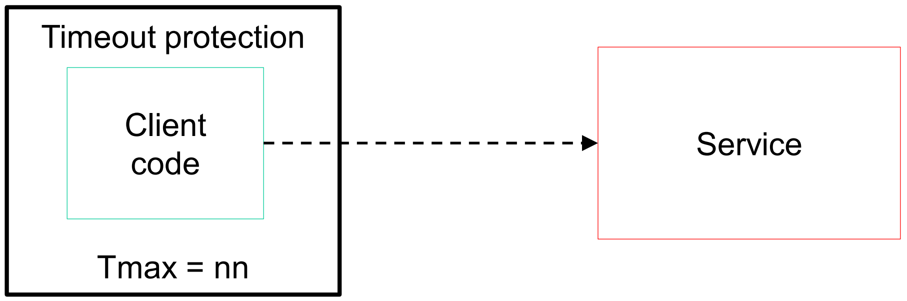
**Solution**: set the max time your client will wait before giving up. If timeout expired, perform some fault tolerance procedure

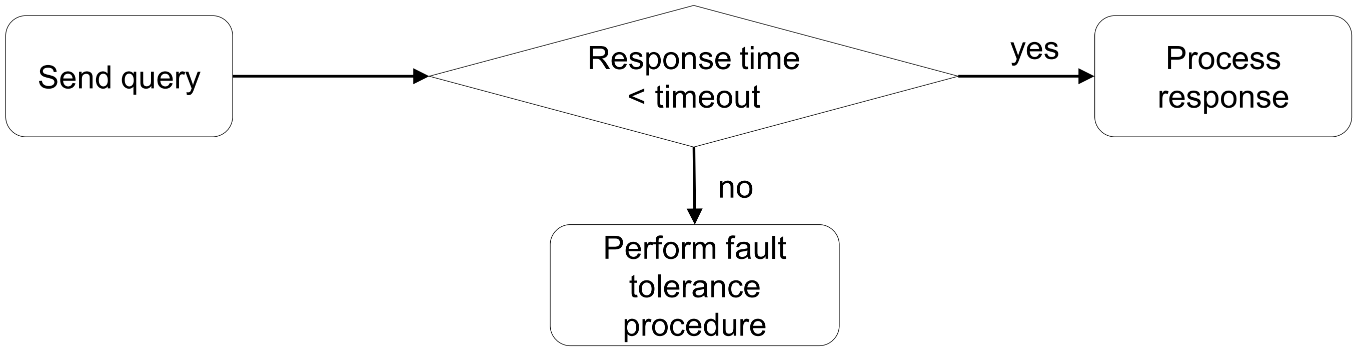
Network timeouts:

- Connection Request Timeout:  timeout in milliseconds when requesting a connection from the connection manager.
- Connect Timeout: timeout in milliseconds until a connection is established.
- Socket Timeout: timeout in milliseconds for waiting for data (i.e. reply from the server)

By default Jersey sets a connect or socket timeout of *0 millis*, meaning that the timeout is infinite.

# Timeout in pictures





# RequestConfig

The HttpClient is configured not to wait more than timeout period

- Implementation: RequestConfig object to be passed to the ClientBuilder factory to set up connection parameters to the HttpClient.

```
RequestConfig requestConfig = RequestConfig.custom()
            .setConnectionRequestTimeout(t1)
            .setConnectTimeout(t2)
            .setSocketTimeout(t3)
            .build();
HttpClient client = HttpClientBuilder.create().setDefaultRequestConfig(requestConfig).build();
```

t1, t2, t3 : time in milliseconds

## What if some timeout expires?

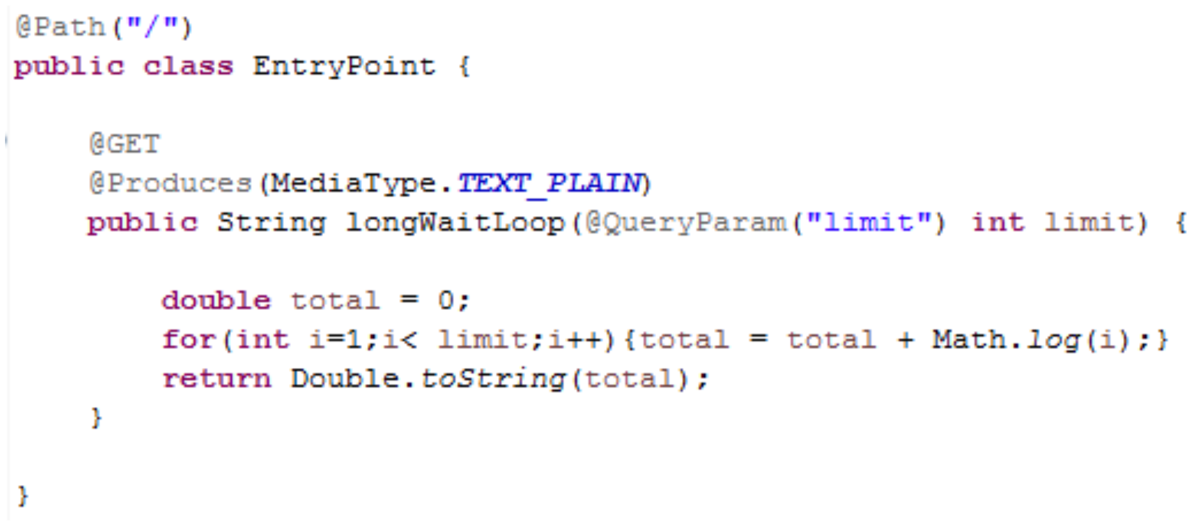
- Exceptions are raised:
  - SocketTimeoutException
  - ConnectTimeoutException

- Catch the exception to perform fault tolerance procedure

```
public static void main(String[] args) throws Exception {

    RequestConfig requestConfig = RequestConfig.custom()
            .setConnectionRequestTimeout(1)
            .setConnectTimeout(1)
            .setSocketTimeout(100).build();
    HttpClient client = HttpClientBuilder.create()
            .setDefaultRequestConfig(requestConfig).build();

    HttpGet request1 = new HttpGet("http://localhost:8080/HelloWorld/helloworld");

    try{
        HttpResponse response = client.execute(request1);
        print( response);
    }
    catch(Exception e){
        System.out.println("Timeout");
    }
}

public static void print(HttpResponse response) throws Exception {
    System.out.println(response.getStatusLine().getStatusCode());
    if(response.getStatusLine().getStatusCode() < 300) {
        BufferedReader rd = new BufferedReader(new InputStreamReader(response.getEntity()
                .getContent()));
        StringBuffer result = new StringBuffer();
        String line = "";
        while ((line = rd.readLine()) != null) result.append(line);
        System.out.println(result);
        }
}
```

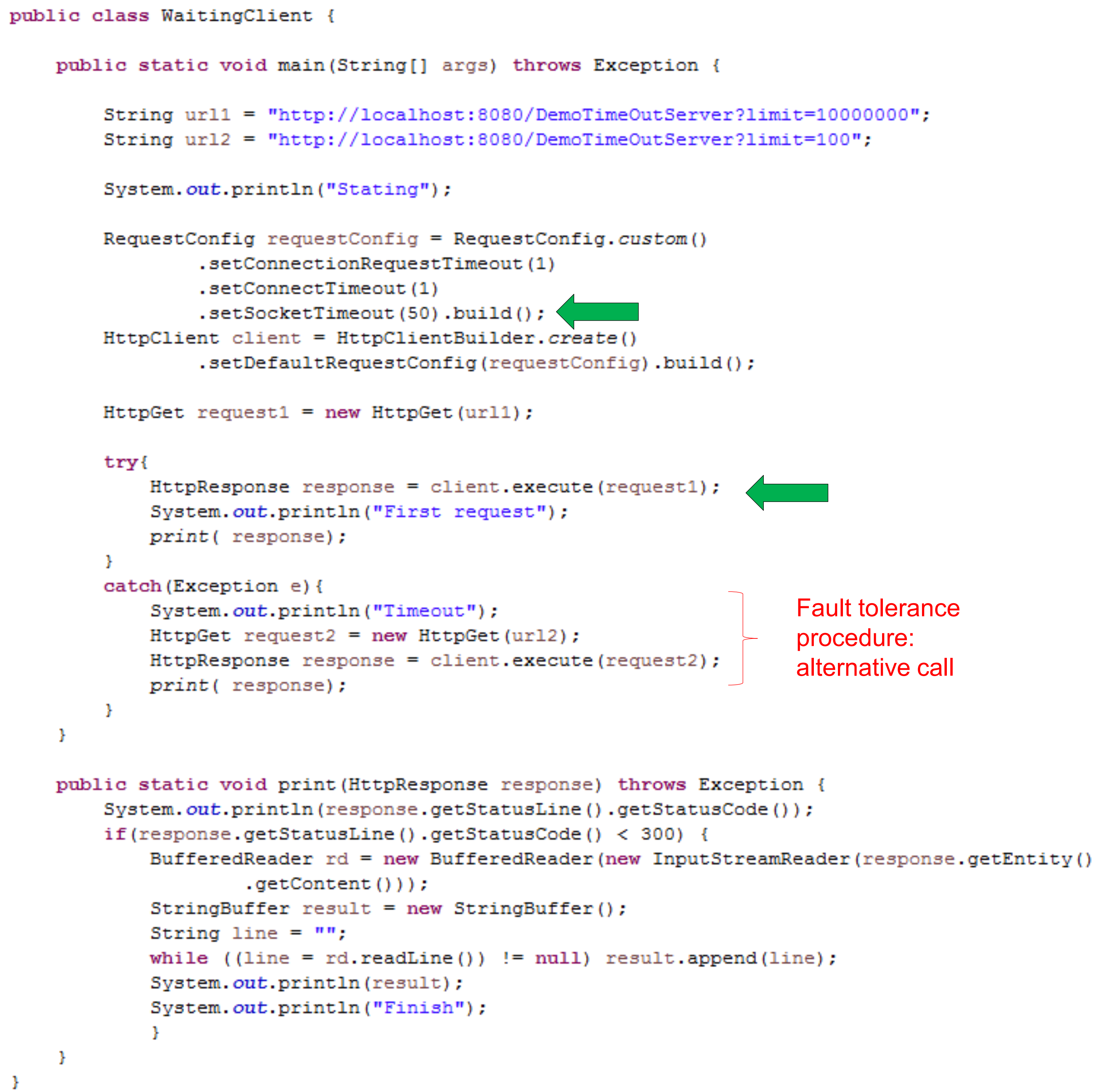
Fault tolerance procedure

# Exemple: slow services



```java
@Path("/")
public class EntryPoint {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String longWaitLoop(@QueryParam("limit") int limit) {

        double total = 0;
        for(int i=1;i< limit;i++){total = total + Math.log(i);}
        return Double.toString(total);
    }

}
```

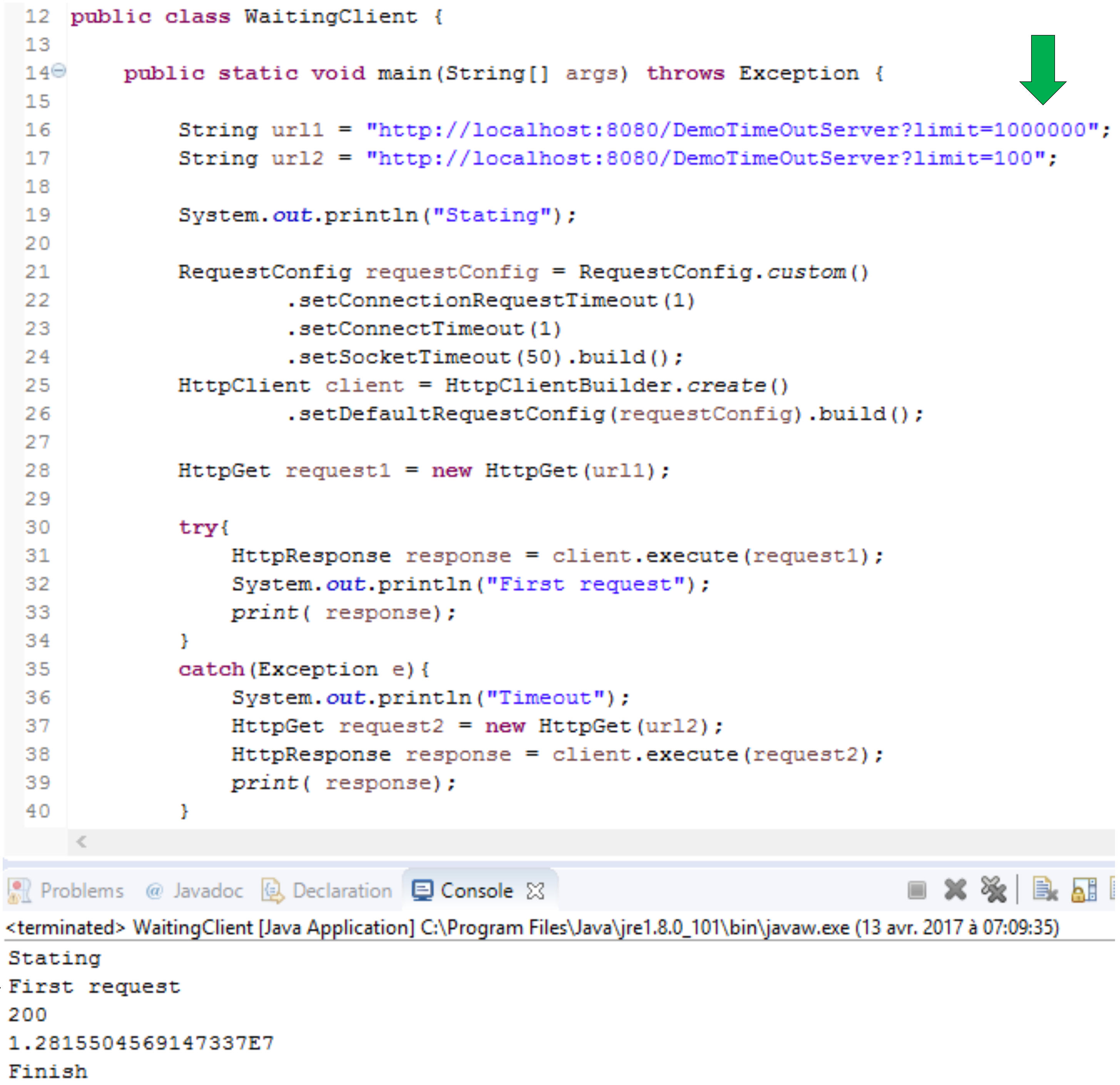
# Client for the slow service



```
public class WaitingClient {

    public static void main(String[] args) throws Exception {

        String url1 = "http://localhost:8080/DemoTimeOutServer?limit=10000000";
        String url2 = "http://localhost:8080/DemoTimeOutServer?limit=100";

        System.out.println("Stating");

        RequestConfig requestConfig = RequestConfig.custom()
                .setConnectionRequestTimeout(1)
                .setConnectTimeout(1)
                .setSocketTimeout(50).build();
        HttpClient client = HttpClientBuilder.create()
                .setDefaultRequestConfig(requestConfig).build();

        HttpGet request1 = new HttpGet(url1);

        try{
            HttpResponse response = client.execute(request1);
            System.out.println("First request");
            print( response);
        }
        catch(Exception e){
            System.out.println("Timeout");
            HttpGet request2 = new HttpGet(url2);
            HttpResponse response = client.execute(request2);
            print( response);
        }
    }

    public static void print(HttpResponse response) throws Exception {
        System.out.println(response.getStatusLine().getStatusCode());
        if(response.getStatusLine().getStatusCode() < 300) {
            BufferedReader rd = new BufferedReader(new InputStreamReader(response.getEntity()
                    .getContent()));
            StringBuffer result = new StringBuffer();
            String line = "";
            while ((line = rd.readLine()) != null) result.append(line);
            System.out.println(result);
            System.out.println("Finish");
            }
    }
}
```

Fault tolerance procedure: alternative call

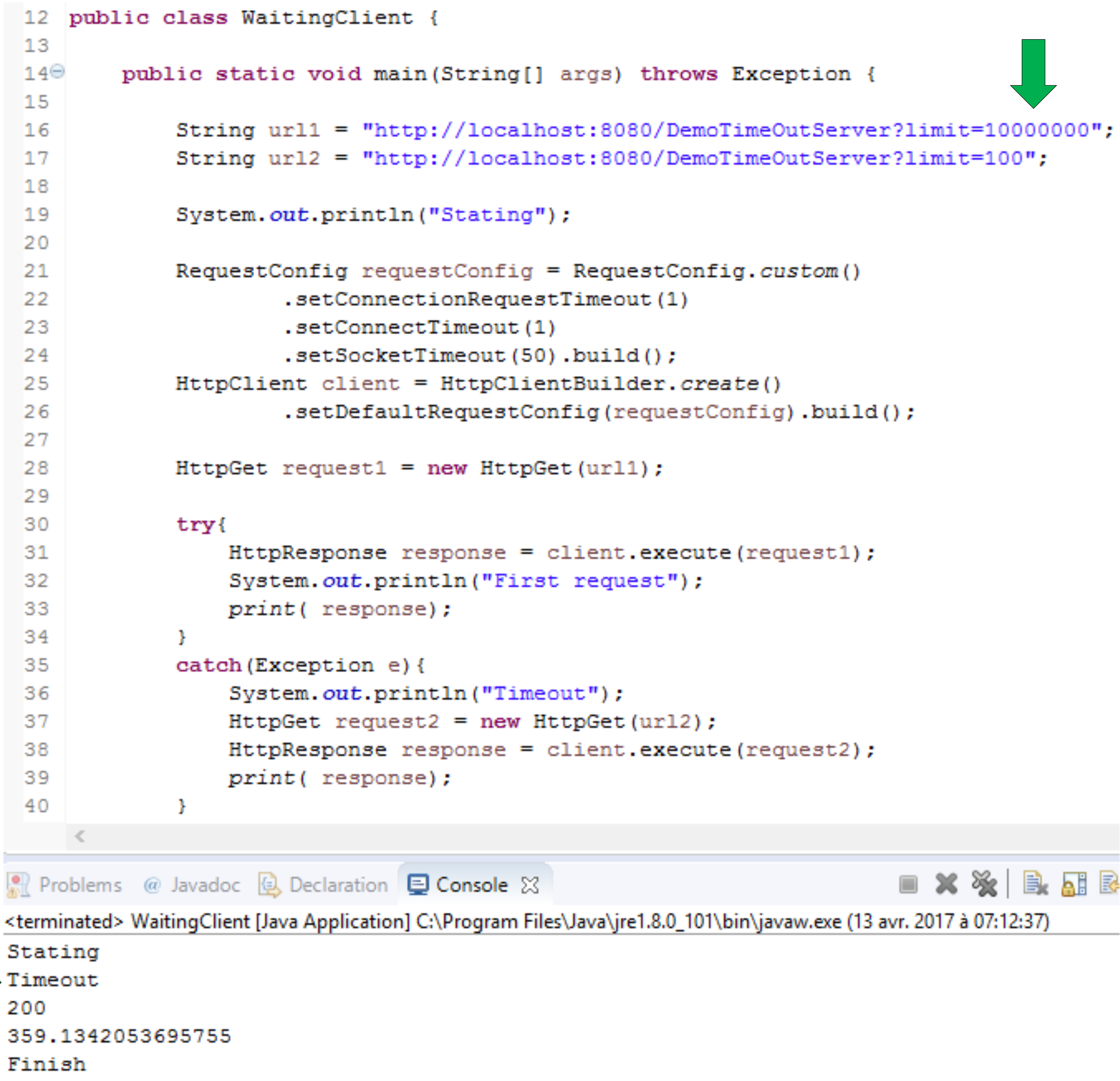
## Test1: Limit 1’000’000



```java
public class WaitingClient {

    public static void main(String[] args) throws Exception {

        String url1 = "http://localhost:8080/DemoTimeOutServer?limit=1000000";
        String url2 = "http://localhost:8080/DemoTimeOutServer?limit=100";

        System.out.println("Stating");

        RequestConfig requestConfig = RequestConfig.custom()
                .setConnectionRequestTimeout(1)
                .setConnectTimeout(1)
                .setSocketTimeout(50).build();
        HttpClient client = HttpClientBuilder.create()
                .setDefaultRequestConfig(requestConfig).build();

        HttpGet request1 = new HttpGet(url1);

        try{
            HttpResponse response = client.execute(request1);
            System.out.println("First request");
            print( response);
        }
        catch(Exception e){
            System.out.println("Timeout");
            HttpGet request2 = new HttpGet(url2);
            HttpResponse response = client.execute(request2);
            print( response);
        }
```

```
<terminated> WaitingClient [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\javaw.exe (13 avr. 2017 à 07:09:35)
Stating
First request
200
1.2815504569147337E7
Finish
```

## Test2: Limit 10’000’000



```java
public class WaitingClient {

    public static void main(String[] args) throws Exception {

        String url1 = "http://localhost:8080/DemoTimeOutServer?limit=10000000";
        String url2 = "http://localhost:8080/DemoTimeOutServer?limit=100";

        System.out.println("Stating");

        RequestConfig requestConfig = RequestConfig.custom()
                .setConnectionRequestTimeout(1)
                .setConnectTimeout(1)
                .setSocketTimeout(50).build();
        HttpClient client = HttpClientBuilder.create()
                .setDefaultRequestConfig(requestConfig).build();

        HttpGet request1 = new HttpGet(url1);

        try{
            HttpResponse response = client.execute(request1);
            System.out.println("First request");
            print( response);
        }
        catch(Exception e){
            System.out.println("Timeout");
            HttpGet request2 = new HttpGet(url2);
            HttpResponse response = client.execute(request2);
            print( response);
        }
```

<terminated> WaitingClient [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\javaw.exe (13 avr. 2017 à 07:12:37)

```
Stating
Timeout
200
359.1342053695755
Finish
```

## Demo

DemoTimeoutClient & DemoTimeOutServer

## Circuit Breaker

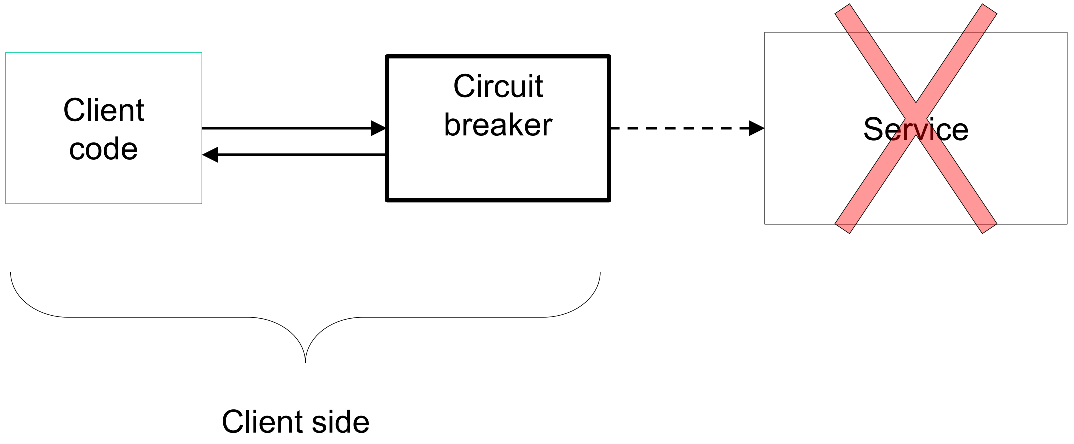
**Problem**: avoid calling some service, when we know it is unavailable

**Solution**: use a circuit breaker object that filters out requests to known unavailable service

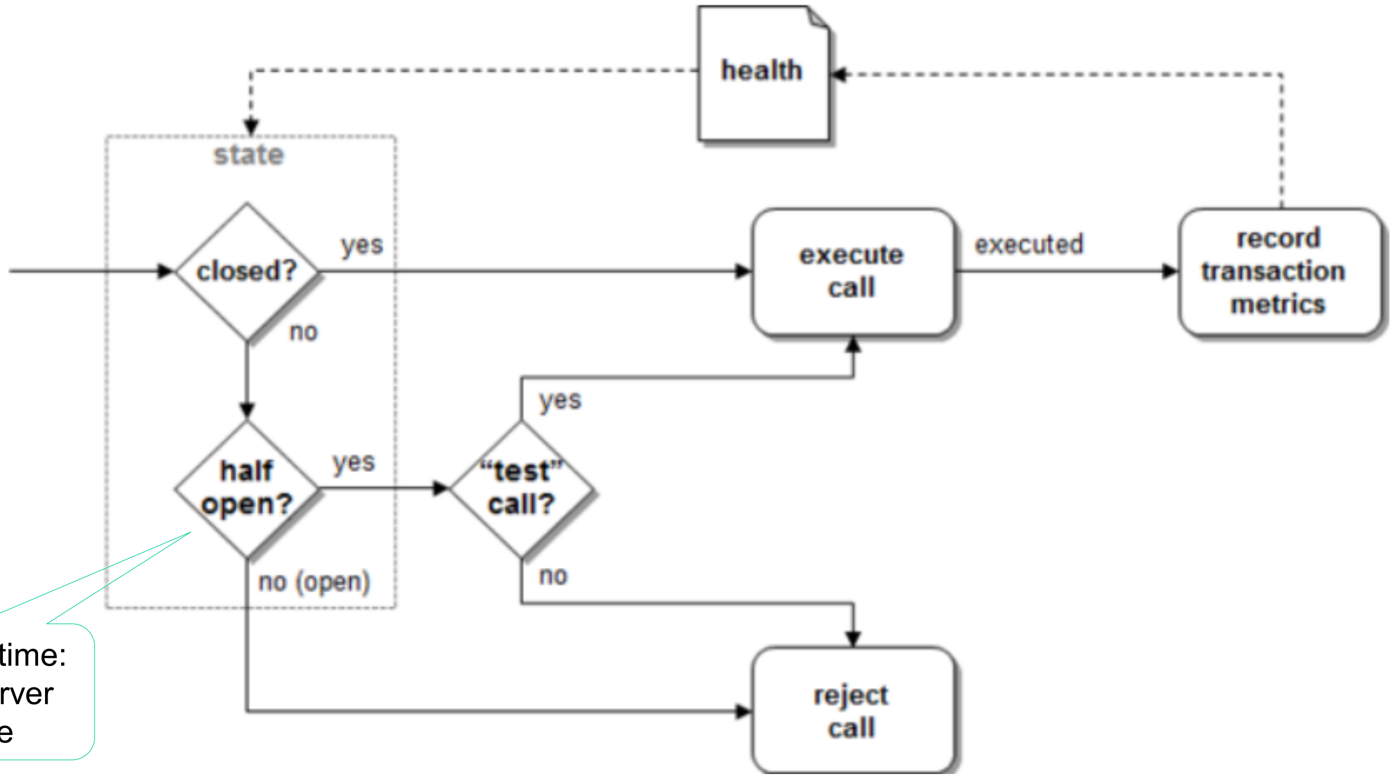
- When a failure is detected several times (must be recorded in some service health log), the breaker opens
- Next time the call returns immediately (this must be logged too), without calling the servicer and the client must perform some alternative work
- After some time, the circuit breaker will let the call go to the service to check for availability. If OK, the circuit breaker closes and will forward the next requests to the server

h e g

# Circuit Breaker in pictures



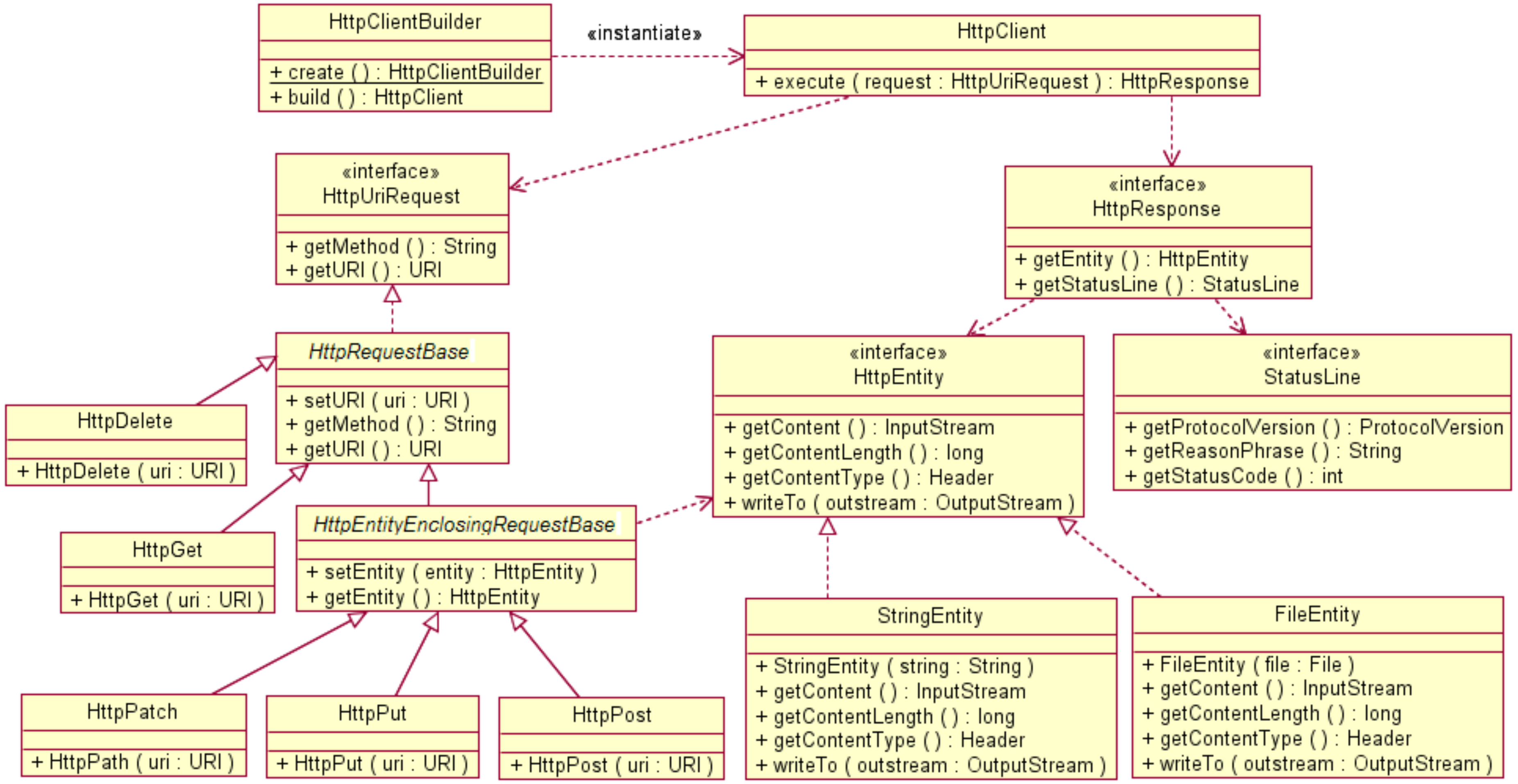
# Circuit breaker state graph



After some time: check if server available

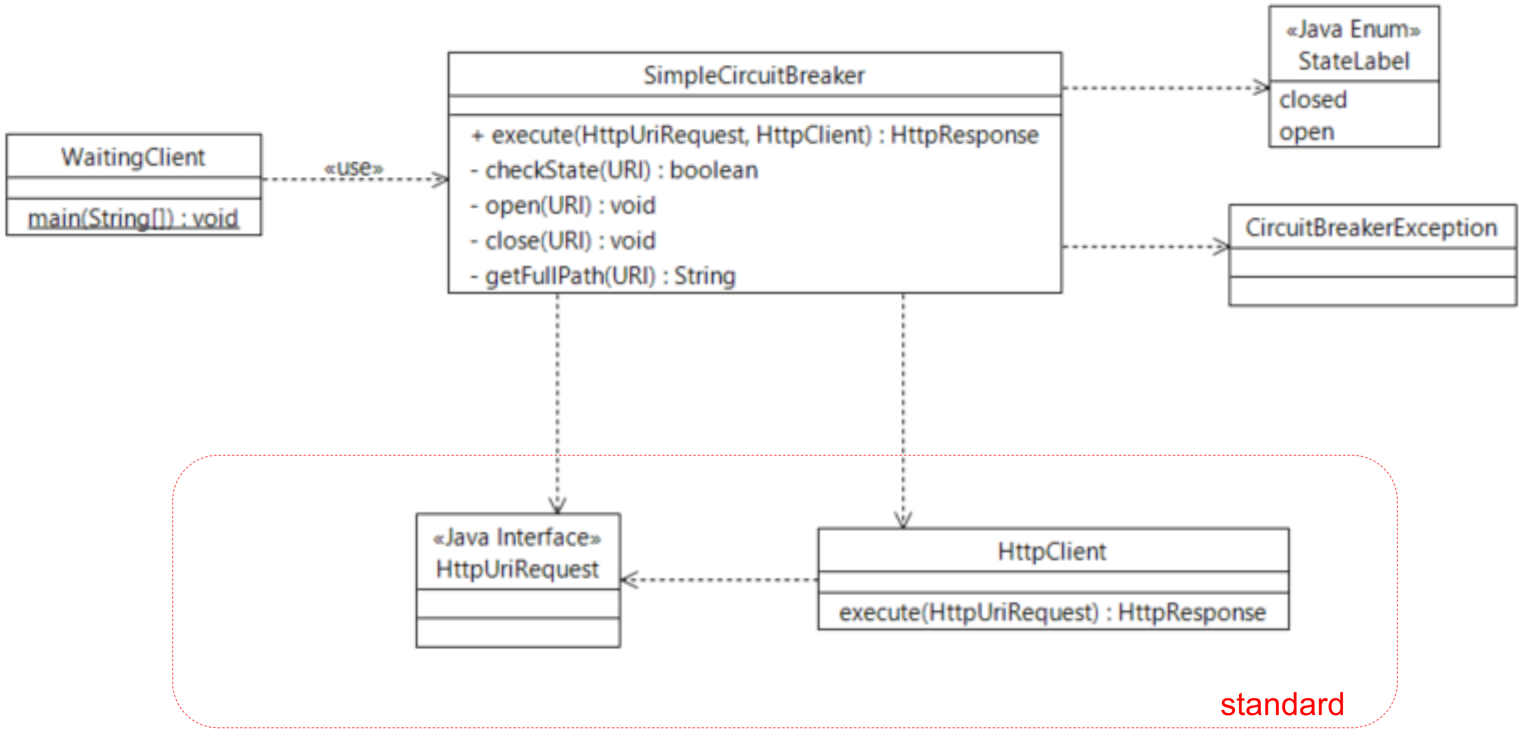
Source: http://www.javaworld.com/article/2824163/application-performance/stability-patterns-applied-in-a-restful-architecture.html?page=2
Accessed: 2017.05.10

# Reminder: main HTTP Client classes


HttpClientBuilder
+ create ( ) : HttpClientBuilder
+ build ( ) : HttpClient
«instantiate»
HttpClient
+ execute ( request : HttpUriRequest ) : HttpResponse
«interface» HttpUriRequest
+ getMethod ( ) : String
+ getURI ( ) : URI
«interface» HttpResponse
+ getEntity ( ) : HttpEntity
+ getStatusLine ( ) : StatusLine
HttpRequestBase
+ setURI ( uri : URI )
+ getMethod ( ) : String
+ getURI ( ) : URI
HttpDelete
+ HttpDelete ( uri : URI )
HttpGet
+ HttpGet ( uri : URI )
HttpEntityEnclosingRequestBase
+ setEntity ( entity : HttpEntity )
+ getEntity ( ) : HttpEntity
«interface» HttpEntity
+ getContent ( ) : InputStream
+ getContentLength ( ) : long
+ getContentType ( ) : Header
+ writeTo ( outstream : OutputStream )
«interface» StatusLine
+ getProtocolVersion ( ) : ProtocolVersion
+ getReasonPhrase ( ) : String
+ getStatusCode ( ) : int
HttpPatch
+ HttpPath ( uri : URI )
HttpPut
+ HttpPut ( uri : URI )
HttpPost
+ HttpPost ( uri : URI )
StringEntity
+ StringEntity ( string : String )
+ getContent ( ) : InputStream
+ getContentLength ( ) : long
+ getContentType ( ) : Header
+ writeTo ( outstream : OutputStream )
FileEntity
+ FileEntity ( file : File )
+ getContent ( ) : InputStream
+ getContentLength ( ) : long
+ getContentType ( ) : Header
+ writeTo ( outstream : OutputStream )

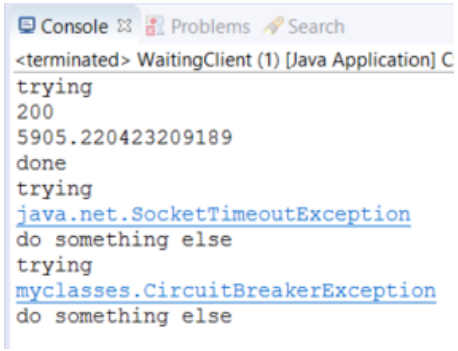
## Example

A simple version of circuit breaker will be demonstrated, with only 2 states: open and closed


WaitingClient
main(String[]) : void
«use»
SimpleCircuitBreaker
+ execute(HttpUriRequest, HttpClient) : HttpResponse
- checkState(URI) : boolean
- open(URI) : void
- close(URI) : void
- getFullPath(URI) : String
«Java Enum»
StateLabel
closed
open
CircuitBreakerException
«Java Interface»
HttpUriRequest
HttpClient
execute(HttpUriRequest) : HttpResponse
standard

# Client

```java
public class WaitingClient {

    public static void main(String[] args) throws Exception {

        String url1 = "http://localhost:8080/DemoTimeOutServer?limit=1000";
        String url2 = "http://localhost:8080/DemoTimeOutServer?limit=1000000000";
        SimpleCircuitBreaker.clear();

        RequestConfig requestConfig = RequestConfig.custom()
                .setConnectionRequestTimeout(1)
                .setConnectTimeout(1)
                .setSocketTimeout(50).build();
        HttpClient client = HttpClientBuilder.create()
                .setDefaultRequestConfig(requestConfig).build();

        int loop = 0;
        while(loop <3){
                try{
                    HttpGet request1;
                    if (loop == 0) request1 = new HttpGet(url1);
                    else request1 = new HttpGet(url2);
                    System.out.println("trying");
                    HttpResponse response = new SimpleCircuitBreaker().execute(request1, client);
                    print(response);
                    System.out.println("done");
                }
                catch(Exception e){
                    System.out.println(e.getClass().getName());
                    System.out.println("do something else");
                }
            loop++;
        }
    }
```



```
<terminated> WaitingClient (1) [Java Application] C
trying
200
5905.220423209189
done
trying
java.net.SocketTimeoutException
do something else
trying
myclasses.CircuitBreakerException
do something else
```

# SimpleCircuitBreaker

Server state
« dabatase »

```
public class SimpleCircuitBreaker {

    private static HashMap<String,StateLabel> serviceState = new HashMap<String,StateLabel>();

    //clears the URL state database
    public static void clear(){serviceState = new HashMap<String,StateLabel>();}

    public HttpResponse execute(HttpUriRequest request, HttpClient client) throws Exception{
        URI uri = request.getURI();
        HttpResponse response = null;
        if(checkState(uri)){
            try{
                response = client.execute(request);
                if(response.getStatusLine().getStatusCode()>= 500) open(uri);
            }
            catch(Exception e){
                open(uri);
                throw e;
                }
        }
        else throw new CircuitBreakerException();

        return response;
    }
```

## SimpleCircuitBreaker II

```java
//returns true if service can be reached
private boolean checkState(URI uri){
    String path = getFullPath(uri);
    if(serviceState.containsKey(path))
        return serviceState.get(path).equals(StateLabel.closed);
    else return true;
}
//set the circuit breaker to open for the service
private void open(URI uri){
    String path = getFullPath(uri);
    serviceState.put(path,StateLabel.open);
}
//set the circuit breaker to close for the service
private void close(URI uri){
    String path = getFullPath(uri);
    serviceState.put(path,StateLabel.closed);
    }
//returns the full path of the uri (removing query parameters)
private String getFullPath(URI uri){
    String path = uri.toString();
    int index = path.indexOf("?");
    if(index > 1)
        path = path.substring(0,index);
    return path;
}
```

```java
public enum StateLabel {
    open,closed
}
```

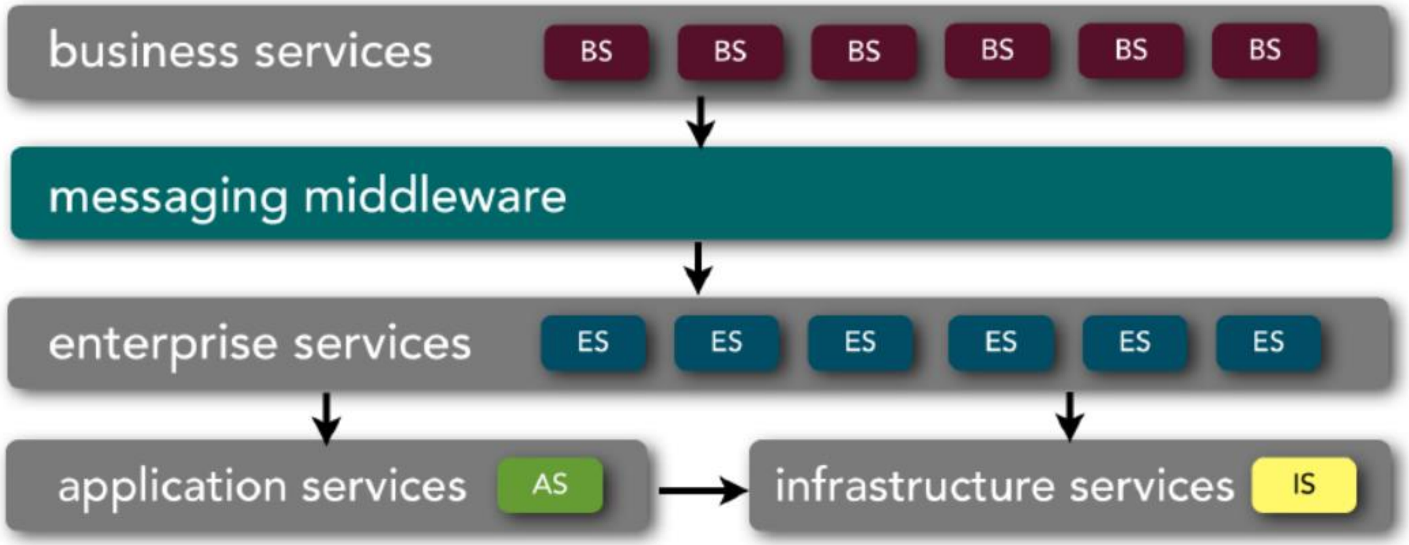
## Demo

DemoCircuitBreakerClient &
DemoTimeOutServer

## Services composition

## Once upon the time : SOA

- Attempt to architect large systems made from services
- Enterprise services were intended to be shared across the organization
- Enterprise architecture mindset when designing the system



[Richards M. - Microservices vs Service Oriented Architecture. O'Reilly, 2016]

## SOA

- Service components can range in size anywhere from small application services to very large enterprise services.

- It is common to have a service component within SOA represented by a large product or even a subsystem.

[Richards M. - Microservices vs Service Oriented Architecture. O'Reilly, 2016]

## Quality Attributes

- The key QA with SOA is **adaptability**
  - Through service composition, this architecture is intended to be adaptable to changes in the business process of a company
- But
  - Performance, scalability, deployability are not the primary QAs
- Nowadays SOA is much less talked about

So what’s next?

Microservices!

But is it again some new fashion word ?

h e g

## What is microservice

- Microservices is a specialisation of an implementation approach for service-oriented architectures (SOA) used to build flexible, independently deployable software systems.

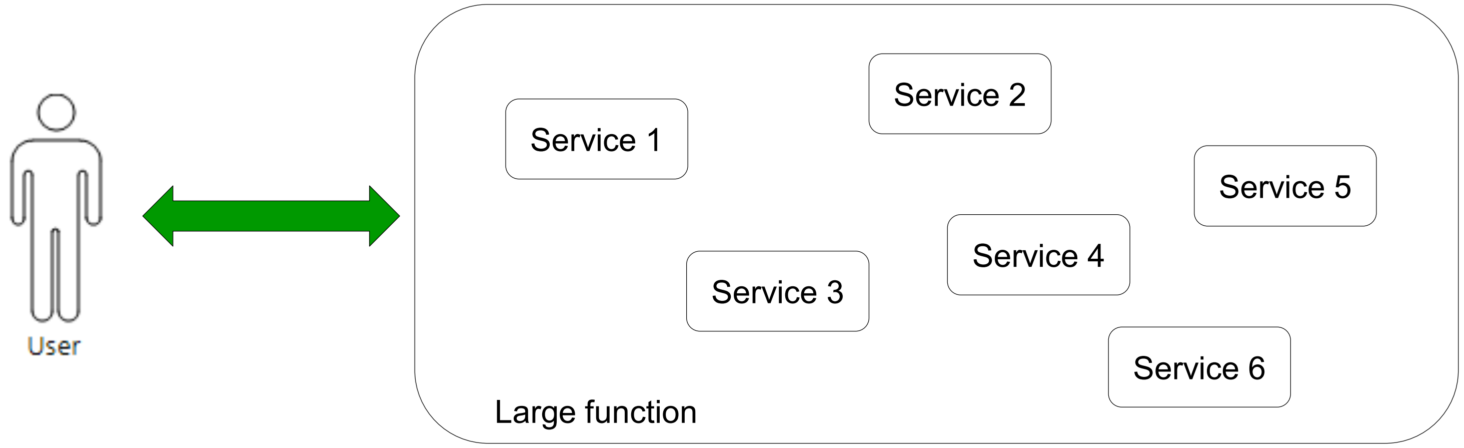
  [Wikipedia]

- The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.

[https://martinfowler.com/articles/microservices.html#CharacteristicsOfAMicroserviceArchitecture]

## Constraint: large systems

- **Services** are used either in isolation or in groups to perform some larger task (composition)



- But mainframe's **transactions**, by definition, are not suited to build large business functions (not composable)

## What’s new with microservices?

Same QA as mainframe’s transactions

- Performance
- *Isolation (beware of ACID constraints)*
- *Changeability  (but constraints on the data model)*

New QA’s for a new world

- Scalability
- Independance
- Deployability
- Changeability

(~~ACID~~)

## QAs

- Scalability
  - If one of the feature of the program is heavily used, one should be able to scale up the resources
- Independence
  - Services should not interact with each other. A change in one service should not have an impact on other services
- Deployability
  - Services must be independently deployable on different VMs
- Changeability
  - Services should be easily changeable without impacting the rest of the system

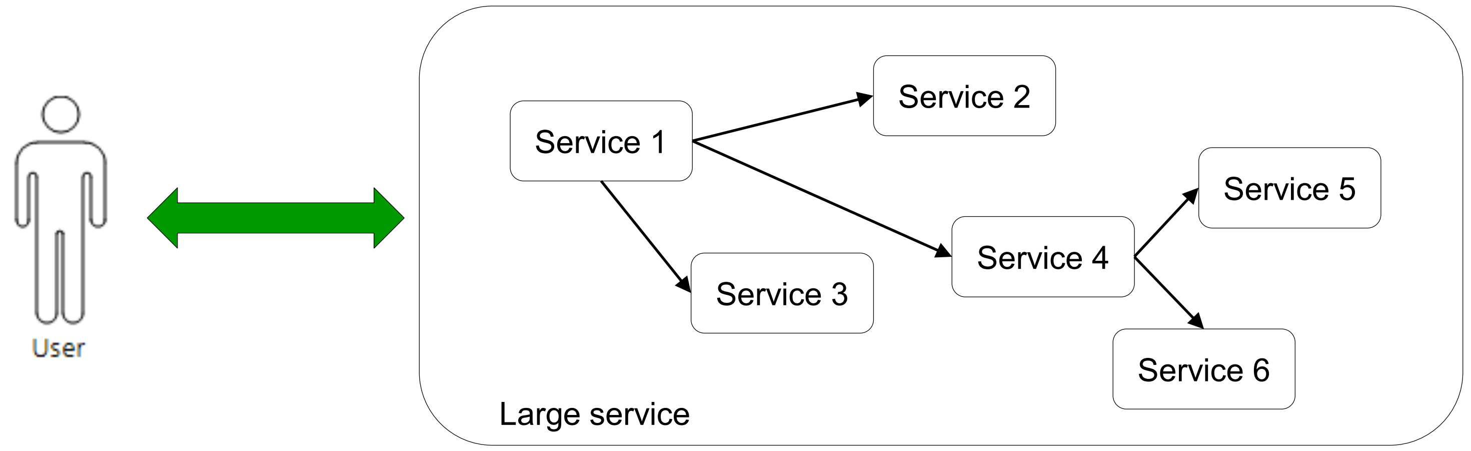
## Each Microservice is independently called through the network

API: Message oriented (REST)

## Architecting large systems

We must build large services (business functions) from simpler ones

- What are the possible calling architectures ?

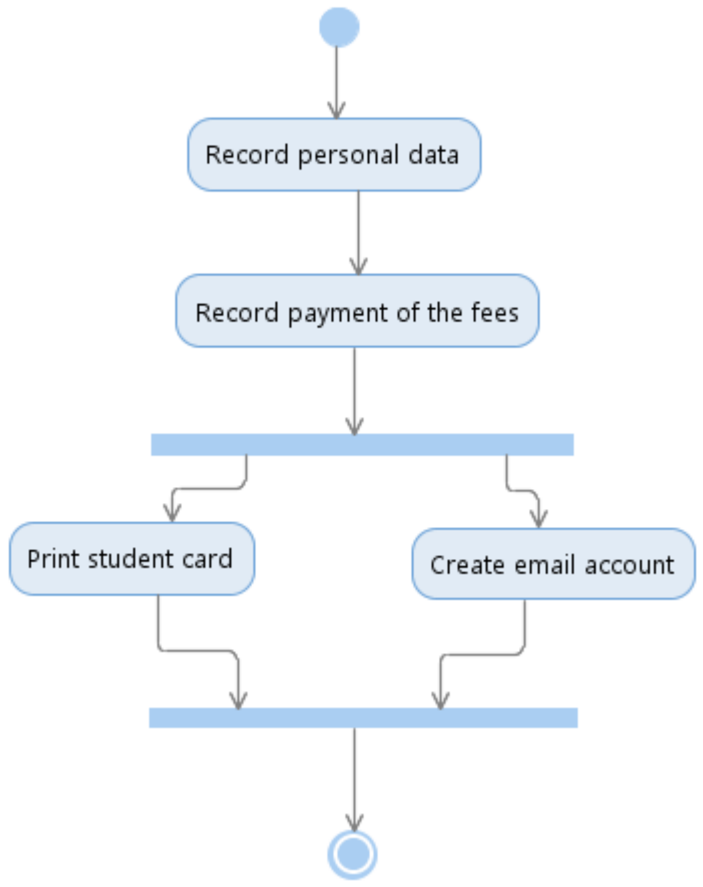


# Calling architecture: Orchestration and Choreography

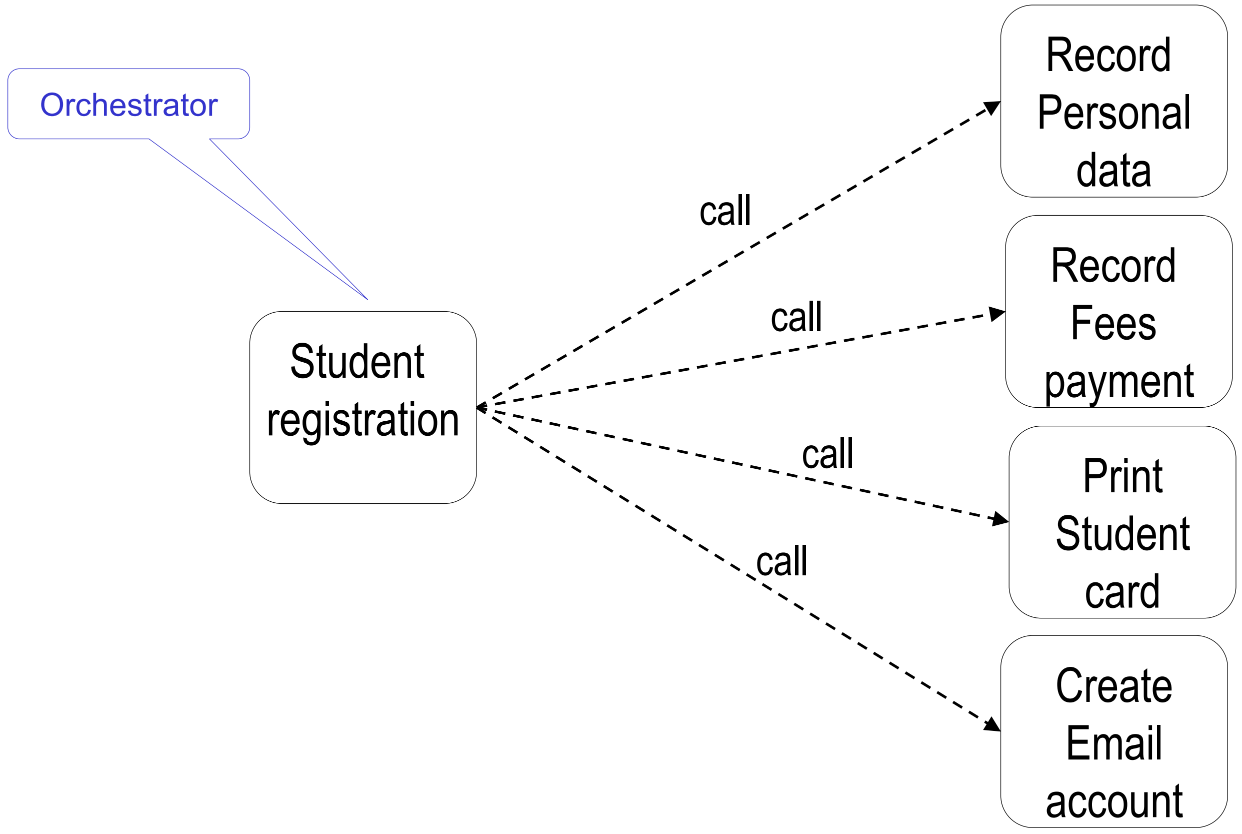
- Orchestration
  - A manager orchestrates the work of the other services to implement the business function /process
    - ✓Each service is explicitly called by the manager

- Choreography
  - Publish/subscribe kind of architecture : the scheduling is implicit in the sequence of event generated by each service.
    - ✓Each service registers to the event it must process.

# Example of a simple business process: enrollment of students

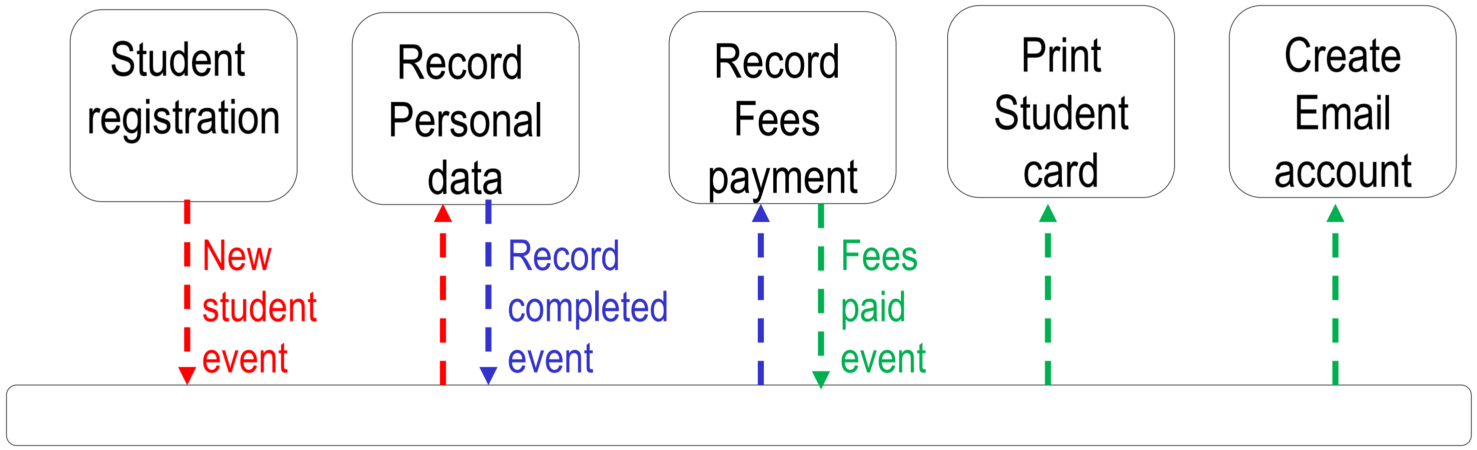


# Orchestration



- Good Auditability: central monitoring of the subtask's completion
- Extensibility: to call new services, we must adapt the orchestrator

# Choreography



The services must first subscribe to the proper events (publish/subscribe architecture)

- Auditability: harder to monitor the subtask's completion
- Good Extensibility: simple registration of the new component to the bus

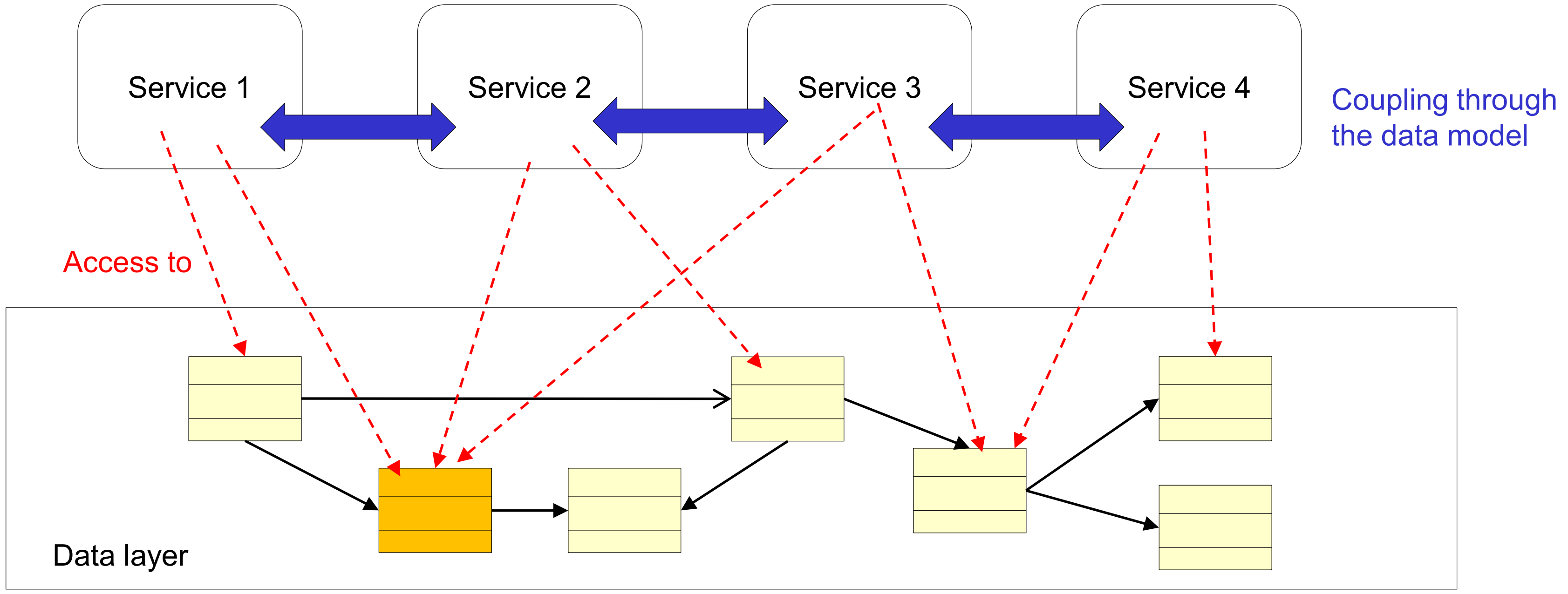
## Designing the microservices

## Traditional approach to system design

- Design of the data model
- Build the services that manage each part of the model

- But this will not comply with the independence & deployability QAs
  - The services will be linked through the data model
  - The service cannot be independently deployed

# Example of drawbacks



- ~~Independence~~
- ~~Scalability~~
- ~~Deployability~~

# Consequences

- ~~Independence~~
  - If the part of the data model required by service 1 is changed (dark yellow class) there will be an impact on other services
- ~~Deployability~~
  - Services are strongly linked to the data layer. How could they be independently deployed on several VMs?

## Learning

- Avoid data sharing among services for large systems based on microservices

- This leads to a new way to design systems:
    - Start designing systems based on capabilities, functionalities, services, not data.

# Findings about distributed data management

1. Many clients query the data while a few actually update it
   - Scaling happens mainly on the read service
   - But several updates may happen simultaneously.

2. The data people watch are not guaranteed to be up to date
   - When someone retrieves data, someone else may have updated it simultaneously. So data read may be partially obsolete. This is a fact in large distributed systems
     - ✓ Do not over engineer a solution to avoid obsolete data be displayed. The impact on QA will be too heavy

## Conclusion

1. All the services do not need the same access to the data (read / write)

2. Read services may need to display several parts of the data model while write services generally focus on a single one

3. The “Permanent data consistency” * constraint must be released for large distributed systems

*All data being consistent across all clients at all times