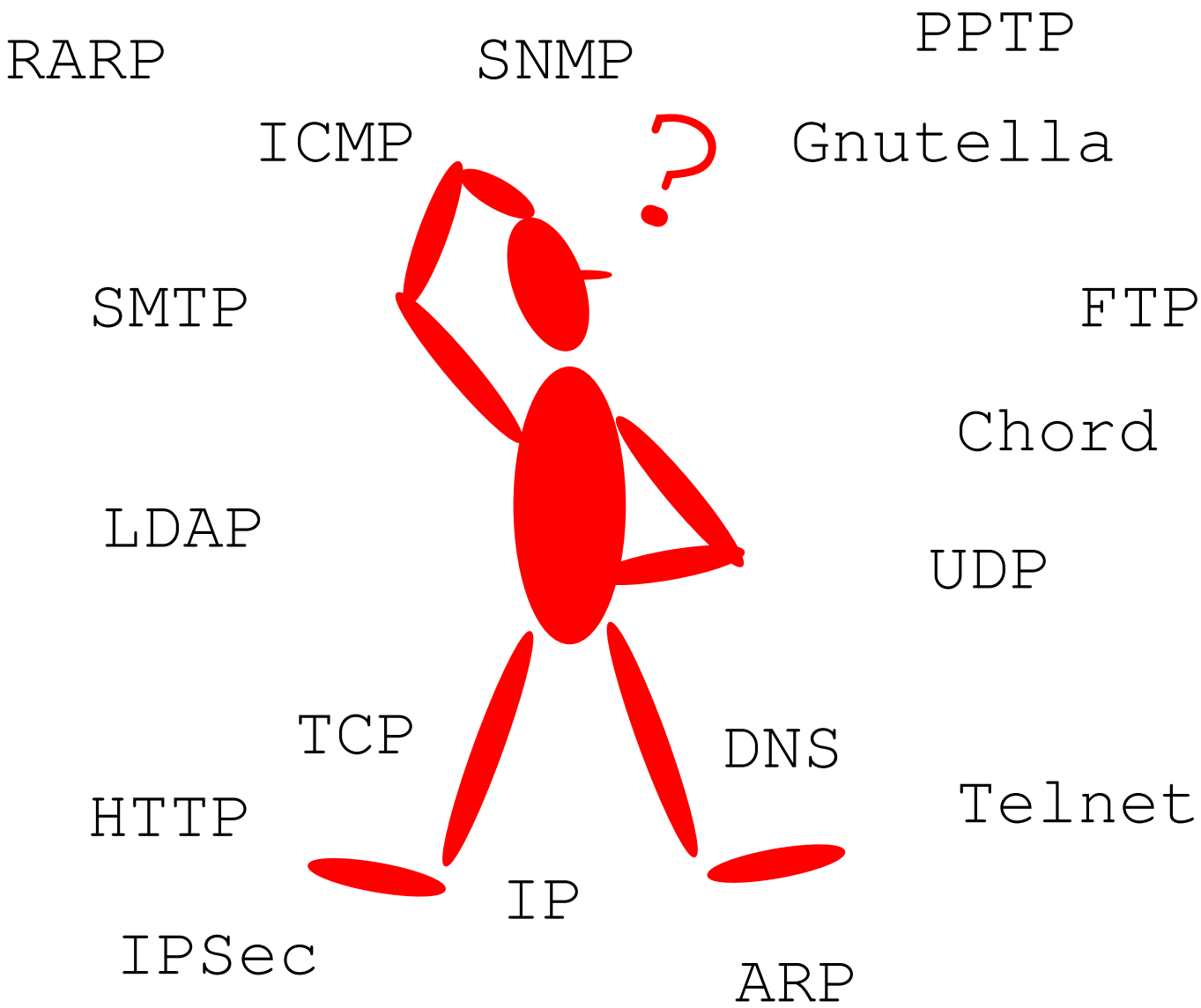
Saburo, a tool for I/O and concurrency management in servers

Gautier Loyauté Rémy Forax Gilles Roussel

Université de Marne la Vallée
Laboratoire d'Informatique de l'Institut Gaspard-Monge
UMR-CNRS 8049
F-77454 Marne la Vallée, Cedex 2, France

25 April 2006

More and more protocol



Internet constraints

More and more clients:

- Google: 250 millions queries per day

Increasing demands for:

- scalability
- minimization of latency
- maximisation of bandwidth

Interlace the handling of several requests

Servers more and more complex

Increasing demands for:

- effectiveness
- dynamicity
- dynamic variability

Induce an increasing number of errors:

- random behavior
- deadlocks
- livelocks

Need tools that helps the implementation

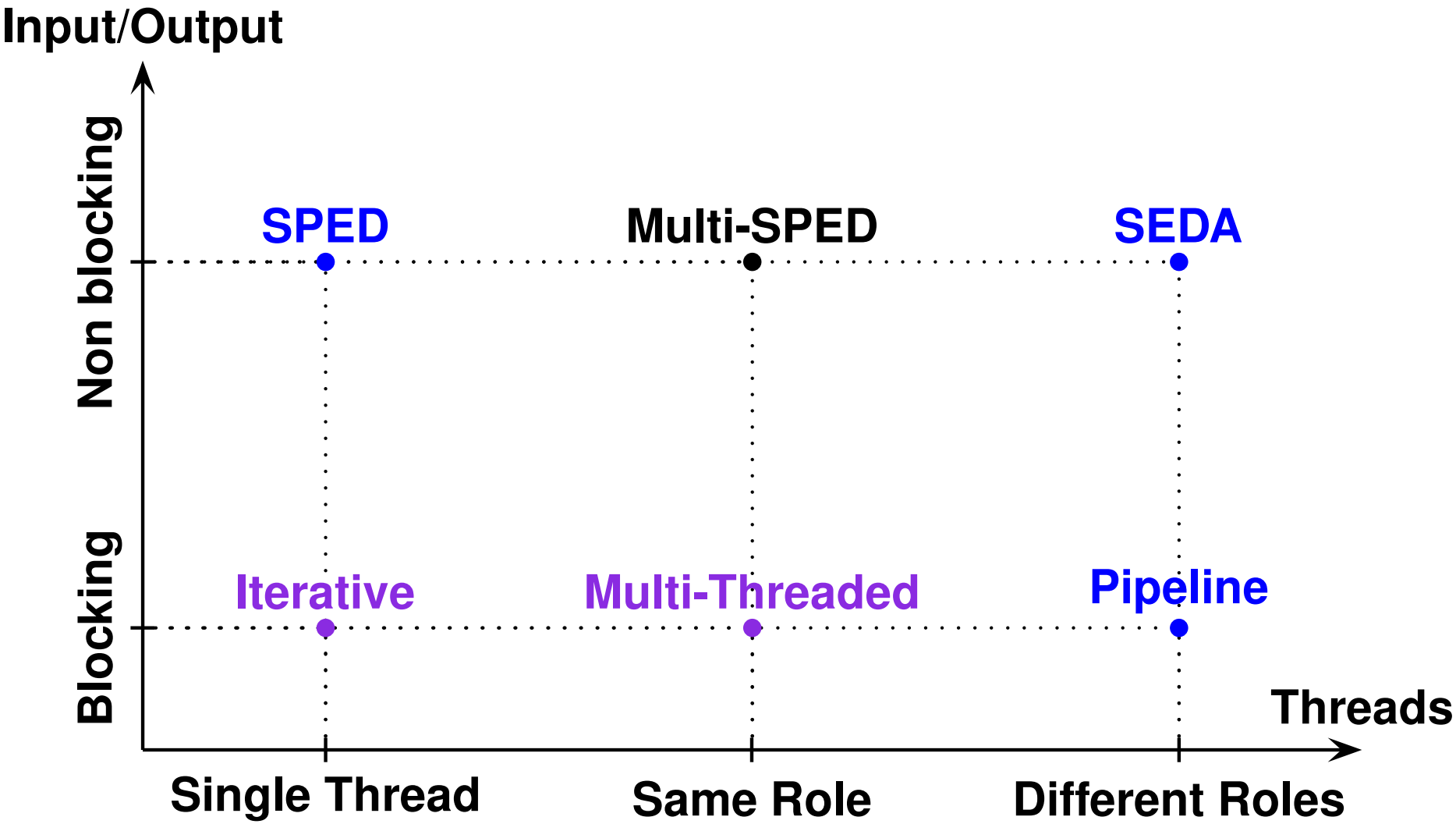
Concurrency

Interlacing the handling of several requests ?

Two phylosophies:

- **competition:** only one process is selected to handle a request
- **cooperation:** communication between processes in order to handle a request

Taxonomy of the server's architectures



No consensus on the best model

Description of Saburo

Directed graph: models the application

- Specified by developer using a UI
- Reusable code

Business code: stage (or vertex) of the graph

- Zero or one I/O call
- Sequence of instructions
- Specified by developer using Java
- Reusable code

Concurrency code: context (or edge) of the graph

- Channels between stages
  - method calls
  - local queues
  - sockets
- Generated

Development process illustration

Describe the implementation of a simple "**Echo**" server:

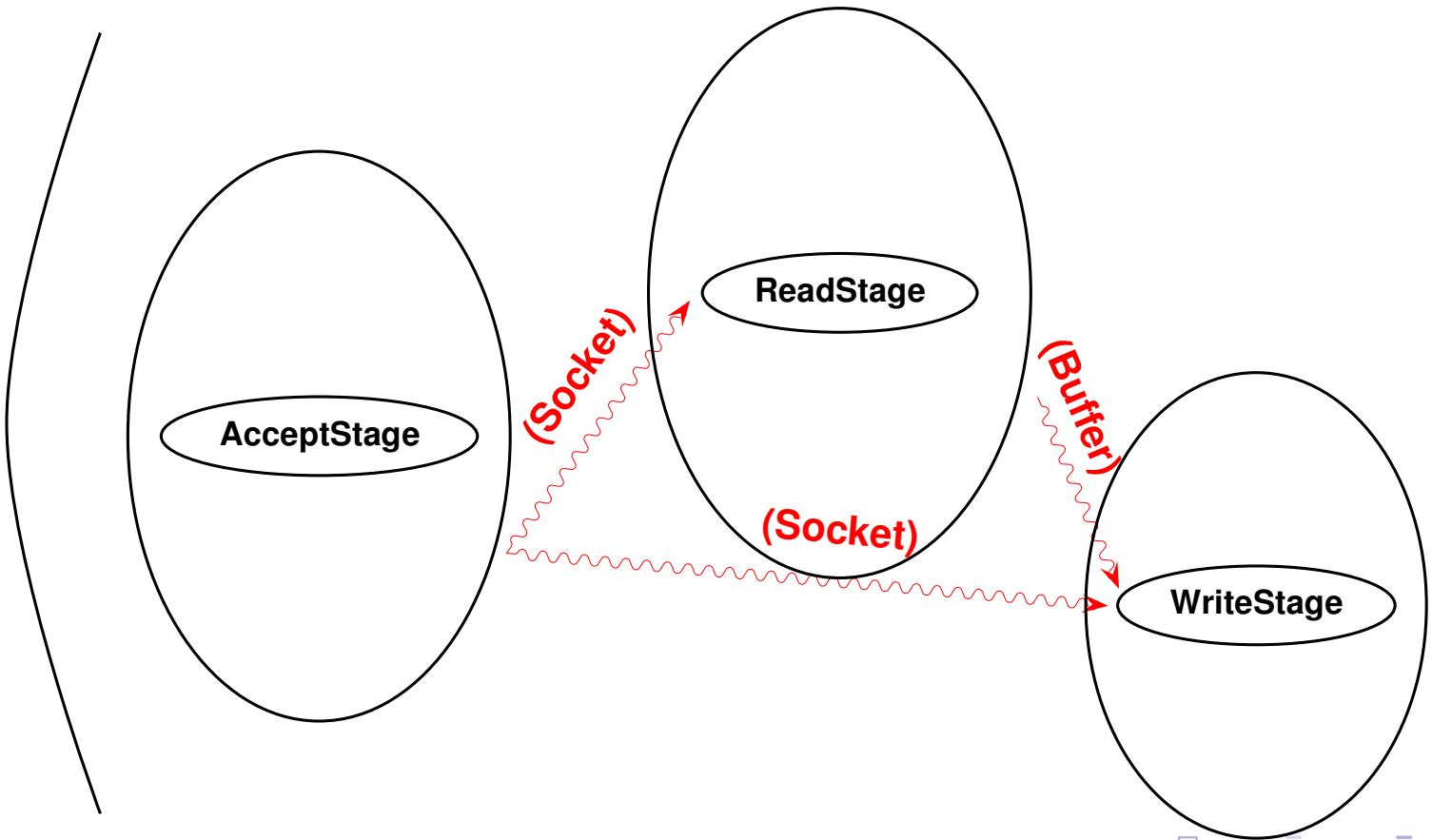
- The **Echo** graph modeled by the code below:


```
StageManagerImpl manager = new StageManagerImpl();
manager.connect(AcceptStage.class, ReadStage.class);
manager.connect(ReadStage.class, WriteStage.class);
```

Communications between stages

Defines input/output event interfaces for each stage:

- allow the communication between stages
- according to the position of a stage in the graph
- direct/centralized



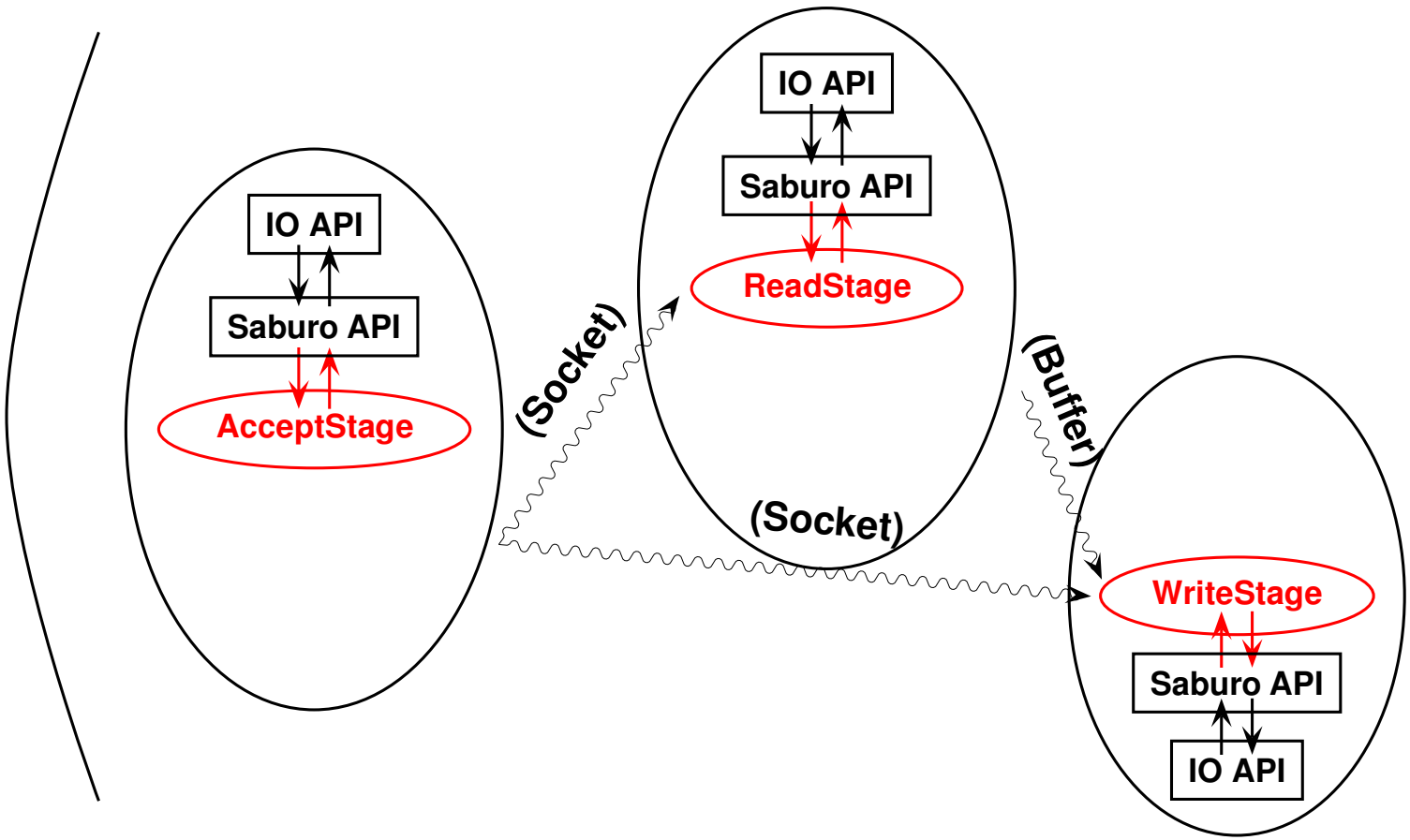
The **Echo**'s description of events


```
public interface OutputAcceptEvent {
 public void setAcceptSaburoSocket(SaburoSocket s);
}

public interface InputReadEvent {
 public SaburoSocket getAcceptSaburoSocket();
}

public interface OutputReadEvent {
 public void setReadByteBuffer(ByteBuffer b);
}

public interface InputWriteEvent {
 public SaburoSocket getAcceptSaburoSocket();
 public ByteBuffer getReadByteBuffer();
}
```

Description of stages

Implementation of the stages:

- *handle(...)* method: business code
- its parameters are the context and input/output events



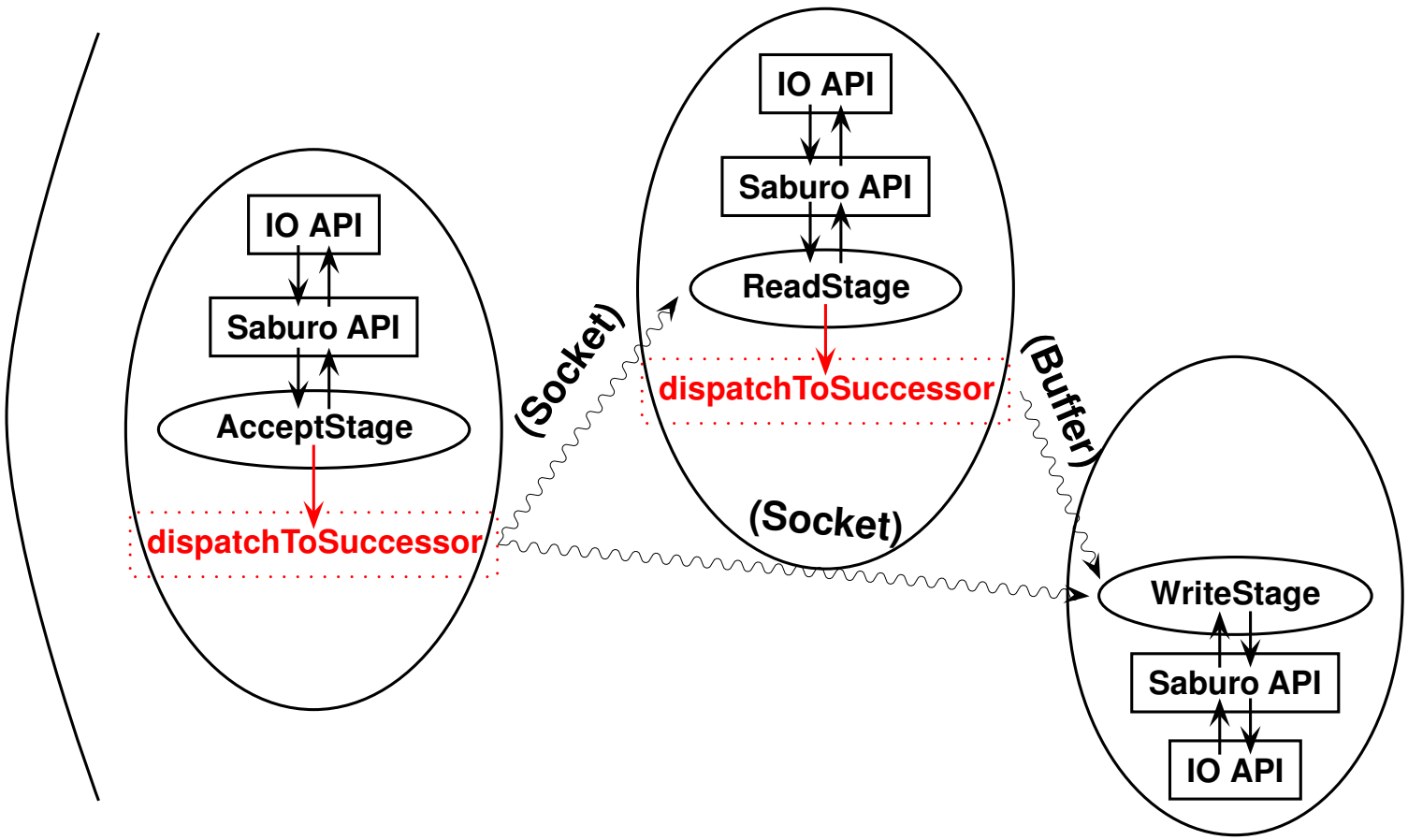
The **Echo**'s description of stages


```
public class ReadStage {
 public void handle(StageContext ctx,
                    InputReadEvent in,
                    OutputReadEvent out) {
  SaburoSocket client = in.getAcceptSaburoSocket();
  ByteBuffer buffer;

  while((buffer = client.read()) != null) {
   buffer.flip();
   // send to successor
  }
 }
}
```

Connection of stages

The context is the way to reach successor(s) in the graph:

- **according to the concurrency model**
- automatically generated (Java or bytecode)



The **Echo**'s description of stages


```
public class ReadStage {
 public void handle(StageContext ctx,
                    InputReadEvent in,
                    OutputReadEvent out) {
  SaburoSocket client = in.getAcceptSaburoSocket();
  ByteBuffer buffer = null;

  while((buffer = client.read()) != null) {
   buffer.flip();
   out.setReadByteBuffer(buffer); // <---
   ctx.dispatchToSuccessor(out);  // <---
  }
 }
}
```

Generation of the **Echo** concurrency: Iterative model

Generation of the ReadContext:

```
public class ReadContext implements StageContext{
  private WriteStage successor;

  public void dispatchToSuccessor(EchoEvent event){
    successor.handle(event);
  }
}
```

If there is only one process, the context is a function call

Generation of the **Iterative model**:

```
public class IterativeModel {
  public void service() throws Exception {
    while(true)
      acceptStageWrapper.handle();
  }
}
```

Generation of the **Echo** concurrency: Seda model

Generation of the ReadContext:

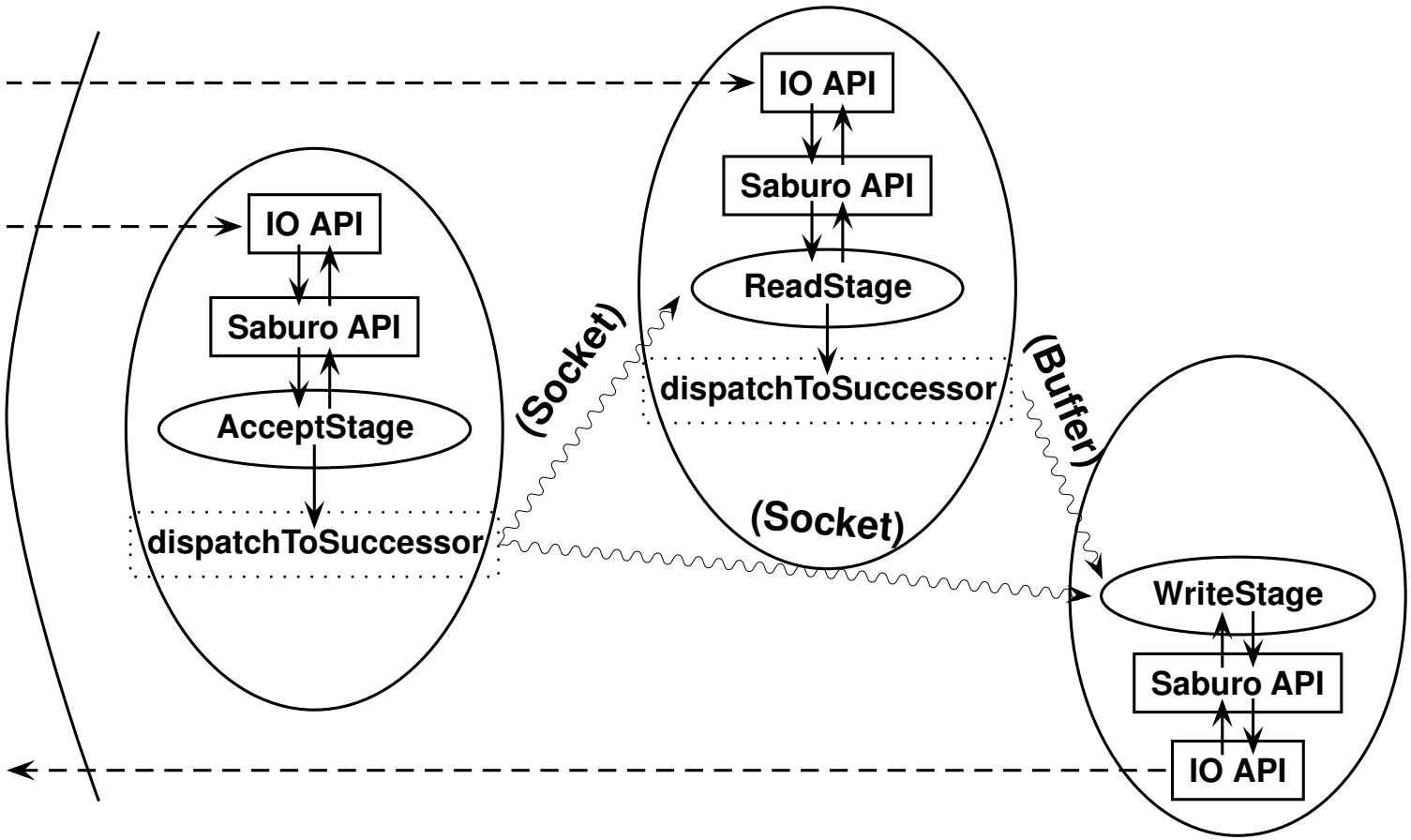
```
public class ReadContext implements StageContext{
  private WriteQueue successor;

  public void dispatchToSuccessor(EchoEvent event){
    successor.pushInQueue(event);
  }
}
```

If several processes, we use queues to implement it

Generation of the **Echo** concurrency: Seda model

```
public class SedaModel {
  public void service() throws Exception {
    new Thread(new Runnable() {
      public void run() {
        while(true) {
          writeSelector.doSelect();
        }
      }
    }).start();

    new Thread(new Runnable() {
      public void run() {
        while(true) {
          readSelector.doSelect();
        }
      }
    }).start();

    while(true) {
      acceptSelector.doSelect();
    }
  }
```

Finally the "**Echo**" server



Development process steps

Input / Output interfaces	specified in Java by user
Events	generated from interfaces
Functionnal code of a stage	specified in Java by user
Technical code of a stage	generated from concurrency
Stages's connection	specified by user using UI
Select the concurrency	
Concurrency	generated from concurrency

Summary

Weak interlacing between business code and concurrency:

- separation of concerns + code generation

Switch easily between different concurrent models:

- select the model best adapted to underlying architecture
- at compile time or runtime

Extend very quickly applications:

- addition of vertices and edges in a graph
- at compile time or runtime

Specifications and code generations are 100% Java:

- ensure the portability of the applications

Future

Distributed applications:

- the context establishes the connection between peers

HTTP & non blocking parser

Static analysis the application:

- applying model checker such as SPIN
- detect deadlock
- unreachable states
- temporal properties